Practical Programming with Substructural Types

A dissertation presented
by

Jesse A. Tov

to the Faculty of the Graduate School
of the College of Computer and Information Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Northeastern University
Boston, Massachusetts
February, 2012

Abstract

Substructural logics remove from classical logic rules for reordering, duplication, or dropping of assumptions. Because propositions in such a logic may no longer be freely copied or ignored, this suggests understanding propositions in substructural logics as representing resources rather than truth. For the programming language designer, substructural logics thus provide a framework for considering type systems that can track the changing states of logical and physical resources.

While several substructural type systems have been proposed and implemented, many of these have targeted substructural types at a particular purpose, rather than offering them as a general facility. The more general substructural type systems have been theoretical in nature and too unwieldy for practical use. This dissertation presents the design of a general purpose language with substructural types, and discusses several language design problems that had to be solved in order to make substructural types useful in practice.

So design is a constant challenge to balance comfort with luxe, the practical with the desirable.

— Donna Karan

Acknowledgments

Thanks go first to my advisor, Riccardo Pucella, for his steady support and guidance. His suggestions and criticism have improved this dissertation in ways too numerous to list. I am also grateful to the rest of my thesis committee: Mitchell Wand, from whom I learned to think like a semanticist; Matthew Fluet, whose work prompted and shaped my own; and Matthias Felleisen, whose example and high expectations are an inspiration.

I owe much gratitude to the faculty and the present and former students at the Northeastern University College of Computer and Information Science, who together create a collegial, intellectual environment that I am sad to leave: Ahmed Abdelmeged, Jay Aslam, Dan Brown, Bryan Chadwick, Agnes Chan, Stephen Chang, Will Clinger, Richard Cobbe, Ryan Culpepper, Christos Dimoulas, Carl Eastlund, Felix Klock, Tony Garnock-Jones, Dave Herman, Ian Johnson, Karl Lieberherr, Olin Shivers, Vincent St-Amour, Paul Stansifer, Stevie Strickland, Asumu Takikawa, Sam Tobin-Hochstadt, Aaron Turon, David Van Horn, and Dimitris Vardoulakis. Alec Heller's insights and demands for clarity have been especially valuable.

This journey began when Norman Ramsey first introduced me to the study of programming languages as an academic discipline, and for that I am forever grateful. I would also like to thank Henry Leitner, Radhika Nagpal, Margo Seltzer, and Chris Thorpe, from whom I learned much about teaching; Chung-chieh Shan, who first told me to listen to the types; and my colleagues at FNMOC, Oleg Kiselyov and Andrew Pimlott.

Last but not in the least least, I would like to thank my family: Yaron, Shoshanah, Sarah, Michael, Maryann, Lau, Evelyn, and my infinitely patient wife, Elizabeth, without whom nothing is possible.

This research was supported in part by AFOSR grant FA9550-09-1-0110.

Contents

List of Figures

CHAPTER 1

Practical Substructural Types

IN THE LAST two decades, researchers have proposed a myriad of stateful type systems, in which types reflect and regiment the dynamic states of program resources. These type systems span a range from minimalistic models (Wadler 1992) to production-quality general-purpose programming languages (Brus et al. 1987). Many of the more theoretical systems (Wadler 1991; Bierman 1993; Benton 1995; Barber 1996; Morrisett et al. 2005; Ahmed et al. 2005) are based explicitly on Girard's linear logic (1987). Actual implemented programming languages, however, tend to support less general approaches to statefulness targeted at specific problems, such as *memory regions* and *typestate* for safety in low-level languages (DeLine and Fähndrich 2001; Grossman et al. 2002; Zhu and Xi 2005), *session types* for static checking of communication protocols (Fähndrich et al. 2006), or *security-oriented types* (Swamy et al. 2010).

These special-purpose type systems are often elegant and effective, but they are of little use to a programmer who wants to write a program using the *next* as-yet-uninvented stateful type system. However, many of the specific cases for statefulness are instances of a more general case: Given a *substructural type system*—which limits how many times some values may be used—and sufficiently flexible abstraction mechanisms, many of the special-purpose type systems can be expressed within the language. For example, rather than provide session types as a primitive language feature, session types can be programmed as a library in a language with substructural types.

If realized successfully, a general-purpose substructural type system can provide several benefits. First, it eliminates the need to design a new language

for each new stateful type discipline, and second, it allows for several different designs for a particular stateful type discipline within the same language. Together, these properties facilitate experimentation in stateful types. Third, a general-purpose substructural programming language can support different notions of stateful types within the same program, which is not possible if each stateful type system is confined to its own language. However, this flexibility comes with a trade-off, because a language designed with a particular state discipline in mind may be tuned specifically for that purpose, potentially making it easier to use than a general-purpose system. Thus, it is important to show that programming with a general-purpose substructural type system is not especially onerous.

This brings me to my thesis:

> *A programming language with general-purpose substructural types can be practical and expressive.*

By *substructural types*, I mean type systems that restrict the usual structural rules of *contraction* and *weakening* in order to control the number of times values may be used, in the style of Girard's linear logic (1987); in this thesis I focus mainly on *affine types,* which can prevent values from being used more than once. By *general-purpose substructural types,* I mean a type system in which substructural types are not devoted to the management of a particular kind of stateful resource, but available as a general mechanism for building program abstractions. By *practical,* I mean that the language offers a full complement of modern language features suitable for writing a wide range of programs, and that using the language is not inordinately difficult; by *expressive,* I mean that the language can express a variety of stateful resource disciplines found in the literature.

To support this thesis, I have developed *Alms,* a general-purpose programming language with affine types. Even the most elementary affine type system is sufficient to express a variety of stateful type disciplines, but the challenge in designing a language such as Alms is to make the resulting language practical. Solving this problem required the introduction of several novel type system features, such as *dereliction subtyping* and *dependent qualifier kinds,* which

promote reuse by allowing the same abstractions to apply to both affine and unlimited (non-affine) types. Also in pursuit of pragmatics, I had to come to grips with the interaction between substructural types and control effects such as exceptions, and practical concerns led me to consider a mechanism for the safe interaction between code written in a new, affine language such as Alms and a similar but conventional (non-affine) language. In each case, I developed a formal model to validate the soundness of language design ideas motivated by the pragmatics of substructural types.

1.1 The Structure of This Dissertation

In this dissertation, I introduce the Alms programming language informally, give a formal model of its semantics, and describe its implementation. I relate Alms to prior work by others and discuss how the design of Alms is influenced by that prior work.

In **chapter 2**, I survey stateful type systems; I show how substructural types can express a variety of stateful type disciplines, but also highlight the extent to which the resulting interfaces are awkward. **Chapter 3** describes Alms, a programming language with affine types, and introduces its features in a series of examples. In **chapter 4**, I revisit the examples from chapter 2 in Alms, demonstrating its expressiveness and elegance.

In **chapter 5**, I describe a model of Alms and prove two propositions, one about principal types and the other a syntactic type soundness theorem; this work, along with some of chapter 3, previously appeared the 2011 Symposium on Principles of Programming Languages (Tov and Pucella 2011b). Additional proofs for chapter 5 appear in appendix A. **Chapter 6** describes the implementation of Alms, focusing on type inference.

The next two chapters describe and formalize interactions of substructural types with other language features. **Chapter 7**, which originally appeared in the 2010 European Symposium on Programming (Tov and Pucella 2010), shows how a programming language with affine types can safely interact with a similar language that lacks affine types. **Chapter 8**, which originally appeared in the 2011 Conference on Object-Oriented Programming, Systems,

Languages and Applications (Tov and Pucella 2011a), explores the relationship between substructural types and control effects, and proposes a type-and-effect system to support their safe interaction. Additional proofs for these chapters appear in appendices B and C, respectively.

In **chapter 9**, I compare Alms to related work and show how some of that work influenced the design of Alms. Finally, I propose some future work and conclude in **chapter 10**.

CHAPTER 2

Background: Stateful Type Systems

IN A STATEFUL type system, types reflect the dynamic state of resources and can be used to regulate the usage of such resources. In a weak sense, this is true of many type systems. For example, in OCaml (Leroy et al. 2011), the type of a reference cell indicates the type of the value in the corresponding store location, and the reference acts as a capability to read and write the location at the proper type. OCaml's state as reflected in its types is monotonic, in that the set of operations permitted increases monotonically over time. This chapter surveys type systems that are stateful in a stronger, non-monotonic sense: Operations admitted by the type system at one point in a program may be rejected at a later point.

In §2.1, I begin by introducing intuitionistic linear logic and several related type systems, which form the basis for much of this chapter. While these type systems are far from practical, I use them to demonstrate the expressiveness of substructural types. The subsequent three sections introduce type systems for managing specific kinds of stateful resources: objects with simple protocols (§2.2), manually allocated memory (§2.3), and communication channels with statically checked protocols (§2.4). In each section, I show how linear logic or one of its variants from §2.1 can be used to express the specific resource management discipline described in that section. The argument for expressiveness continues in chapter 4, where I show that these resource management disciplines are also expressible in Alms.

2.1 Substructural Logics and λ Calculi

Substructural logics arise from removing structural rules such as *weakening* and *contraction* from conventional logics:[1]

$$\text{IL-Weakening}\quad \frac{\Gamma \vdash \tau}{\Gamma, \sigma \vdash \tau} \qquad \text{IL-Contraction}\quad \frac{\Gamma, \sigma, \sigma \vdash \tau}{\Gamma, \sigma \vdash \tau}.$$

Weakening means that if we can deduce a result from some premises, then we can deduce the same result from the same premises along with additional, unused premises. Contraction means that any result deducible from some duplicated premise is also deducible from only one copy of the premise. (This only makes sense if we consider premises as sequences or bags, rather than mere sets.) By rejecting contraction, weakening, or both, we can understand propositions as standing for resources, which are conserved, rather than as arbitrarily duplicable or potentially irrelevant truths.

Girard (1987) noticed that an interesting thing happens when structural rules are removed. Consider these two versions of a conjunction introduction rule for intuitionistic logic:

$$\text{IL-}\wedge\text{I}\quad \frac{\Gamma \vdash \sigma \qquad \Gamma \vdash \tau}{\Gamma \vdash \sigma \wedge \tau} \qquad \text{IL-}\wedge\text{I}^\star\quad \frac{\Gamma \vdash \sigma \qquad \Delta \vdash \tau}{\Gamma, \Delta \vdash \sigma \wedge \tau}.$$

In the presence of weakening and contraction, these rules are interderivable, but in a substructural logic, conjunction splits into two different connectives, the additive conjunction ($\&$) and multiplicative conjunction ($\otimes$) of linear logic:

$$\text{ILL-}\&\text{I}\quad \frac{\Gamma \vdash \sigma \qquad \Gamma \vdash \tau}{\Gamma \vdash \sigma \,\&\, \tau} \qquad \text{ILL-}\otimes\text{I}\quad \frac{\Gamma \vdash \sigma \qquad \Delta \vdash \tau}{\Gamma, \Delta \vdash \sigma \otimes \tau}.$$

We can consider these linear connectives in light of the resource interpretation. In the derivation of the additive conjunction $\Gamma \vdash \sigma \,\&\, \tau$, resources σ and τ

[1] I use natural deduction throughout this section because it makes the transition to computational interpretations of the logics smoother. Similarly, metasyntactic variables are chosen with an eye toward type systems.

are constructed from the same resources Γ; whereas, in the multiplicative conjunction $\Gamma, \Delta \vdash \sigma \otimes \tau$, resources σ and τ are constructed from separate resources Γ and Δ. Elimination rules for both conjunctions appear along with the term assignment in figure 2.1.

Intuitionistic linear logic. In Girard's linear logic, several more connectives arise naturally from prohibiting weakening and contraction. For example, the usual disjunction ($\vee$) splits into additive ($\oplus$) and multiplicative ($\parr$) disjunctions. Additive disjunction admits a simple resource interpretation: $\sigma \oplus \tau$ is either resource σ or resource τ, as chosen by its two potential introduction rules. Multiplicative disjunction is harder to understand. While several computational interpretations involving concurrency and control have been proposed (Abramsky 1993; Mazurak and Zdancewic 2010), in this treatment I follow Bierman's (1993) development of intuitionistic linear logic (henceforth ILL), which omits $\parr$.

Instead of multiplicative disjunction, ILL includes as primitive the (multiplicative) linear implication ($\multimap$), which is a derived connective in Girard. It too has a simple resource interpretation: The linear implication $\sigma \multimap \tau$, from which linear logic gets its name, represents the ability to transform resource σ into resource τ without duplicating σ. ILL also includes three nullary connectives, which are units for the binary connectives: $\top$ for additive conjunction ($\&$), 0 for additive disjunction ($\oplus$), and 1 for multiplicative conjunction ($\otimes$).

The exponential. While the logic that results from rejecting weakening and contraction gives fine-grained control over resources, its inability to contend with durable "truth" renders it unacceptably weak. Girard (1987) remedies this problem by reintroducing weakening and contraction under controlled conditions, using the new *exponential* connectives ? and ! (only the latter of which appears in ILL). In the resource interpretation, $!\sigma$ represents the ability to produce zero, one, or more copies of resource σ. Such resources, which I will

call *unlimited*, admit weakening and contraction:[2]

$$\text{ILL-WEAKENING}\quad \frac{\Gamma \vdash \tau}{\Gamma, !\sigma \vdash \tau} \qquad \text{ILL-CONTRACTION}\quad \frac{\Gamma, !\sigma, !\sigma \vdash \tau}{\Gamma, !\sigma \vdash \tau}.$$

Girard shows that the addition of exponentials makes it possible to embed intuitionistic logic in linear logic, in particular by factoring the usual intuitionistic implication into linear implication and an exponential that allows dropping or duplicating the antecedent:

$$\sigma \to \tau \quad \equiv \quad !\sigma \multimap \tau.$$

Exponential introduction and elimination are of particular concern in this dissertation, as dealing with them cleanly is necessary for a practical substructural programming language. The rules are

$$\text{ILL-PROMOTION}\quad \frac{!\Gamma \vdash \tau}{!\Gamma \vdash !\tau} \qquad \text{ILL-DERELICTION}\quad \frac{\Gamma \vdash !\tau}{\Gamma \vdash \tau},$$

where $!\Gamma$ stands for a context $!\sigma_1, \ldots, !\sigma_k$ containing only unlimited resources. We can understand promotion to mean that if a resource is derivable from only unlimited resources, then that resource is also unlimited.[3] Dereliction means that, given some unlimited resource $!\tau$, we may "forget" that it is unlimited and obtain one copy of τ.

[2] Bierman (1993) uses more complicated versions of the weakening and contraction rules for his natural deduction formulation:

$$\text{ILL-WEAKENING}^\star\quad \frac{\Gamma \vdash !\sigma \qquad \Delta \vdash \tau}{\Gamma, \Delta \vdash \tau} \qquad \text{ILL-CONTRACTION}^\star\quad \frac{\Gamma \vdash !\sigma \qquad \Delta, !\sigma, !\sigma \vdash \tau}{\Gamma, \Delta \vdash \tau}.$$

I use his sequent calculus versions of the same rules, which are easily interderivable with these versions, because they smooth the transition to implicit weakening and contraction.

[3] Bierman also uses a more complicated version of the promotion rule:

$$\text{ILL-PROMOTION}^\star\quad \frac{\Gamma_1 \vdash !\sigma_1 \quad \cdots \quad \Gamma_k \vdash !\sigma_k \qquad !\sigma_1, \ldots, !\sigma_k \vdash \tau}{\Gamma_1, \ldots, \Gamma_k \vdash \tau}.$$

This is necessary for his natural deduction formulation to be closed under substitution, but when modeling a call-by-value language, we need to substitute only normal proofs, which makes the simple version of the promotion rule given here sufficient.

A term assignment. In order to consider ILL as a type system for a programming language, we require a term assignment. A term assignment based on Bierman's (1993), which uses terms similar to a core functional language, appears in figure 2.1.

2.1.1 Use Types and Standard Types

Wadler (1991) considers several changes to make ILL more suitable as a programming language. First, he makes the exponential rules (promotion, dereliction, contraction, and weakening) implicit rather than syntax-directed. Going further, however, he observes that dereliction is similar to a subtyping rule, whereby a term of type $!\sigma$ may be used where a term of type σ is expected. Thus, we may define a subtype relation for linear types that extends dereliction through other type constructors:

$$\frac{}{!\sigma <: \sigma} \qquad \frac{\sigma' <: \sigma \qquad \tau <: \tau'}{\sigma \multimap \tau <: \sigma' \multimap \tau'} \qquad \frac{\sigma <: \sigma' \qquad \tau <: \tau'}{\sigma \otimes \tau <: \sigma' \otimes \tau'}.$$

Then add a subsumption rule for terms:

$$\text{ILL-SUBSUME}\quad \frac{\Gamma \vdash e : \sigma \qquad \sigma <: \tau}{\Gamma \vdash e : \tau}.$$

It is straightforward to show that a derivation using subsumption can be translated to a derivation without subsumption, using the lemma that if $\sigma <: \tau$ then there exists a term e such that $\vdash e : \sigma \multimap \tau$.

Unfortunately, as Wadler points out, the resulting type system does not enjoy principal type schemes. Consider, for example, term $\lambda x.\lambda y.x\,y$. It has several types, including $!(\sigma \multimap \tau) \multimap !(\sigma \multimap \tau)$ and $(\sigma \multimap \tau) \multimap (\sigma \multimap \tau)$. However, the greatest lower bound of those two types under the proposed subtyping order, $(\sigma \multimap \tau) \multimap !(\sigma \multimap \tau)$, is not a type of term $\lambda x.\lambda y.x\,y$, nor is it even a theorem of ILL. Thus, the term does not have a principal type scheme in ILL-with-subtyping.

$\rho,\sigma,\tau ::= \top \mid 0 \mid 1 \mid \sigma \multimap \tau \mid \sigma \otimes \tau \mid \sigma \,\&\, \tau \mid \sigma \oplus \tau \mid !\sigma$ propositions

$x,y,z \in \mathit{Var}$ term variables

$e,f,g ::= x \mid \mathsf{discard}\, x \,\mathsf{in}\, e \mid \mathsf{copy}\, x \,\mathsf{as}\, y,z \,\mathsf{in}\, e \mid \mathsf{derelict}\, e$ proof terms
$\mid \mathsf{promote}\, e \mid \lambda x.e \mid e\,f \mid \langle e,f\rangle \mid \mathsf{let}\, \langle x,y\rangle = e \,\mathsf{in}\, f \mid [e,f] \mid \mathsf{fst}\, e \mid \mathsf{snd}\, e$
$\mid \mathsf{inl}\, e \mid \mathsf{inr}\, e \mid \mathsf{case}\, e \,\mathsf{of}\, \mathsf{inl}\, x \to f ; \mathsf{inr}\, y \to g \mid \mathsf{true} \mid \mathsf{false}\, e \mid \langle\rangle$
$\mid \mathsf{let}\, \langle\rangle = e \,\mathsf{in}\, f$

$\boxed{\Gamma \vdash e : \tau}$ *(intuitionistic linear logic)*

ILL-IDENTITY
$$\frac{}{x{:}\sigma \vdash x : \sigma}$$

ILL-WEAKENING
$$\frac{\Gamma \vdash e : \tau}{\Gamma, x{:}!\sigma \vdash \mathsf{discard}\, x \,\mathsf{in}\, e : \tau}$$

ILL-CONTRACTION
$$\frac{\Gamma, y{:}!\sigma, z{:}!\sigma \vdash e : \tau}{\Gamma, x{:}!\sigma \vdash \mathsf{copy}\, x \,\mathsf{as}\, y,z \,\mathsf{in}\, e : \tau}$$

ILL-DERELICTION
$$\frac{\Gamma \vdash e : !\sigma}{\Gamma \vdash \mathsf{derelict}\, e : \sigma}$$

ILL-PROMOTION
$$\frac{!\Gamma \vdash e : \sigma}{!\Gamma \vdash \mathsf{promote}\, e : !\sigma}$$

ILL-$\multimap$I
$$\frac{\Gamma, x{:}\sigma \vdash e : \tau}{\Gamma \vdash \lambda x.e : \sigma \multimap \tau}$$

ILL-$\multimap$E
$$\frac{\Gamma \vdash e : \sigma \multimap \tau \qquad \Delta \vdash f : \sigma}{\Gamma, \Delta \vdash e\,f : \tau}$$

ILL-$\otimes$I
$$\frac{\Gamma \vdash e : \sigma \qquad \Delta \vdash f : \tau}{\Gamma, \Delta \vdash \langle e,f\rangle : \sigma \otimes \tau}$$

ILL-$\otimes$E
$$\frac{\Gamma \vdash e : \sigma \otimes \tau \qquad \Delta, x{:}\sigma, y{:}\tau \vdash f : \rho}{\Gamma, \Delta \vdash \mathsf{let}\, \langle x,y\rangle = e \,\mathsf{in}\, f : \rho}$$

ILL-&I
$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma \vdash f : \tau}{\Gamma \vdash [e,f] : \sigma \,\&\, \tau}$$

ILL-&E$_1$
$$\frac{\Gamma \vdash e : \sigma \,\&\, \tau}{\Gamma \vdash \mathsf{fst}\, e : \sigma}$$

ILL-&E$_2$
$$\frac{\Gamma \vdash e : \sigma \,\&\, \tau}{\Gamma \vdash \mathsf{snd}\, e : \tau}$$

ILL-$\oplus$E
$$\frac{\Gamma \vdash e : \sigma \oplus \tau \qquad \Delta, x{:}\sigma \vdash f : \rho \qquad \Delta, y{:}\tau \vdash g : \rho}{\Gamma, \Delta \vdash \mathsf{case}\, e \,\mathsf{of}\, \mathsf{inl}\, x \to f ; \mathsf{inr}\, y \to g : \rho}$$

ILL-$\oplus$I$_1$
$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \mathsf{inl}\, e : \sigma \oplus \tau}$$

ILL-$\oplus$I$_2$
$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{inr}\, e : \sigma \oplus \tau}$$

ILL-$\top$I
$$\frac{}{\Gamma \vdash \mathsf{true} : \top}$$

ILL-0E
$$\frac{\Gamma \vdash e : 0}{\Gamma \vdash \mathsf{false}\, e : \tau}$$

ILL-1I
$$\frac{}{\vdash \langle\rangle : 1}$$

ILL-1E
$$\frac{\Gamma \vdash e : 1 \qquad \Delta \vdash f : \sigma}{\Gamma, \Delta \vdash \mathsf{let}\, \langle\rangle = e \,\mathsf{in}\, f : \sigma}$$

Figure 2.1: Term assignment for ILL

Use types. Principal types can be recovered, Wadler suggests, if there is some way to connect the presence or absence of the exponential on the domain and codomain of the type. To achieve this, Wadler introduces *use types,* which allow parametrizing over the presence of exponentials. A *use* is either a *use variable* or a constant 0 or 1, and the exponential is annotated with a use:

$$\begin{aligned} \mu, \nu &\in \mathit{UVar} && \text{use variables} \\ i, j, k &::= \mu \mid 0 \mid 1 && \text{uses} \\ \rho, \sigma, \tau &::= \cdots \mid !^i\tau && \text{use types} \end{aligned}$$

Then $!^1\sigma$ is like $!\sigma$ in ILL, and $!^0\sigma$ is merely σ.

Type schemes now include a constraint, which is a set of inequalities on uses, where $1 \geq 0$. Then $\lambda x.\, \lambda y.\, x\, y$ has the principal type scheme

$$!^i(\tau \multimap \sigma) \multimap !^j(\tau \multimap \sigma) \quad [i \geq j].$$

The two types given above for the term, as well as $!(\tau \multimap \sigma) \multimap \tau \multimap \sigma$, are instances of the scheme, while the incorrect type is not.

Standard types. Use types seem like a significant improvement over ILL, since they allow polymorphism of linearity. As a simplification, Wadler suggests that the syntax of use types be regularized as follows. Since the exponential is idempotent, the ability to repeat exponentials can result in more complicated types but does not increase expressiveness. Thus, Wadler suggests building the exponential into the syntax of types in only certain places:

$$\rho, \sigma, \tau ::= !^i\tau \multimap \sigma \mid !^i\tau \otimes !^j\sigma \mid !^i\tau \oplus !^j\sigma \qquad \text{standard types}$$

Wadler gives a type inference algorithm for standard types, asserts that it finds principal type schemes, and suggests what a proof might look like. Standard types are sound for linear types, in that if a term has a standard type, then all linear type instances of that standard type are also types of the term. Additionally, standard types are complete for linear types, in that if a term has a linear type, then it has a standard type of which that linear type is an instance.

2.1.2 Steadfast Types, λ^{URAL}, and Uniqueness Types

A common misconception about linear types is that a value of linear type cannot be aliased. A linear type system prevents aliasing of a value whose type is linear, but it does not actually guarantee the uniqueness of values of linear type, because of dereliction. In particular, the systems described in this chapter so far allow an unlimited value, which may already be aliased, to be derelicted to a linear type:

$$\frac{\Gamma \vdash e : !\sigma}{\Gamma \vdash e : \sigma} \qquad \frac{\Gamma \vdash e : !^i\sigma \qquad i \geq j}{\Gamma \vdash e : !^j\sigma}.$$

If we want better control of aliasing, rather than treating dereliction as subtyping, we might do away with dereliction altogether.

Steadfast standard types. Wadler (1991) suggests a way forward with steadfast (standard) types, in which the promotion and dereliction rules for exponential introduction and elimination are removed. Instead, promotion is rolled into the introduction rules for all type constructors, and dereliction is likewise performed by each elimination rule. For example, here are the introduction and elimination rules for the linear function type in Wadler's system of steadfast standard types:

SST-$\multimap$I
$$\frac{C; !^I\Gamma, x : !^i\sigma \vdash e : \tau}{C \wedge I \geq j; !^I\Gamma \vdash \lambda x.e : !^j(!^i\sigma \multimap \tau)}$$

SST-$\multimap$E
$$\frac{C; !^I\Gamma \vdash e : !^j(!^i\sigma \multimap \tau) \qquad D; !^J\Delta \vdash f : !^i\sigma}{C \wedge D; !^I\Gamma, !^J\Delta \vdash e\,f : \tau}.$$

(I and J are sets of uses, where $!^I\Gamma$ is an environment whose range contains types with uses I. C and D are constraints comprising use inequalities, where $I \geq j$ means that $i \geq j$ for each $i \in I$.) Rule SST-$\multimap$I includes promotion, because it gives the resulting function type a use based on the uses in the environment, which the promotion rule does for use types and ILL. Similarly, rule SST-$\multimap$E allows applying a function with any use on its type, which means there is no need for a separate dereliction step.

Thus, with steadfast standard types, the use on a type is determined when that type is introduced, never changes, and does not need to be removed by

dereliction before that type is eliminated. This guarantees that a value of linear type is not aliased, since it is now impossible to alias an unlimited value and then derelict it to get an aliased, linear value. This strong non-aliasing property of steadfast types means that when a heap-allocated value of linear type is eliminated in a steadfast system, it is guaranteed that there are no other pointers to the same heap value, which makes it safe to immediately free or reuse the memory at the point of elimination.

λ^{URAL}. A small change to the type structure of steadfast standard types yields Ahmed et al.'s (2005) λ^{URAL}, a polymorphic, substructural λ calculus with even finer control of resource usage. In λ^{URAL}, uses are replaced by substructural *qualifiers,* which determine which structural rules apply to a given type. Qualifiers include type variables and four qualifier constants: L, for *linear,* allows neither contraction nor weakening; A, for *affine,* allows weakening but not contraction; R, for *relevant,* allows contraction but not weakening; and U, for *unlimited,* allows both contraction and weakening. The available structural rules naturally induce a subsumption lattice on qualifiers.

In λ^{URAL}, a type is composed of a qualifier, which specifies which structural rules apply, and a *pretype*, which specifies the introduction and elimination rules:

$$
\begin{array}{rcll}
\alpha, \beta, \gamma & \in & \mathit{TVar} & \text{type variables} \\
\xi & ::= & \alpha \mid \mathsf{U} \mid \mathsf{R} \mid \mathsf{A} \mid \mathsf{L} & \text{qualifiers} \\
\overline{\rho}, \overline{\sigma}, \overline{\tau} & ::= & \alpha \mid \sigma \multimap \tau \mid \sigma \otimes \tau \mid \sigma \oplus \tau \mid 1 & \text{pretypes} \\
\rho, \sigma, \tau & ::= & \alpha \mid {}^{\xi}\overline{\tau} & \text{types}
\end{array}
$$

For example, consider type ${}^{\mathsf{R}}(\sigma \otimes \tau)$. Its qualifier, R, indicates that contraction but not weakening applies to variables of this type; its pretype, $\sigma \otimes \tau$, indicates that terms of this type are introduced and eliminated as pairs.

Unlike use types and standard types, λ^{URAL} has neither subtyping nor qualifier constraints on type schemes, which means that term $\lambda x.\, \lambda y.\, x\, y$ has all types of the form

$$
{}^{\xi}({}^{\xi_1}(\sigma \multimap \tau) \multimap {}^{\xi_2}(\sigma \multimap \tau))
$$

where $\xi_1 \sqsubseteq \xi_2$. Such a constraint is not expressible as a λ^{URAL} type.

In chapter 8, I use λ^{URAL} to explore the interaction of substructural types and control, so a full presentation of the system appears there.

Uniqueness types. Steadfast types and λ^{URAL} eliminate dereliction and promotion altogether. Another possibility, if the goal is to track uniqueness, is to reverse the direction of dereliction. That is, rather than $!^1\sigma <: !^0\sigma$, allow that $!^0\sigma <: !^1\sigma$, where $!^0\sigma$ indicates a unique, unaliased value, and $!^1\sigma$ indicates a value that is potentially aliased. Subsumption then amounts to forgetting the uniqueness of a value. Uniqueness types guarantee that a value has not been aliased, linear types guarantee that a value will not be aliased, and steadfast types guarantee both.

This is the direction taken by the Clean programming language (Brus et al. 1987). Uniqueness types are similar to λ^{URAL} types, but instead of use qualifiers, types are composed of pretypes and uniqueness attributes:

$$\begin{array}{rll}
\alpha, \beta, \gamma & \in & \mathit{TVar} & \text{type variables} \\
i, j, k & ::= & \alpha \mid \bullet \mid \times & \text{uniqueness attributes} \\
\overline{\rho}, \overline{\sigma}, \overline{\tau} & ::= & \alpha \mid \sigma \to \tau \mid 1 & \text{pretypes} \\
\rho, \sigma, \tau & ::= & \alpha \mid {}^{i}\overline{\tau} & \text{types}
\end{array}$$

Uniqueness attribute $\bullet$ indicates a unique value and $\times$ a potentially shared value. In the partial order of uniqueness attributes, unique is bottom and shared is top:

$$\bullet \leq i \qquad\qquad i \leq \times$$

In Clean, weakening applies to all types, but contraction applies only to non-unique types. However, (first order) unique values may be duplicated by forgetting their uniqueness, which is permitted by Clean's subtyping relation:

$$\frac{i \leq j \qquad \overline{\sigma} \text{ is not a function pretype}}{{}^{i}\overline{\sigma} <: {}^{j}\overline{\sigma}}.$$

In addition to subtyping, Clean types involve uniqueness constraints analogous to the use constraints found in Wadler's standard types.

One complication of Clean is that uniqueness subtyping does not apply to function types. The uniqueness attribute of a function is determined in a similar manner to the promotion rules of previous systems in this chapter: A function type gets attribute • if any of the values in its closure are unique. To guarantee that the uniqueness of those attributes is accurate, the system must ensure that a unique function is applied at most once. In Clean parlance, such a value is *necessarily unique,* which means essentially the same thing as *affine.*

A simplification. De Vries et al. (2007) propose a new treatment of uniqueness types that eliminates the need for constraints and subtyping. To get rid of the need for uniqueness constraints, they first extend the syntax of uniqueness attributes to a Boolean algebra:

$$i, j, k ::= \alpha \mid \bullet \mid \times \mid \neg i \mid i \wedge j \mid i \vee j \qquad \text{equality-based uniqueness attrs.}$$

Then any inequality constraint on uniqueness attributes may be solved, by Boolean unification, to yield a substitution that gives the same type scheme without the constraint. For example, given a constrained type scheme of the form

$$\ldots {}^{i}\overline{\sigma} \ldots {}^{j}\overline{\tau} \ldots \ [i \leq j],$$

the most-general Boolean unifier is $\{i \mapsto i, j \mapsto i \vee k\}$. Substituting, we get the equivalent, constraint-free type scheme

$$\ldots {}^{i}\overline{\sigma} \ldots {}^{i \vee k}\overline{\tau} \ldots.$$

Instead of subtyping, de Vries et al. (2007) use slack variables according to the polarity of types, as follows. Where • (unique, the bottom uniqueness attribute) appears in a covariant position, it is replaced with a type variable; similarly, a type variable replaces × (shared, the top uniqueness attribute) in contravariant positions. If the types of library functions have slack variables that correctly reflect their polymorphism over uniqueness, then useful polymorphic types are inferred for user functions as well.

A similar technique could be used to replace use variable constraints in Wadler's systems of use types and standard types.

2.2 Typestate

As a first special-purpose, stateful type system, we consider typestate. The original concept of typestate predates the substructural logics and type systems of the previous section. Thus, while the original theory does not rely on substructural types, we will see that typestate is a straightforward application of substructural types.

Strom and Yemini (1986) introduce typestate as a way to track the states of resources, to determine which operations on those resources are valid at a given program point. In this section, I discuss a more recent approach to typestate embodied by the safe, low-level programming language Vault (DeLine and Fähndrich 2001).

In Vault, values may be associated with compile-time *keys,* which track value ownership and state. The type checker maintains a *held key set* at each program point, ensuring that keys are neither duplicated nor dropped. (That is, keys are linear.) The key for a particular tracked value must be in the held key set at any program point where the associated value is accessed. DeLine and Fähndrich give the simple example of allocating a tracked object p with a new key named P:

tracked(P) point p = new tracked point { $x = 3$; $y = 4$; };

After such a declaration, P is in the held key set, which permits access to p.

A function that operates on a tracked value is annotated with a specification for how it treats the associated key. For example, consider a function that computes some property of a point:

double *norm*(tracked(K) point p) [K];

Function *norm* takes any point p tracked by some key K. The annotation [K] indicates that key K must be held when *norm* is called and continues to be held after *norm* returns. Annotations can also indicate that a function adds to or removes from the held key set. Attempting to access a tracked value when its key is not in the held key set is a type error. For example, *norm*(p) is a type error here because delete has already removed P from the held key set:

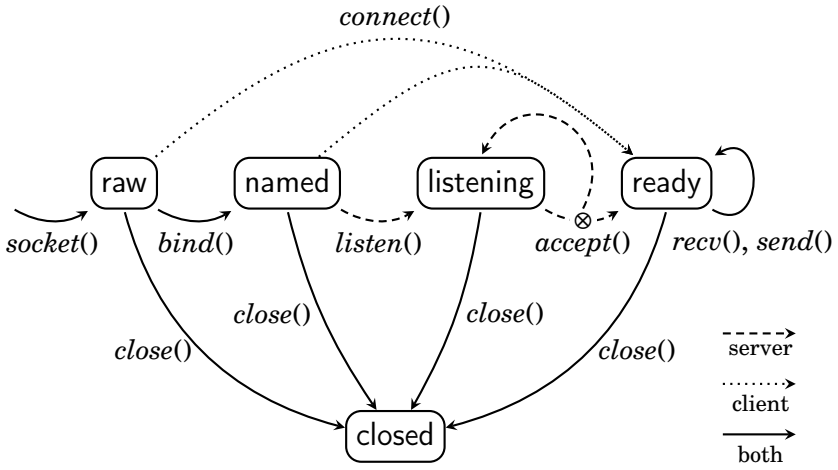


Figure 2.2: States and transitions for TCP (simplified)

delete p;
$f = norm(p)$;

2.2.1 Key States

Keys alone track ownership, but Vault adds to this the ability to associate with each key a state token, which reflects the state of the tracked value. As an example, DeLine and Fähndrich show a Vault interface for Berkeley sockets, the standard C language interface to network communication (Stevens 1990). Transmission Control Protocol (TCP), which provides reliable byte streams, is the standard transport layer protocol used by most internet applications (*e.g.,* SMTP, HTTP, and SSH). Setting up a TCP session using Berkeley sockets is a multi-step process (figure 2.2).

A network client must first create a communication end-point, called a *socket,* via the *socket*() system call. It may optionally select a port to use with *bind*(), and then it establishes a connection with *connect*(). Once a connection is established, the client may *send*() and *recv*() until either the client or the other side closes the connection.

For a server, the process is more involved: It begins with *socket*() and *bind*() as the client does, and then it calls *listen*() to allow connection requests to

```
interface SOCKET {
  type sock;

  variant domain [ `UNIX | `INET ];
  variant comm_style [ `STREAM | `DGRAM ];
  tracked(@raw) sock socket(domain, comm_style, int);

  struct sockaddr { ... };
  void bind(tracked(S) sock, sockaddr)   [S@raw→named];
  void listen(tracked(S) sock, int)   [S@named→listening];
  tracked(N) sock accept(tracked(S) sock)   [S@listening, new N@ready];

  void recv(tracked(S) sock, byte[])   [S@ready];
  void close(tracked(S) sock)   [−S];
}
```

Figure 2.3: Vault interface to TCP (server only)

begin queuing. The server calls *accept*() to accept a connection request. When *accept*() succeeds, it returns a new socket that is connected to a client, and the old, listening socket is available for further *accept*() calls. (For simplicity, I omit error transitions for now, but I discuss errors in §2.2.2.)

DeLine and Fähndrich's Vault interface for setting up a server-side TCP socket (2001, p. 4) appears in figure 2.3. The interface uses the same state names as the diagram in figure 2.2, and the same transition names, except that it omits *send* and *connect*. Function *socket* returns a new socket tracked by a key in the raw state, indicated by the return type **tracked**(@raw) sock. (The prototype for *socket* avoids naming the key, since the type of *socket* only needs to mention the key or its state once.) Function *bind* takes a socket tracked by some key S which must be in the raw state, and transitions the key to the named state; similarly, *listen* transitions from state named to state listening. Function *accept* takes a socket tracked by some key S in state listening, and leaves socket S in that same state, but it also returns a *new* socket tracked by a new key N in state ready. Function *recv* requires a socket in state ready and leaves it in that state. Finally, *close* takes a socket whose key S is in *any* state and removes S from the held key set, which prohibits further operations on

the corresponding socket.

2.2.2 Dynamic and Disjunctive Typestate

The Vault interface presented in the previous section relies on simplifying the state machine to obtain a property that does not necessarily hold in real-world APIs: a given transition can start in only one state or in any state (as *close* does), rather than a select subset of states; a transition always ends in one defined final state. This simplified notion of state machine cannot deal with two features of the actual Berkeley sockets TCP interface:

- When setting up a client, *connect* requires as a precondition *either* state raw or named; that is, the precondition is disjunctive.
- Each operation may fail, leaving the socket in the same state that it was in before attempted state transition; that is, the postcondition is disjunctive.

Vault provides a way to represent disjunctive typestates using algebraic data types, which allow dynamic management of keys and key states. For example, to add the *connect* operation, we can declare an algebraic data type with two constructors, one for each potential state in the precondition of *connect*:

```
variant connectable<key S> [ 'Raw {S@raw} | 'Named {S@named} ];
```

Constructing an instance of the variant requires a held key in the given state, and removes that key from the held key set; variants are destroyed by pattern matching, which reintroduces the key and its state into the held key set. Then *connect* can have the following prototype:

```
void connect(tracked(S) sock, string, int, tracked(C) connectable<S>)
  [−C, +S@ready];
```

That is, *connect* takes a socket tracked by key S, but does not require any precondition on S in its effect annotation. Instead, it requires a value of type

tracked(*C*) connectable<*S*>, which is a dynamic witness that *S* is in either the raw or named state. Furthermore, the connectable variant is itself tracked by key *C* to ensure that the dynamic witness is not duplicated. The typestate effect of *connect* is to remove *C* from the held key set, since the variant no longer accurately reflects the state of the socket, and to add key *S* at state ready back to the held key set.

Similarly, algebraic data types can be used to encode failure by having operations that can fail return a variant, which must then be pattern matched to check for failure. As an example, DeLine and Fähndrich give a prototype for a version of *bind* that can fail:

```
variant status<key K> [ `Ok {K@named} | `Error(error_code) {K@raw} ];
tracked status<S> bind(tracked(S) sock, sockaddr) [−S@raw];
```

Now *bind* takes a socket in state raw and removes its key from the held key set, and returns a status value. If the operation succeeds, then pattern matching the status value returns the key to the held key set at state named. If the operation fails, then pattern matching the status value yields an error code and restores the key at state raw.

2.2.3 Typestate in Linear Types

Typestate may be encoded easily using linear types, provided some mechanism for associating keys—which become ordinary values—with the values that they track.

Using dependent types. One way to tie values to their states is with dependent types, where a tracked state then appears as a predicate on values. For example, figure 2.4 contains the sockets API from figure 2.2, recast in a hypothetical linear, dependent type theory. In the example, sock and state are types, where the former represents a socket at run time and the latter has values standing for each possible socket state. Witnesses that a socket is in a particular state are made with binary type constructor $\cdot @ \cdot$, where a value of type $s@st$ is evidence that socket s is in state st.

$$
\begin{array}{ll}
\mathsf{sock} & : \mathrm{TYPE} \\
\mathsf{state} & : \mathrm{TYPE}\ \textbf{where}\ \mathsf{raw}, \mathsf{named}, \mathsf{listening}, \mathsf{ready} : \mathsf{state} \\
\cdot\,@\,\cdot & : \mathsf{sock} \xrightarrow{!} \mathsf{state} \xrightarrow{!} \mathrm{TYPE} \\
\\
\mathit{socket} & : \mathsf{domain} \xrightarrow{!} \mathsf{comm\_style} \xrightarrow{!} \mathsf{int} \xrightarrow{!} \Sigma(s : !\mathsf{sock}).\, s@\mathsf{raw} \\
\mathit{bind} & : \Pi(s : !\mathsf{sock}).\, \mathsf{sockaddr} \xrightarrow{!} s@\mathsf{raw} \xrightarrow{!}\!\circ\, s@\mathsf{named} \\
\mathit{listen} & : \Pi(s : !\mathsf{sock}).\, \mathsf{int} \xrightarrow{!} s@\mathsf{named} \xrightarrow{!}\!\circ\, s@\mathsf{listening} \\
\mathit{accept} & : \Pi(s : !\mathsf{sock}).\, s@\mathsf{listening} \xrightarrow{!}\!\circ\, (\Sigma(t : !\mathsf{sock}).\, t@\mathsf{ready}) \otimes s@\mathsf{listening} \\
\\
\mathit{recv} & : \Pi(s : !\mathsf{sock}).\, \mathsf{int} \xrightarrow{!} s@\mathsf{ready} \xrightarrow{!}\!\circ\, \mathsf{string} \otimes s@\mathsf{ready} \\
\mathit{close} & : \Pi(s : !\mathsf{sock})\,(st : !\mathsf{state}).\, s@st \xrightarrow{!}\!\circ\, 1
\end{array}
$$

(Sugar: $\sigma \xrightarrow{!}\!\circ\, \tau \triangleq\, !(\sigma \multimap \tau)$ and $\sigma \xrightarrow{!} \tau \triangleq\, !\sigma \xrightarrow{!}\!\circ\, \tau$)

Figure 2.4: Sockets API in a dependent ILL

Unlike in Vault, where keys and states are threaded implicitly by the type checker, these evidence tokens are threaded explicitly by a program, so each operation takes a witness to its precondition as an argument and returns a witness to its postcondition. For example, consider the (desugared) type of *listen*:

$$\Pi(s : !\mathsf{sock}).\, !(!\mathsf{int} \multimap !(s@\mathsf{named} \multimap s@\mathsf{listening})).$$

This means that *listen* takes an unlimited socket, named s, an unlimited integer (the port for listening), and linear evidence that socket s is in state named. It returns evidence that socket s is in state listening.

A socket's state changes non-monotonically. An operation such as *listen*, which applies at some point in time, will not apply at a later point in time when the socket is no longer in the same state. For the state to evolve safely, the API must be able to *revoke* each old state witness when the client performs an operation that changes the state. Essential to making this encoding of typestate work is that there is always exactly one state witness for each socket value. Linear types alone do not guarantee this property, but linear types do guarantee that the client of the API cannot alias the witness tracking a socket's state. The other necessary condition is that each of the operations maintains the invariant of one witness per socket, which they clearly do.

Using phantom types. Dependent types are not the only way to associate run-time values with their states. Another way is to add a phantom parameter to both the type of the run-time value and the state witness type, and to use parametric polymorphism to generate a fresh abstract type to tie the two together. This approach, rendered in λ^{URAL}, appears in figure 2.5.

In this version of the sockets API, the type of sockets (sock) takes a type parameter, which will be used to identify it. The witness pretype constructor $(\cdot \mathbin{@} \cdot)$ takes a matching type to tie it to the socket and a pretype indicating the state. For example, given a socket of unlimited type $^{\mathsf{U}}(\mathsf{sock}\ \alpha)$, a linear witness value of type $^{\mathsf{L}}(\alpha @ \mathsf{ready})$ is evidence that the socket is in the ready state. Function *socket* returns an existential package of type

$$^{\mathsf{L}}\exists\alpha{:}\mathrm{TYPE}.\ ^{\mathsf{L}}(^{\mathsf{U}}(\mathsf{sock}\ \alpha) \otimes {}^{\mathsf{L}}(\alpha @ \mathsf{raw})).$$

The existential quantifier ensures that type α matches no other type in the program. This effectively ties the socket to its state, because the type tag of the witness for one socket will never match the tag of a different socket.

Other than the mechanism for tying a socket to its state, the λ^{URAL} version of the API in figure 2.5 follows the dependently-typed version of figure 2.4 very closely.

2.3 Region-Based Memory Management

As a second example of a special-purpose, stateful type system, we consider region-based memory management.

2.3.1 Nested Regions

Memory regions are a technique to allow safe, flexible memory management without requiring a tracing garbage collector. Tofte and Talpin (1997) introduce stack-like regions as a generalization of the stack allocation regime common to block-structured programming languages. Under stack allocation, the lifetimes of objects are known because they correspond to activation records or other lexical structures in a program. This can be efficient, but it is limiting because

$$
\begin{array}{ll}
\mathsf{sock} & : \textsc{type} \Rightarrow \textsc{pretype} \\
\mathsf{raw}, \mathsf{named}, \mathsf{listening}, \mathsf{ready} & : \textsc{pretype} \\
\cdot\,@\,\cdot & : \textsc{type} \Rightarrow \textsc{pretype} \Rightarrow \textsc{pretype}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{socket} & : \mathsf{domain} \xrightarrow{\mathsf{U}} \mathsf{comm\_style} \xrightarrow{\mathsf{U}} \mathsf{int} \xrightarrow{\mathsf{U}} \\
& \quad {}^{\mathsf{L}}\exists\alpha{:}\textsc{type}.\, {}^{\mathsf{L}}({}^{\mathsf{U}}(\mathsf{sock}\ \alpha) \otimes {}^{\mathsf{L}}(\alpha @ \mathsf{raw})) \\
\mathit{bind} & : {}^{\mathsf{U}}\forall\alpha{:}\textsc{type}.\, \mathsf{sock}\ \alpha \xrightarrow{\mathsf{U}} \mathsf{sockaddr} \xrightarrow{\mathsf{U}} {}^{\mathsf{L}}(\alpha @ \mathsf{raw}) \xrightarrow{\mathsf{U}}_{\circ} {}^{\mathsf{L}}(\alpha @ \mathsf{named}) \\
\mathit{listen} & : {}^{\mathsf{U}}\forall\alpha{:}\textsc{type}.\, \mathsf{sock}\ \alpha \xrightarrow{\mathsf{U}} \mathsf{int} \xrightarrow{\mathsf{U}} {}^{\mathsf{L}}(\alpha @ \mathsf{named}) \xrightarrow{\mathsf{U}}_{\circ} {}^{\mathsf{L}}(\alpha @ \mathsf{listening}) \\
\mathit{accept} & : {}^{\mathsf{U}}\forall\alpha{:}\textsc{type}.\, \mathsf{sock}\ \alpha \xrightarrow{\mathsf{U}} {}^{\mathsf{L}}(\alpha @ \mathsf{listening}) \xrightarrow{\mathsf{U}}_{\circ} \\
& \quad {}^{\mathsf{L}}({}^{\mathsf{L}}(\exists\beta{:}\textsc{type}.\, {}^{\mathsf{L}}({}^{\mathsf{U}}(\mathsf{sock}\ \beta) \otimes {}^{\mathsf{L}}(\beta @ \mathsf{ready}))) \otimes {}^{\mathsf{L}}(\alpha @ \mathsf{listening})) \\
\mathit{recv} & : {}^{\mathsf{U}}\forall\alpha{:}\textsc{type}.\, \mathsf{sock}\ \alpha \xrightarrow{\mathsf{U}} {}^{\mathsf{L}}(\alpha @ \mathsf{ready}) \xrightarrow{\mathsf{U}}_{\circ} {}^{\mathsf{L}}({}^{\mathsf{L}}(\alpha @ \mathsf{ready}) \otimes {}^{\mathsf{U}}\mathsf{string}) \\
\mathit{close} & : {}^{\mathsf{U}}\forall\alpha{:}\textsc{type}.\, {}^{\mathsf{U}}\forall\beta{:}\textsc{pretype}.\, \mathsf{sock}\ \alpha \xrightarrow{\mathsf{U}} {}^{\mathsf{L}}(\alpha @ \beta) \xrightarrow{\mathsf{U}}_{\circ} {}^{\mathsf{U}}1
\end{array}
$$

(Sugar: $\sigma \xrightarrow{\xi}_{\circ} \tau \triangleq {}^{\xi}(\sigma \multimap \tau)$ and $\overline{\sigma} \xrightarrow{\xi} \tau \triangleq {}^{\mathsf{U}}\overline{\sigma} \xrightarrow{\xi}_{\circ} \tau$)

Figure 2.5: Sockets API in λ^{URAL}

an object allocated in a particular stack frame becomes inaccessible once that frame is removed from the stack.

Tofte and Talpin propose regions as a way to allow more flexible object lifetimes, without giving up memory safety or adding tracing garbage collection. In their formulation, a region is a subspace of the heap in which objects may be allocated, like a stack frame, and region lifetimes are nested like stack frames. However, objects are not always allocated in the top-most enclosing region, but rather in the top-most (and thus shortest-lived) region whose lifetime contains the necessary lifetime of the object. Tofte and Talpin give a type-and-effect system for nested regions and an algorithm for annotating any typeable ML program with two kinds of region operations:

- **letregion** ρ **in** e **end** allocates a new, lexically-scoped region, binding the region variable ρ to it, and evaluates expression e in its context.
- e **at** ρ allocates the value of expression e in the region named by ρ.

They prove that the annotation algorithm, under a suitable dynamic semantics for regions, preserves the meaning of the original ML program.

Tofte and Talpin's (1997) region system is implemented in the ML Kit (Birkedal et al. 1993) compiler for Standard ML (Milner et al. 1997). Tofte and Talpin found that programs ported from Standard ML of New Jersey (Appel and MacQueen 1991) to the ML Kit often ran slower and used more memory than before. With some profiler-guided refactoring, however, their benchmarks could be made to use less memory than under SML/NJ.

2.3.2 Calculus of Capabilities

Walker et al. (2000) introduce capability-based regions as a more flexible alternative to the nested regions of Tofte and Talpin. They observe that nested regions are efficient provided that object lifetimes mostly follow the lexical structure of a program, but that for other programs, nested regions may result in holding onto memory long after it is actually needed. Seemingly benign changes in program structure can have radical effects on a program's memory usage. Some program transformations have especially bad consequences; for example, transforming a program to continuation-passing style results in delaying *all* deallocation until the end of the program. To fix the fragility of nested region inference, Walker et al. propose a calculus of capabilities, CL, which no longer relies on lexical nesting to manage region lifetimes.

Unlike Tofte and Talpin, Walker et al. wish to support safe, *explicit* memory management. (Explicit memory management is possible in Tofte and Talpin's internal, annotated language, but not in their ML-like source language.) For low-level applications where tracing garbage collection is inappropriate or unavailable, such as when implementing a garbage collector, manual control over memory allocation and deallocation is necessary. Capability-based regions provide a way to gain that control without giving up memory safety.

In CL, the lexical **letregion** form is split into two separate operations, **newrgn** for region allocation and **freergn** for region deallocation. Furthermore, the representation of regions is split into run-time region handles, which are ordinary values, and their corresponding compile-time region capabilities, which are threaded through a program similarly to how Vault's type system

threads keys.[4] Allocating in or freeing a region requires both the run-time handle value and the compile-time capability to that region. Accessing values allocated in a region does not require the run-time handle, but does require that the region's capability be held, which ensures that no value is accessed after its region and associated capability have been released.

Walker et al. want to support manual deletion of regions (which deallocates all of a region's contents), but they also want maximal flexibility when using regions. As an example of the desired flexibility, they discuss the problem of a function that takes two heap-allocated values that may be (but are not necessarily) in different regions. The function needs to take a capability to access the regions, and to return the capability so that the rest of the program can continue to access the region. They point out that linear regions will not work for this example, because if the two values happen to be in the same region then the region capability must become aliased to pass it for each value.

Walker et al. solve the problem in CL as follows. First, they observe that allocating in a region or accessing pointers to a region requires only that the region be live, not that the reference to the region be unaliased at that point. However, deleting a region requires a unique capability, to ensure that after the region is deleted no other capabilities to it exist. To express this distinction, Walker et al. distinguish unique region capabilities $^{\bullet}\mathsf{cap}\,\rho$ from shared region capabilities $^{\times}\mathsf{cap}\,\rho$.[5] CL has a capability subtyping relation that, among other things, allows a unique capability to become shared (as in uniqueness types) and allows duplication of shared capabilities:

$$^{\bullet}\mathsf{cap}\,\rho \leq {}^{\times}\mathsf{cap}\,\rho \qquad\qquad {}^{\times}\mathsf{cap}\,\rho \leq {}^{\times}\mathsf{cap}\,\rho \otimes {}^{\times}\mathsf{cap}\,\rho$$

Region deletion then requires a unique capability, but all other region operations work with either a unique or shared capability.

Capability subtyping alone is insufficient to solve the example problem. On the one hand, if the function takes (and returns to its continuation) shared

[4] This is no coincidence; Vault's type system is based on Walker et al.'s capability calculus.

[5] This is not precisely CL's syntax for capabilities, but instead follows the treatment of uniqueness types in §2.1.2.

capabilities to the regions,

$$\forall\alpha_1{:}\mathrm{RGN},\alpha_2{:}\mathrm{RGN}.({}^{\times}\mathsf{cap}\,\alpha_1 \otimes {}^{\times}\mathsf{cap}\,\alpha_2,\ldots,({}^{\times}\mathsf{cap}\,\alpha_1 \otimes {}^{\times}\mathsf{cap}\,\alpha_2,\ldots)\to 0)\to 0,$$

then the regions can never be freed, since the unique capabilities to the regions cannot be recovered. On the other hand, if the function takes (and returns) unique capabilities to the regions,

$$\forall\alpha_1{:}\mathrm{RGN},\alpha_2{:}\mathrm{RGN}.({}^{\bullet}\mathsf{cap}\,\alpha_1 \otimes {}^{\bullet}\mathsf{cap}\,\alpha_2,\ldots,({}^{\bullet}\mathsf{cap}\,\alpha_1 \otimes {}^{\bullet}\mathsf{cap}\,\alpha_2,\ldots)\to 0)\to 0,$$

then the two regions cannot be the same, since unique capabilities cannot be aliased. Instead, Walker et al. use CL's bounded quantification to get this type:

$$\forall\alpha_1{:}\mathrm{RGN},\alpha_2{:}\mathrm{RGN}.\,\forall\beta\le {}^{\times}\mathsf{cap}\,\alpha_1 \otimes {}^{\times}\mathsf{cap}\,\alpha_2.(\beta,\ldots,(\beta,\ldots)\to 0)\to 0.$$

That is, the function takes (and returns) a capability that subsumes shared capabilities to access the two regions. If the two regions are distinct, then the capability to access both of them clearly subsumes the capability granting shared access to both of them, and thus the function may be called; if there is only one region, then subsumption allows sharing and duplicating the region, so the function may be called with the single region capability. In either case, the continuation gets the original capability back.

2.3.3 Regions in Vault

Walker et al.'s (2000) region capabilities provide very flexible, safe, manual control over memory management, but the essentials of substructural regions are realizable in other systems as well. For example, DeLine and Fähndrich give an interface for regions in Vault, which appears in figure 2.6. In the Vault interface, function *create* allocates a new region, tracked by a new key R, and function *delete* destroys a region and its key.

An example using Vault regions, also from DeLine and Fähndrich (2001), appears in figure 2.7. The first line allocates a new region, *rgn*, tracked by new key R. To allocate values in a region and use those values, Vault allows the **new** operator to be parametrized by the region in which to allocate the value. The second line allocates a new point object in region *rgn*. The new value has

```
module Region : {
  type region;
  tracked(R) region create()   [new R];
  void delete(tracked(R) region)   [−R];
}
```

Figure 2.6: Vault region API

```
tracked(R) region rgn = Region.create();
R:point pt = new(rgn) point { x = 1; y = 2; };
pt.x++;
Region.delete(rgn);
```

Figure 2.7: Vault region client example

$$
\begin{array}{ll}
\mathit{newrgn} & : {}^{\mathsf{U}}({}^{\mathsf{L}}1 \multimap {}^{\mathsf{L}}\exists\alpha.\,{}^{\mathsf{L}}({}^{\mathsf{L}}\mathsf{cap}\,\alpha \otimes {}^{\mathsf{U}}\mathsf{hnd}\,\alpha)) \\
\mathit{freergn} & : {}^{\mathsf{U}}\forall\alpha.\,{}^{\mathsf{U}}({}^{\mathsf{L}}({}^{\mathsf{L}}\mathsf{cap}\,\alpha \otimes {}^{\mathsf{U}}\mathsf{hnd}\,\alpha) \multimap {}^{\mathsf{L}}1) \\
\mathit{new} & : {}^{\mathsf{U}}\forall\alpha.\,{}^{\mathsf{U}}\forall\gamma.\,{}^{\mathsf{U}}({}^{\mathsf{L}}({}^{\mathsf{L}}\mathsf{cap}\,\alpha \otimes {}^{\mathsf{U}}\mathsf{hnd}\,\alpha \otimes {}^{\mathsf{U}}\gamma) \multimap {}^{\mathsf{L}}({}^{\mathsf{L}}\mathsf{cap}\,\alpha \otimes {}^{\mathsf{U}}(\mathsf{ref}\,\alpha\,{}^{\mathsf{U}}\gamma))) \\
\mathit{read} & : {}^{\mathsf{U}}\forall\alpha.\,{}^{\mathsf{U}}\forall\gamma.\,{}^{\mathsf{U}}({}^{\mathsf{L}}({}^{\mathsf{L}}\mathsf{cap}\,\alpha \otimes {}^{\mathsf{U}}(\mathsf{ref}\,\alpha\,{}^{\mathsf{U}}\gamma)) \multimap {}^{\mathsf{L}}({}^{\mathsf{L}}\mathsf{cap}\,\alpha \otimes {}^{\mathsf{U}}\gamma)) \\
\mathit{write} & : {}^{\mathsf{U}}\forall\alpha.\,{}^{\mathsf{U}}\forall\gamma.\,{}^{\mathsf{U}}({}^{\mathsf{L}}({}^{\mathsf{L}}\mathsf{cap}\,\alpha \otimes {}^{\mathsf{U}}(\mathsf{ref}\,\alpha\,{}^{\mathsf{U}}\gamma) \otimes {}^{\mathsf{U}}\gamma) \multimap {}^{\mathsf{L}}({}^{\mathsf{L}}\mathsf{cap}\,\alpha \otimes {}^{\mathsf{U}}1))
\end{array}
$$

Figure 2.8: λ^{rgnUL} region API

the *guarded* type R:point, which means that key R must be in the held key set to access the point. The third line mutates a field in the new point, and the fourth line deletes the region, thereby removing key R from the held key set. Any attempt to access point pt after deleting the region would be a type error.

2.3.4 Linear Regions

Unsurprisingly, a linear type system is sufficient to support regions. Fluet et al. (2006) describe λ^{rgnUL}, a derivative of λ^{URAL} with linear regions. The λ^{rgnUL} interface to regions appears in figure 2.8.

The style is similar to the λ^{URAL} rendering of typestate in figure 2.5, in that both static capabilities and dynamic region handles are represented as ordinary values that share an existentially quantified type parameter. Type $^{\mathsf{L}}\mathsf{cap}\,\alpha$ is a static capability to access a region identified by type tag α, and type $^{\mathsf{U}}\mathsf{hnd}\,\alpha$ is a run-time region handle to the same region. Type $^{\mathsf{U}}(\mathsf{ref}\,\alpha\,{}^{\mathsf{U}}\gamma)$ is a reference to an unlimited value of type $^{\mathsf{U}}\gamma$ allocated in region α.

Functions *newrgn* and *freergn* correspond to Vault functions *Region.create* and *Region.delete* of figure 2.6. Function *newrgn* returns a pair of a region capability (which corresponds to the key in Vault) and a region handle (which corresponds to the actual region value); these are tied together with a shared, existentially quantified type variable. Function *freergn* takes a linear region capability and matching region handle, and consumes the linear capability, thereby revoking access to the region.

Functions *new*, *read*, and *write* allocate and access references in a region. (Because this interface supports only unlimited references, there is no operation to free an individual reference, but of course references are deallocated when their region is freed.) All three operations take a linear region capability, which witnesses that the region is still live, and returns the same capability to allow further operations on the region. Allocating a new reference requires both the capability and the unlimited region handle, which contains the actual run-time information necessary to allocate a reference in a region. As in previous versions of regions in this section, reading and writing a reference requires the region capability, again to witness liveness, and the reference value, but does not require the region handle.

Linear and affine references, which support strong updates, as well as references to non-unlimited data, are available in rgnURAL, another extension to λ^{URAL} (Fluet 2007).

2.4 Session Types

As a final instance of a stateful type system, I consider session types, a technique for checking communication protocols in the context of message-passing concurrency. As a simple example, Honda et al. (1998) consider a

protocol that an automatic teller machine (ATM) might use to communicate with a bank. This protocol, from the perspective of the ATM, begins with a three-way branch from which the ATM may choose to make a balance inquiry, a deposit, or a withdrawal. Following that choice, there is a different protocol for each kind of transaction:

deposit The ATM sends the bank the amount to deposit as an integer, the bank responds with a transaction number (an integer), and the session ends.

balance The bank sends the ATM the account balance as an integer, and the session ends.

withdraw The ATM sends the bank the amount to withdraw. The bank then can select from a two-way branch:

success The withdrawal has succeeded. The bank sends a transaction number and the session ends.

failure The withdrawal has failed. The bank sends the reason for failure as a string and the session ends.

A state diagram summarizing the ATM protocol appears in figure 2.9. Notation in the diagram is as follows. Each branch node is labeled by one of two symbols: $\oplus$ indicates an internal choice, which means that the ATM selects which branch to take and the bank must follow; or $\&$ indicates an external choice, which means that the ATM must be prepared to follow whichever branch the bank selects. (The similarity of these symbols to connectives of linear logic is no coincidence.) Edges following a branch have labels (*e.g., deposit*), which name the branches. Other edges are labeled either $\uparrow\tau$ for some type τ, meaning that the ATM sends a value of type τ, or $\downarrow\tau$, meaning that the ATM receives a value of type τ.

Such a protocol can be implemented in a typed, message passing language such as Concurrent ML (Reppy 1999, henceforth CML), but the types in such a language will do little or nothing to enforce the protocol. In CML, threads communicate over bidirectional channels of type τ chan, where τ is the type of

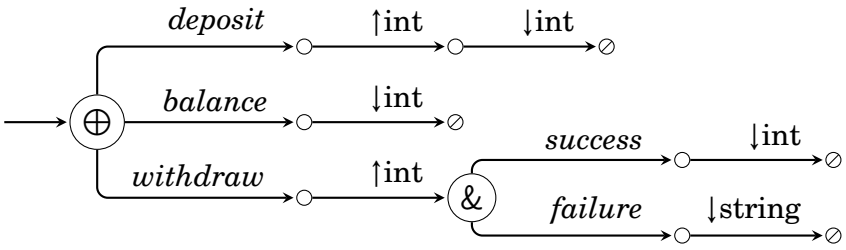


Figure 2.9: State diagram for ATM–bank protocol

messages that can be sent over the channel. For the ATM protocol, messages need to include the client's choice of transaction type, integers, the server's withdrawal response, and strings. One way to encode the messages for the protocol appears in figure 2.10, and a client that uses the protocol to retrieve an account balance appears in figure 2.11. The communication channel has type `atm_message chan`, which means that upon receiving a message from the server, the client needs to pattern match to extract the expected message type. This leaves room for two kinds of protocol errors: a thread can send (or expect to receive) the wrong kind of message at some point, or a thread can attempt to receive (or send) when the protocol requires it to send (or receive).

2.4.1 Session Types

In order to enforce statically that message-passing communication proceeds according to consistent protocols, Honda et al. (1998) introduce the notion of session types. A session type describes a protocol as a regular expression, using the following syntax:[6]

$$\begin{array}{rcll} \chi & \in & SVar & \text{session variables} \\ l & \in & Lab & \text{labels} \\ S,T & ::= & \uparrow\tau;S \mid \downarrow\tau;S \mid \uparrow T;S \mid \downarrow T;S \mid \mu\chi.S \mid \chi \mid \mathsf{end} & \text{session types} \\ & & \mid \underline{\oplus}\langle l_1 : S_1 \parallel \cdots \parallel l_k : S_k \rangle \mid \underline{\&}\langle l_1 : S_1 \parallel \cdots \parallel l_k : S_k \rangle & \end{array}$$

[6] I underline type constructors $\underline{\oplus}$ and $\underline{\&}$ to distinguish these session type constructors from the usual connectives of linear logic. Similarly, I use ↑ and ↓ for sending and receiving, rather than the usual ! and ?, to avoid confusion.

```
datatype atm_message = Deposit of int
                     | TransId of int
                     | GetBalance
                     | Balance of int
                     | Withdraw of int
                     | Failure of string
```

Figure 2.10: Message type for ATM protocol in CML

```
fun getBalance (chan : atm_message chan) : int =
  (send (chan, GetBalance);
   case recv chan of
     Balance x → x
   | _         → raise ProtocolError)
```

Figure 2.11: ATM client code for getting the balance in CML

We assume denumerable sets of session variables χ and labels l. Type $\uparrow\tau;S$ (resp. $\downarrow\tau;S$) represents a protocol in which the next step is to send (resp. receive) a value of type τ, followed by protocol S. Session delegation is given by types $\uparrow T;S$ and $\downarrow T;S$—rather than sending and receiving values, they allow a thread to send or receive a session over a channel. Recursive sessions, which represent protocols with cycles, are written using $\mu\chi.S$ and χ in the style of equirecursive types. Type end represents the completed session. Finally, type $\underline{\oplus}\langle\, l_1 : S_1 \parallel \cdots \parallel l_k : S_k \,\rangle$ is a k-way internal choice, in which the thread must select some label l_i ($1 \le i \le k$), followed by performing protocol S_i; dually, $\underline{\&}\langle\, l_1 : S_1 \parallel \cdots \parallel l_k : S_k \,\rangle$ is a k-way external choice, whereby a thread must be prepared to follow whichever label is chosen by the other thread in the session.

Using session types, the ATM protocol may be written as

$$\begin{aligned}\mathsf{atm\_prot} = \underline{\oplus}\langle\, & \mathit{deposit} : \uparrow\mathsf{int};\downarrow\mathsf{int};\mathsf{end} \parallel \\ & \mathit{balance} : \downarrow\mathsf{int};\mathsf{end} \parallel \\ & \mathit{withdraw} : \uparrow\mathsf{int};\underline{\&}\langle\, \mathit{success} : \downarrow\mathsf{int};\mathsf{end} \parallel \mathit{failure} : \downarrow\mathsf{string};\mathsf{end} \,\rangle \,\rangle.\end{aligned}$$

$$\overline{\uparrow\tau;S} = \downarrow\tau;\overline{S} \qquad \overline{\downarrow\tau;S} = \uparrow\tau;\overline{S} \qquad \overline{\uparrow T;S} = \downarrow T;\overline{S} \qquad \overline{\downarrow T;S} = \uparrow T;\overline{S}$$

$$\overline{\chi} = \chi \qquad \overline{\mu\chi.S} = \mu\chi.\overline{S} \qquad \overline{\mathsf{end}} = \mathsf{end}$$

$$\overline{\underline{\oplus}\langle l_1 : S_1 \parallel \cdots \parallel l_k : S_k \rangle} = \underline{\&}\langle l_1 : \overline{S_1} \parallel \cdots \parallel l_k : \overline{S_k} \rangle$$

$$\overline{\underline{\&}\langle l_1 : S_1 \parallel \cdots \parallel l_k : S_k \rangle} = \underline{\oplus}\langle l_1 : \overline{S_1} \parallel \cdots \parallel l_k : \overline{S_k} \rangle$$

Figure 2.12: Session type duality

Session type duality. For a session to run smoothly, the two communicating threads must speak not the same protocol, but dual protocols. Session type duality is defined in figure 2.12. Duality exchanges sending with receiving and external choice with internal choice. Note that duality is an involution.

In order to initiate sessions with two threads speaking dual protocols, session types languages often include *rendezvous values* of type $[S]$, where S is a session type, along with operations *request* and *accept*. These operations each take a rendezvous value of type $[S]$, but they return different types: *request* returns a channel for the client-side protocol S, and *accept* returns a channel for the server-side protocol $\overline{S}$. Either operation blocks until it can be paired with the complementary operation on the same rendezvous value, which guarantees that the new channel will be typed at S on one side and $\overline{S}$ on the other side. The details of how this works, however, depend on the particular language.

Functional session types. Session types were originally developed in the context of the π calculus, but here I follow Vasconcelos et al.'s (2004) version of session types in a λ-calculus setting. Assume a denumerable set (*CVar*) of channel names (c); then a session environment (Σ) associates channel names with sessions, tracking the protocol state of each channel. Types include $\mathsf{chan}\ c$, which is the run-time value for channel c; function types of the form $(\Sigma, \sigma) \to (\Sigma', \tau)$, where σ and τ are the domain and codomain of the function, and Σ and Σ' are a precondition and postcondition giving the state of the channels used by the function; and $[S]$, which is the type of rendezvous values as described above.

$$c, d \in \mathit{CVar} \qquad \text{channel names}$$

$$\Sigma ::= c_1 : S_1, \ldots, c_k : S_k \qquad \text{session environments}$$

$$\rho, \sigma, \tau ::= \cdots \mid \mathsf{chan}\, c \mid (\Sigma, \sigma) \to (\Sigma', \tau) \mid [S] \qquad \text{types}$$

One way to understand this system of session types is by analogy to typestate in Vault. Channel names c function like Vault keys, where a value of type $\mathsf{chan}\, c$ is tracked by key c. A channel environment Σ is like the held key set, threaded linearly through the program. Thus, channels are unlimited, and may be shared, but may only be used according to the session recorded in the current channel environment.

Vasconcelos et al. (2004) provide operations for creating rendezvous values, requesting and accepting sessions, sending, receiving, selecting on internal choices, dispatching on external choices, and closing sessions. For example, expression $\mathbf{send}\, v\, \mathbf{on}\, v'$ sends value v on channel v'. The expression is typed using one of two rules, depending on whether v is a non-channel or channel value:

$$\frac{\Gamma \vdash v : \tau \qquad \neg(\exists d)\, \tau = \mathsf{chan}\, d \qquad \Gamma \vdash v' : \mathsf{chan}\, c}{\Gamma \vdash \Sigma, c : \uparrow \tau; S \rhd \mathbf{send}\, v\, \mathbf{on}\, v' : 1 \lhd \Sigma, c : S}$$

$$\frac{\Gamma \vdash v : \mathsf{chan}\, d \qquad \Gamma \vdash v' : \mathsf{chan}\, c}{\Gamma \vdash \Sigma, c : \uparrow T; S, d : T \rhd \mathbf{send}\, v\, \mathbf{on}\, v' : 1 \lhd \Sigma, c : S}.$$

(The expression typing judgment has the form $\Gamma \vdash \Sigma \rhd e : \tau \lhd \Sigma'$, where Σ and Σ' are the precondition and postcondition for e, respectively.) The first rule is for sending a non-channel value of type τ. It requires that value v' have type $\mathsf{chan}\, c$ for some channel name c, where the session of c in the precondition says to send τ, followed by some session S; then the postcondition has channel c at session S. The second rule is for when the value to send is some channel d. It requires that the precondition has d at session T, and that the channel on which to send, c, has the protocol to send session T followed by some protocol S. Then the postcondition has channel c at session S, and no longer mentions channel d, since d is delegated to another thread.

$\lambda(c : \mathsf{atm\_prot};\ chan : \mathsf{chan}\ c).$
  select *balance* **on** *chan*;
  let *result* = **receive** *chan* **in**
  close *chan*;
  result

Figure 2.13: ATM client code in Vasconcelos et al.'s (2004) language

$$
\begin{array}{ll}
\uparrow \cdot ; \cdot & : \textsc{type} \Rightarrow \textsc{session} \Rightarrow \textsc{session} \\
\downarrow \cdot ; \cdot & : \textsc{type} \Rightarrow \textsc{session} \Rightarrow \textsc{session} \\
\cdot \mathbin{\underline{\&}} \cdot & : \textsc{session} \Rightarrow \textsc{session} \Rightarrow \textsc{session} \\
\cdot \mathbin{\underline{\oplus}} \cdot & : \textsc{session} \Rightarrow \textsc{session} \Rightarrow \textsc{session} \\
\mathsf{end} & : \textsc{session} \\
\\
[\cdot] & : \textsc{session} \Rightarrow \textsc{pretype} \\
\mathsf{chan}\cdot & : \textsc{channel} \Rightarrow \textsc{pretype} \\
\cdot @ \cdot & : \textsc{channel} \Rightarrow \textsc{session} \Rightarrow \textsc{pretype} \\
\\
\mathit{newRendezvous} & : {}^{\mathsf{U}}\forall\gamma.\,{}^{\mathsf{U}}1 \xrightarrow{\mathsf{U}}_{\circ} {}^{\mathsf{U}}[\gamma] \\
\mathit{request} & : {}^{\mathsf{U}}\forall\gamma.\,{}^{\mathsf{U}}[\gamma] \xrightarrow{\mathsf{U}}_{\circ} {}^{\mathsf{L}}\exists\alpha.\,{}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{chan}\ \alpha \otimes {}^{\mathsf{L}}(\alpha @ \gamma)) \\
\mathit{accept} & : {}^{\mathsf{U}}\forall\gamma.\,{}^{\mathsf{U}}[\gamma] \xrightarrow{\mathsf{U}}_{\circ} {}^{\mathsf{L}}\exists\alpha.\,{}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{chan}\ \alpha \otimes {}^{\mathsf{L}}(\alpha @ \overline{\gamma})) \\
\\
\mathit{send} & : {}^{\mathsf{U}}\forall\alpha\beta\gamma.\,{}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{chan}\ \alpha \otimes {}^{\mathsf{L}}(\alpha @ (\uparrow\beta;\gamma)) \otimes \beta) \xrightarrow{\mathsf{U}}_{\circ} {}^{\mathsf{L}}(\alpha @ \gamma) \\
\mathit{receive} & : {}^{\mathsf{U}}\forall\alpha\beta\gamma.\,{}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{chan}\ \alpha \otimes {}^{\mathsf{L}}(\alpha @ (\downarrow\beta;\gamma))) \xrightarrow{\mathsf{U}}_{\circ} {}^{\mathsf{L}}(\beta \otimes {}^{\mathsf{L}}(\alpha @ \gamma)) \\
\mathit{follow} & : {}^{\mathsf{U}}\forall\alpha\gamma_1\gamma_2.\,{}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{chan}\ \alpha \otimes {}^{\mathsf{L}}(\alpha @ (\gamma_1 \mathbin{\underline{\&}} \gamma_2))) \xrightarrow{\mathsf{U}}_{\circ} \\
 & \qquad\qquad {}^{\mathsf{L}}({}^{\mathsf{L}}(\alpha @ \gamma_1) \oplus {}^{\mathsf{L}}(\alpha @ \gamma 2)) \\
\mathit{selectL} & : {}^{\mathsf{U}}\forall\alpha\gamma_1\gamma_2.\,{}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{chan}\ \alpha \otimes {}^{\mathsf{L}}(\alpha @ (\gamma_1 \mathbin{\underline{\oplus}} \gamma_2))) \xrightarrow{\mathsf{U}}_{\circ} {}^{\mathsf{L}}(\alpha @ \gamma_1) \\
\mathit{selectR} & : {}^{\mathsf{U}}\forall\alpha\gamma_1\gamma_2.\,{}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{chan}\ \alpha \otimes {}^{\mathsf{L}}(\alpha @ (\gamma_1 \mathbin{\underline{\oplus}} \gamma_2))) \xrightarrow{\mathsf{U}}_{\circ} {}^{\mathsf{L}}(\alpha @ \gamma_2) \\
\mathit{close} & : {}^{\mathsf{U}}\forall\alpha.\,{}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{chan}\ \alpha \otimes {}^{\mathsf{L}}(\alpha @ \mathsf{end})) \xrightarrow{\mathsf{U}}_{\circ} {}^{\mathsf{U}}1
\end{array}
$$

Figure 2.14: Session types in λ^{URAL}

A client that uses the ATM protocol to retrieve an account balance, written in Vasconcelos et al.'s (2004) session types language, appears in figure 2.13.

2.4.2 Linear Session Types

As with the previous examples of stateful type systems described in this chapter, a session types interface is expressible using linear types, given the right set of types and constants (Pucella and Heller 2008).[7] Figure 2.14 contains an interface for session types in λ^{URAL}. I assume two new kinds, SESSION and CHANNEL; it works if both of those are replaced with TYPE, but distinguishing them makes things clearer. The interface contains types for sending, receiving, choice, and the finished session. Instead of k-way labeled choice, I use a two-way unlabeled choice for simplicity. Unlike in Vasconcelos et al. (2004), there is no need for distinguished session types for sending and receiving sessions, since session capabilities are now ordinary values.

Rendezvous values have unlimited type $^{\mathsf{U}}[\gamma]$, where γ is the session type available for requesting or accepting. Function *newRendezvous* creates a new rendezvous value for any protocol. A channel, of unlimited type $^{\mathsf{U}}\mathsf{chan}\ \alpha$, has type parameter α to associate it with a linear session capability of type $^{\mathsf{L}}(\alpha \mathbin{@} \gamma)$, where γ is the session state of that channel. (This is the same technique used to associate sockets with their states in figure 2.5 on p. 23.) Functions *request* and *accept* each take a rendezvous value and return an existentially quantified pair of a channel and a session capability; given a rendezvous value of type $[S]$, *request* returns a capability for session S, and *accept* returns a capability for dual session $\overline{S}$. The operations for running a session all evolve the session capability in the expected way. For example, the type of *send* is

$$^{\mathsf{U}}(^{\mathsf{U}}\forall\alpha\beta\gamma.\,^{\mathsf{L}}(^{\mathsf{U}}\mathsf{chan}\ \alpha \otimes {}^{\mathsf{L}}(\alpha \mathbin{@} (\uparrow\beta;\gamma)) \otimes \beta) \multimap {}^{\mathsf{L}}(\alpha \mathbin{@} \gamma)).$$

That is, for any channel α, type β, and session γ, *send* takes a linear triple having these components: an unlimited channel of type $^{\mathsf{U}}\mathsf{chan}\ \alpha$; a linear capability of type $^{\mathsf{L}}(\alpha \mathbin{@} (\uparrow\beta;\gamma))$, which means that the protocol of channel α requires sending a value of type β, followed by protocol γ; and a value of type β.

[7] There is one additional requirement beyond adding the right types and constants: The type system needs some way to check session type duality. A proof system for session duality is expressible in almost any polymorphic type system (Pucella and Tov 2008), but session types are more convenient to use if the type system proves duality automatically. I assume such a mechanism here.

It returns a linear capability of type $\mathsf{L}(\alpha @ \gamma)$, witnessing that γ is the remaining protocol for channel α.

2.4.3 Session Types Are ILL

While session types are expressible by adding the right types and constants to a linear type system, there is also a direct correspondence between formulae of ILL and session types (Caires and Pfenning 2010). In particular, session types (with binary choice) may be encoded in ILL as follows:[8]

$$\begin{aligned} [\![\uparrow\tau;S]\!] &= \tau \multimap [\![S]\!] & [\![\downarrow\tau;S]\!] &= \tau \otimes [\![S]\!] \\ [\![\uparrow T;S]\!] &= [\![T]\!] \multimap [\![S]\!] & [\![\downarrow T;S]\!] &= [\![T]\!] \otimes [\![S]\!] \\ [\![T \mathbin{\underline{\oplus}} S]\!] &= [\![T]\!] \mathbin{\&} [\![S]\!] & [\![T \mathbin{\underline{\&}} S]\!] &= [\![T]\!] \oplus [\![S]\!] \end{aligned}$$

A function of type $\tau \multimap [\![S]\!]$ represents a capability to consume (that is, send) a value of type τ, followed by $[\![S]\!]$; this clearly corresponds to a protocol which requires sending τ followed by $[\![S]\!]$. Similarly, a value of type $\tau \otimes [\![S]\!]$ represents the capability to produce values of types τ and $[\![S]\!]$, which is similar to receiving a value of type τ followed by protocol $[\![S]\!]$. A value of type $[\![T]\!] \mathbin{\&} [\![S]\!]$ represents the capability to produce either a value of type $[\![T]\!]$ or a value of type $[\![S]\!]$, at the client's option, which corresponds to the internal choice $T \mathbin{\underline{\oplus}} S$; dually for $[\![T]\!] \oplus [\![S]\!]$.

The exponential connective has a session type interpretation as well. We can represent the ability to get any number of channels at session S as a value of type $!{[\![S]\!]}$. This means that a rendezvous value of type $[S]$ may be represented as a value of linear type $!([\![S]\!] \mathbin{\&} [\![\overline{S}]\!])$, which can produce an arbitrary number of channels for protocol S or dual protocol $\overline{S}$.

Caires and Pfenning (2010) give a typed π calculus that uses the session type interpretation of linear types as the types of processes, and prove that communication does not go wrong. In more recent work, Toninho et al. (2011)

[8] This represents a view of session types as channel values, whereas Caires and Pfenning take the dual view of a session type as the type of a process that implements it. This means that they can translate $[\![T \mathbin{\underline{\oplus}} S]\!] = [\![T]\!] \oplus [\![S]\!]$, which may be more congenial but is less consistent with the presentation of session types heretofore.

extend the session type interpretation of linear logic to a dependent, linear type theory, yielding dependent session types.

CHAPTER 3

Programming in Alms

ALMS IS A typed, call-by-value, impure functional language with a full set of high-level language features: algebraic data types, pattern matching, open records and variants, reference cells, exceptions, global type inference, first-class polymorphism (via type annotations), and modules with opaque signature ascription. Unlike most other ML-like languages, Alms supports *affine types,* which allow it to express the stateful type systems of the previous chapter.

Alms provides several novel features that facilitate designing resource-aware abstractions. In this chapter, I introduce these features in a series of examples and argue, in support of my thesis, that these language features help make Alms a practical, general-purpose programming language.

3.1 Alms by Example

Consider a simple OCaml (Leroy et al. 2011) function, *deposit*, that updates one element of an array by adding an integer:

```
let deposit (arr : int array) (acct : int) (amt : int) =
  Array.set arr acct (Array.get arr acct + amt)
```

In a concurrent setting, function *deposit* suffers from a race condition between the read and the write. One way to solve this problem is to use a lock to enforce

```
module type AF_ARRAY = sig
  type 'a array : A
  val new : int → 'a → 'a array
  val set  : 'a array → int -A-> 'a -A-> 'a array
  val get  : 'a array → int -A-> 'a × 'a array
end
module AfArray : AF_ARRAY
```

Figure 3.1: Affine array interface in Alms

mutual exclusion:

```
let deposit (arr : int array) (acct : int) (amt : int) (lock : lock) =
  Lock.acquire lock;
  Array.set arr acct (Array.get arr acct + amt);
  Lock.release lock
```

Affine data. Locks can ensure mutual exclusion, but using them correctly is error-prone. A rather coarse alternative to ensure mutual exclusion is to forbid aliasing altogether. If we have the only reference to an array then no other process can operate on it concurrently.

Both versions of the OCaml function *deposit* are valid Alms functions as well. In Alms, unlike in OCaml, we can prohibit aliasing statically using the type system. We do this by declaring an interface that includes a new, abstract array type (figure 3.1). The base kinds of Alms are *use qualifiers,* which indicate how many times values of that type may be used. The syntax ": **A**" specifies that type *'a AfArray*.array has kind **A**, as in *affine*, which means that any attempt to duplicate a reference to such an array is a type error. Two points about the types of *AfArray.get* and *AfArray.set* are worth noting:

- Each function must return an array because the caller cannot reuse the reference to the array supplied as an argument.

- Function types of the form $\tau_1 \xrightarrow{\mathsf{A}} \tau_2$ have kind **A**, which means that

```
module AfArray : AF_ARRAY = struct
  type 'a array = 'a Array.array

  let new = Array.new
  let set arr ix v = Array.set arr ix v; arr
  let get arr ix = (Array.get arr ix, arr)
end
```

Figure 3.2: Affine array implementation in Alms

such a function can be applied at most once.[1] This is necessary because reusing a function partially applied to an affine value would reuse that value.

We can now rewrite *deposit* to use the AF_ARRAY interface:

```
let deposit (arr : int AfArray.array) (acct : int) (amt : int) =
  let (balance, arr) = AfArray.get arr acct in
    AfArray.set arr acct (balance + amt)
```

If we attempt to use an *AfArray*.array more than once, rather than single-threading it, we get a type error:

```
let deposit (arr : int AfArray.array) (acct : int) (amt : int) =
  let (balance, _) = AfArray.get arr acct in
    AfArray.set arr acct (balance + amt)
```

In this case, Alms reports that affine variable *arr* is duplicated.

Abstract affine types. Implementing *AfArray* is merely a matter of wrapping the primitive array type and operations and sealing the module with an opaque signature ascription (figure 3.2). The underlying, primitive array type, *'a Array*.array, has kind **U**, as in *unlimited*, because it places no limits on duplication. We can use it to represent an abstract type of kind **A**, however, because qualifier **U** is a subkind of qualifier **A**, and Alms's kind subsumption rule allows assigning an abstract type a greater kind than that of its concrete representation.

[1] It is tempting to call function types of kind **A** "affine," but in the standard terminology, an "affine function" is one that uses its argument at most once, not a function that itself may be used at most once, which is what the kind indicates. I refer to functions of kind **A** as "one-shot."

```
module type CAP_ARRAY = sig
  type ('a,'b) array
  type 'b arraycap : A
  val new : int → 'a → ∃'b. ('a,'b) array × 'b arraycap
  val set  : ('a,'b) array → int → 'a → 'b arraycap → 'b arraycap
  val get  : ('a,'b) array → int → 'b arraycap → 'a × 'b arraycap
  val dirtyGet : ('a,'b) array → int → 'a
end
```

Figure 3.3: Interface for unlimited arrays with affine capabilities

We need not change *new* at all, and *get* and *set* are modified slightly to return the array as required by the interface.

Affine capabilities. The affine array interface is quite restrictive. Because it requires single-threading an array through the program, it cannot adequately support operations that do not actually require exclusive access to the array. However, Alms supports creating a variety of abstractions to suit our needs. One way to increase the flexibility of the interface is to separate the reference to the array from the *capability* to read and write the array, in the same style as the capabilities used to implement typestate in λ^{URAL} (figure 2.5 on p. 23). Only the capability, not the array itself, needs to be affine.

For example, we may prefer an interface that supports "dirty reads," which do not require exclusive access but are not guaranteed to observe a consistent state, as in figure 3.3. In signature CAP_ARRAY, type ('a,'b) array is now unlimited and 'b arraycap is affine. Type constructor array's second parameter, 'b, is a "stamp" used to tie it to its capability, which must have type 'b arraycap (where type 'b matches). In particular, the type of *new* indicates that it returns an existential containing an array and a capability with matching stamps. The existential guarantees that the stamp on an array can only match the stamp on the capability created by the same call to *new*.

Operations *set* and *get* allow access to an array only when presented with the matching capability. This ensures that *set* and *get* have exclusive access

```
module CapArray : CAP_ARRAY = struct
  module A = Array

  type ('a,'b) array = 'a A.array
  type 'b arraycap = unit

  let new size init = (A.new size init, ())
  let set arr ix v _ = A.set arr ix v
  let get arr ix _ = (A.get arr ix, ())

  let dirtyGet = A.get
end
```

Figure 3.4: Implementation of unlimited arrays with affine capabilities

with respect to other *set* and *get* operations. They no longer return the array, but they do need to return the capability. On the other hand, *dirtyGet* does not require a capability and thus must not return one.

For example, the CAP_ARRAY interface allows shuffling an array while simultaneously computing an approximate sum:

```
let shuffleAndDirtySum arr cap =
  let th1 = Thread.fork (λ _ → inPlaceShuffle arr cap) in
  let th2 = Thread.fork (λ _ → dirtySumArray arr) in
    (Thread.wait th1, Thread.wait th2)
```

To implement CAP_ARRAY, we need suitable representations for its two abstract types. We represent CAP_ARRAY's arrays by the primitive array type, and capabilities by type unit, which is adequate because these capabilities have no run-time significance. Type unit has kind **U**, but as in the previous example, type abstraction subsumes it to **A** to match the kind of *'b CapArray*.arraycap. The implementation of the operations is in terms of the underlying array operations, with some shuffling to ignore capability arguments (in *set* and *get*) and to construct tuples containing value () to represent the capability in the result (in *new* and *get*).

Capabilities are values. Capabilities such as *'b CapArray*.arraycap often represent the state of a resource, but in Alms they are also ordinary values. They may be stored in immutable or mutable data structures, packed into

exceptions and thrown, or sent over communication channels like any other value. For example, suppose we would like a list of array capabilities. Lists are defined thus in the standard library:

```
type 'a list = (::) of 'a × 'a list | []
```

The type variables we have seen until now could only be instantiated with unlimited types, but the type variable in the definition of `'a list` is different, because it may be instantiated to any type, unlimited or affine. Type variables in Alms indicate their kind lexically. Unlimited type variables, of kind U, are written with a normal quotation mark: `'a 'b 'c`. Affine type variables, of kind A, are written with a backquote: `` `a `b `c ``.

Whether a list is affine or unlimited depends on whether the elements of the list are affine or unlimited. Alms represents this fact by giving the list type constructor a dependent kind, where kind $\langle$`'a`$\rangle$ stands for the actual kind of type variable `'a`:

```
list : Π('a+). ⟨'a⟩
```

That is, the kind of a list is the same as the kind of its element type: Type `int list` has kind U, whereas `'b CapArray.arraycap list` has kind A. (The + is a variance annotation that indicates that `list` is covariant in its parameter.)

In the concrete syntax used for abstract types in Alms signatures, the kind of `list` is written thus:

```
type +'a list : 'a
```

Abstract type declarations must include kind annotations in that form (except where the kind is the default, U), and the kind of an abstract type must be greater than or equal to the kind of its concrete implementation during signature matching. For concrete type definitions, however, Alms infers a principal kind. In general, the kind of a type is the join of the kinds of the types that occur directly in its representation.

Some examples of concrete type definitions and their inferred kinds appear in figure 3.5. Because both `'a` and `'b` are part of the representation of `('a,'b) r`, it must be affine if either of its parameters is affine, hence its kind is the least upper bound of the kinds of the type parameters: `'a` $\vee$ `'b`. On the other hand,

```
type ('a,'b) r  = 'a × 'b                    : 'a ∨ 'b
type ('a,'b) s  = int × 'b                   :      'b
type ('a,'b) t  = T1 of 'a | T2 of ('b,'a) t : 'a ∨ 'b
type ('a,'b) u  = U1 | U2 of ('b,'a) u       :    U
type ('a,'b) v  = 'a × (unit → 'b)           : 'a
type ('a,'b) w  = 'a × (unit -'b-> unit)     : 'a ∨ 'b
```

Figure 3.5: Some type definitions and inferred qualifier kinds

phantom parameter $\text{`}a$ is not part of the representation of type ($\text{`}a$,$\text{`}b$) s, so that has kind $\text{`}b$.

Types t and u are recursive, which means that they are part of their own representations. Following the same rule that the kind of a type is the join of the kinds of its possible representations, this sets up a flow equation for each type:

$$\begin{aligned}
\mathscr{T}(\text{`}a,\text{`}b) &= \text{`}a && \text{(by constructor } T_1 \text{ of `}a\text{)}\\
&\quad \vee\ \mathscr{T}(\text{`}b,\text{`}a) && \text{(by constructor } T_2 \text{ of (`}b\text{,`}a\text{) t)}\\
\mathscr{U}(\text{`}a,\text{`}b) &= \mathsf{U} && \text{(by constructor } U_1\text{)}\\
&\quad \vee\ \mathscr{U}(\text{`}b,\text{`}a) && \text{(by constructor } U_2 \text{ of (`}b\text{,`}a\text{) t)}
\end{aligned}$$

The kinds of ($\text{`}a$,$\text{`}b$) t and ($\text{`}a$,$\text{`}b$) u are given by the least fixpoints of $\mathscr{T}$ and $\mathscr{U}$, respectively, which are easily found by iteration.

Type variable $\text{`}b$ does not appear in the kind of ($\text{`}a$,$\text{`}b$) v because the domain and codomain of a function type are not part of the function's representation. Instead, function types carry their kind in a superscript as in the definition of type ($\text{`}a$,$\text{`}b$) w. This generalizes the one-shot function type ($\xrightarrow{\mathsf{A}}$) that we have seen already to allow any base kind on a function arrow. (All the superscript-free arrows ($\rightarrow$) that we have seen thus far have stood for the unlimited arrow ($\xrightarrow{\mathsf{U}}$). Alms fills in omitted arrow superscripts using a rule described in §3.2.2.)

Principal promotion. Unlike other type constructors, the kind of a function type cannot be derived from the domain and codomain. Whether a function is safe to duplicate depends on whether values referenced from its closure are

safe to duplicate. Thus, the qualifier kind of a function type must upper bound the kinds of the types of the function's free variables. Function types carry this kind as a third parameter.

For example, $\mathsf{int} \xrightarrow{\mathsf{A}} \mathsf{int}$ is the type of an integer-to-integer function that cannot be duplicated and may be applied only once. Some functions have more elaborate types. For example, Alms infers that $\mathbf{\lambda}\ x\ y \to x$ has type $\text{`}a \xrightarrow{\mathsf{U}} \text{`}b \xrightarrow{\text{`}a} \text{`}a$. (Using implicit arrow superscript rules described in §3.2.2, the type may be written $\text{`}a \to \text{`}b \to \text{`}a$.) The second arrow has superscript $\text{`}a$ because a partial application of this function closes over a value of type $\text{`}a$, and thus must be as restricted as that value.

In general, the qualifier inferred for a function type is the least upper bound of the qualifiers of the types of its free variables. This is a generalization of the promotion rule of ILL, whereby a proposition may be made unlimited if all the resources from which it is derived are unlimited (Bierman 1993). Theorem 5.3 on p. 118 shows that my model of Alms always infers the least kinds for function types

Dereliction subtyping. The principal promotion property guarantees that every inferred function type has the least (safe) qualifier kind. What makes this property especially useful is that Alms extends the subkinding relation to subtyping between functions. This is safe because a function's use invariant cannot be violated by using it in a context where its actual treatment will be *more restricted* than required by its qualifier.

For example, consider an operation for starting a new thread:

$\mathsf{val}\ \mathit{Thread.fork} : (\mathsf{unit} \xrightarrow{\mathsf{A}} \mathsf{unit}) \xrightarrow{\mathsf{U}} \mathit{Thread}.\mathsf{thread}$

The type promises that *Thread.fork* will apply its argument function only once, but there is no reason that we should not be able to pass it a function that is allowed to be applied more than once. We may do so because type $\mathsf{unit} \xrightarrow{\mathsf{U}} \mathsf{unit}$ is a subtype of $\mathsf{unit} \xrightarrow{\mathsf{A}} \mathsf{unit}$.

Alms extends the subtyping relation over functions in the usual way, as well as through algebraic and abstract data types based on the variances of their type parameters as reflected in their kinds (as in the variance annotation

+ in the kind of list). Compared to a strict stratification between unlimited and affine types, dereliction subtyping provides many opportunities for code reuse.

The kinds of type variables. As described above, type variables in Alms may be unlimited (’a) or affine (‘a). Alms infers which kind to give each type variable based on its usage. As an example of how the kinds of type variables work in Alms, consider these two function definitions. Given the definitions

```
let swap (x, y) = (y, x)
let dup (x, _)  = (x, x)
```

Alms infers the following types:

```
val swap : ‘a × ‘b → ‘b × ‘a
val dup  : ’a × ‘b → ’a × ’a
```

Both functions destructure a pair and return a pair, but they treat the components of the pair differently. Because *swap* uses each component once in its result, Alms infers that both components have affine types. On the other hand, *dup* uses the first component of the pair twice and discards the second component. The first component must therefore be unlimited, but the second component can be affine, since it is not used more than once. Thus, *swap* may be applied to any pair (by subkinding), but *dup* must be applied to pair whose first component is unlimited.

Qualifier inference extends smoothly to other language features, such as algebraic data types, as well. Consider, for example, the standard definition of a left fold for lists:

```
let rec foldl f z xs = match xs with
  | x :: xs' → foldl f (f x z) xs'
  | []       → z
```

Alms infers type $(\text{‘}a \to \text{‘}b \xrightarrow{\mathsf{A}} \text{‘}b) \to \text{‘}b \to \text{‘}a\ \mathsf{list} \xrightarrow{\text{‘}b} \text{‘}b$ for *foldl*. Both type variables are affine, because *foldl* duplicates neither the elements of the list nor z. We can understand the arrow qualifiers as follows. The first argument has type $\text{‘}a \to \text{‘}b \xrightarrow{\mathsf{A}} \text{‘}b$. This type means that f is unlimited, but partial applications

```
module type CAP_ARRAY = sig
  type ('a,'b) array
  type 'b arraycap : A

  val new : int → (int → 'a) → ∃'b. ('a,'b) array × 'b arraycap

  val set : ('a,'b) array → int → 'a → 'b arraycap -'a→ 'b arraycap
  val get : ('a,'b) array → int → 'b arraycap → 'a × 'b arraycap
  val swap : ('a,'b) array → int → 'a → 'b arraycap -'a→ 'a × 'b arraycap

  val dirtyGet : ('a,'b) array → int → 'a
end
```

Figure 3.6: Interface for arrays with potentially affine elements

of f are affine, because *foldl* duplicates f to apply it to each element of the list but does not duplicate partial applications of f. The last arrow in the type of *foldl* has kind $\text{`}b$, because duplicating the partial application *foldl f z* effectively duplicates z.

By contrast, consider the definition of *scanl*, a variant of *foldl* that accumulates a list of all the intermediate results:

```
let rec scanl f z xs = match xs with
  | x :: xs' → z :: scanl f (f x z) xs'
  | []       → [z]
```

Alms infers type $(\text{`}a \to \text{'}b \xrightarrow{\mathsf{A}} \text{'}b) \to \text{'}b \to \text{`}a\ \mathsf{list} \to \text{'}b\ \mathsf{list}$ for *scanl*. Type variable $\text{'}b$ is now inferred to be unlimited because *scanl* uses z twice in the first case. The last arrow in the type is now unlimited as well, because duplicating the (well typed) partial application *scanl f z* is always safe, because f and z are unlimited.

The kinds of Alms type variables and the subkinding relation can also be used to make interfaces more precise. For example, figure 3.6 contains an improved version of the CAP_ARRAY signature that supports arrays of affine elements. Note that the kind of the array elements in the types of *get* and *dirtyGet* is unlimited, whereas array elements for *new*, *set*, and *swap* may be affine. Thus, all operations in the signature are permitted for arrays

```
module type CAP_LOCK_ARRAY = sig
  include CAP_ARRAY

  val new : int → (int → 'a) → ∃'b. ('a,'b) array
  val acquire : ('a,'b) array → 'b arraycap
  val release : ('a,'b) array → 'b arraycap → unit
end
```

Figure 3.7: Interface to arrays with capabilities and locks

with unlimited elements, but read operations, which effectively duplicate an element by both leaving it in the array and returning it, may be used only on arrays with unlimited elements. To read from arrays with affine elements, one must use the new function *swap*, which takes a new element to swap in place of the element that it reads.

More possibilities. The type system of Alms is sufficiently flexible to express a wide variety of stateful interface designs. For example, figure 3.7 contains an interface that mixes dynamic locking with the static capabilities of CAP_ARRAY. In signature CAP_LOCK_ARRAY, function *new* returns an array (with unique tag *'b*) but no capability. As before, array operations require a capability, and a capability is obtained by requesting it using *acquire*. Subsequent attempts to acquire a capability for the same array block until the capability is *release*d. Thus, access to the capability is granted dynamically, and the capability, once granted, is tracked statically. Since accessing the array or releasing the lock requires the capability, it is not possible to access the array without holding the lock or to release the lock when one does not own it.

The ability to store affine capabilities in data structures makes it possible to implement signature CAP_LOCK_ARRAY in terms of *CapArray* without any privileged knowledge about the representation of *CapArray*.arraycap. The implementation relies on mvars (signature in figure 3.8), which are synchronized variables based on Id's *M-structures* (Barth et al. 1991). An *'a* mvar may hold a value of type *'a* or it may be *empty*. While an mvar may contain an affine value, the mvar itself is always unlimited. This is safe because calling *MVar.take* on

```
module MVar : sig
  type 'a mvar

  val new : 'a → 'a mvar
  val take : 'a mvar → 'a
  val put : 'a mvar → 'a → unit
end
```

Figure 3.8: Interface to mvars (synchronized variables)

```
module CapLockArray : CAP_LOCK_ARRAY = struct
  module A = CapArray

  type 'b arraycap = 'b A.arraycap
  type ('a,'b) array = ('a,'b) A.array × 'b arraycap mvar

  let new size init =
    let (arr, cap) = A.new size init in
      (arr, MVar.new cap)

  let acquire (_, mv) = MVar.take mv
  let release (_, mv) cap = MVar.put mv cap

  let set (arr, _) = A.set arr
  let get (get, _) = A.get arr
  ...
end
```

Figure 3.9: Implementation of to arrays with capabilities and locks

a non-empty mvar removes the value and returns it, while *MVar.take* on an empty mvar blocks until another thread *MVar.put*s a value into it.

To implement CAP_LOCK_ARRAY, we represent an array as a pair of the underlying ('*a*,'*b*) *CapArray*.array and an mvar to store its capability (figure 3.9). The *new* operation creates a new array-capability pair, stores the capability in a new mvar, and returns the array and the mvar as a pair. Operations *acquire* and *release* use the mvar component of the pair, while the old operations such as *set* must be lifted to project the underlying *CapArray*.array out of the pair.

There are many more possibilities. Figures 3.10 and 3.11 show two

```
module type RW_LOCK = sig
  type ('a,'b) array
  type excl
  type shared
  type 'b@'c : A

  val new : int → (int → 'a) → ∃'b. ('a,'b) array

  val acquireW : ('a,'b) array → 'b@excl
  val acquireR : ('a,'b) array → 'b@shared
  val release : ('a,'b) array → 'b@'c → unit

  val set : ('a,'b) array → int → 'a → 'b@excl -'a-> 'b@excl
  val get : ('a,'b) array → int → 'b@'c → 'a × 'b@'c
  val swap : ('a,'b) array → int → 'a → 'b@excl -'a-> 'a × 'b@excl
end
```

Figure 3.10: Reader-writer locks with capabilities

```
module type FRACTIONAL = sig
  type ('a,'b) array
  type 1
  type 2
  type 'c/'d
  type ('b,'c) arraycap

  val new : int → (int → 'a) → ∃'b. ('a,'b) array × ('b,1) arraycap

  val split : ('b,'c) arraycap → ('b,'c/2) arraycap × ('b,'c/2) arraycap
  val join : ('b,'c/2) arraycap × ('b,'c/2) arraycap → ('b,'c) arraycap

  val set : ('a,'b) array → int → 'a → ('b,1) arraycap -'a-> ('b,1) arraycap
  val get : ('a,'b) array → int → ('b,'c) arraycap → 'a × ('b,'c) arraycap
  val swap : ('a,'b) array → int → 'a → ('b,1) arraycap -'a-> 'a × ('b,1) arraycap
end
```

Figure 3.11: Fractional reader-writer capabilities

interfaces for reader-writer capabilities, which at any one time grant either exclusive read-write access or shared read-only access.

Signature RW_LOCK (figure 3.10) describes dynamic reader-writer locks. The signature declares types excl and shared and an affine, binary type constructor $(\cdot \texttt{@} \cdot)$. Capabilities have type *'b*@*'c*, where *'b* ties the capability to a particular array and *'c* records whether the held lock is exclusive or shared. Operations *set* and *swap* require an exclusive lock (*'b*@excl), but *get* allows *'c* to be excl or shared.

Signature FRACTIONAL (figure 3.11) describes fractional reader-writer capabilities (Boyland 2003). As in the previous example, the capability type (*'b*,*'c*) arraycap has a second parameter, which in this case tracks what fraction of the whole capability is held. The fraction is represented using type constructors 1, 2, and $(\cdot/\cdot)$. A capability of type (*'b*,1) arraycap grants exclusive access to the array with tag *'b*, while a fraction less than 1 such as $1/2$ or $1/2/2$ indicates shared access. For managing capabilities, function *split* divides a capability whose fraction is *'c* into two capabilities of fraction *'c*/2, and *join* combines two *'c*/2 capabilities back into one *'c* capability. Again, *set* and *swap* require exclusive access but *get* does not.

3.2 Syntax Matters

The design of Alms balances two sometimes contradictory goals: sufficient expressiveness to support a wide variety of resource management disciplines, but without making the language too unwieldy to use. Type system features such as dependent kinds and dereliction subtyping help with the latter, because they make types and functions applicable in more cases. Another way to increase usability is to decrease noise by optimizing the concrete syntax for the common case. In this section, I describe two such syntactic optimizations.

3.2.1 Implicit Threading Syntax

Given *CapLockArray* as defined above, we can rewrite function *deposit* (the initial example in this chapter) to take advantage of it:

```
open CapLockArray
let deposit arr acct amt =
  let cap             = acquire arr in
  let (balance, cap) = get arr acct cap in
  let cap             = set arr acct (balance + amt) cap in
  release arr cap
```

While this gets the job done, the explicit threading of the capability is inconvenient to write and hard to read. To address this, Alms supports an alternate syntax for implicit threading of values:

```
let deposit arr acct amt =
  let ↯cap = acquire arr in
  set arr acct (get arr acct cap + amt) ▷ cap;
  release arr ◁ cap
```

Pattern ↯*cap* binds *cap* not as an ordinary variable, but as an *implicitly threaded variable*. This initiates a syntactic transformation that automatically single-threads *cap* through the code in its scope, based on the assumption that a function taking an implicitly threaded variable (or tuple of such variables) as an argument will return a pair of the function's *direct* result and a new value for the implicitly threaded variable. Function *get* follows this convention, as it returns a pair of the value read from the array and the array capability. Functions *set* and *release*, however, do not return pairs: *set* returns the capability only, and *release* consumes the capability and returns unit. Infix operators (▷) and (◁) are not treated specially by the transformation, but are ordinary functions that lift other functions to follow the return convention:

```
let (▷) f x = ((), f x)
let (◁) f x = (f x, ())
```

That is, we use (▷) to lift a function that returns only the new threaded variable value to return () as its direct result, as with function *set*, which returns only a capability, not a pair. We use (◁) to lift a function that returns only a direct result to return () as the new value for the threaded variable, as with function *release*, which consumes a capability but does not return it.

The transformation works by locating uses of implicitly threaded variables and rebinding function results make the new values of implicitly threaded

variables available for subsequent uses. In the case of *deposit*, the implicit threading transformation yields this rewritten definition:[2]

```
let deposit arr acct amt =
  let cap        = acquire arr in
  let (r1, cap) = let (r2, cap) = get arr acct cap in
                  set arr acct (r2 + amt) ▷ cap in
  release arr ◁ cap
```

Because the transformation happens on syntax before type checking, it cannot compromise type safety.

The implicit threading transformation handles a variety of language features, including patterns that bind multiple implicitly threaded variables and pattern matching on implicitly threaded variables. An especially interesting case is how it handles functions with free implicitly threaded variables, by adding such variables as parameters and results, and modifying uses of such functions to thread the variables. For example, function *deposits* takes an array, a list of accounts, and a list of amounts to deposit:

```
let deposits arr accts0 amts0 =
  let ↯cap = acquire arr in
  let rec loop accts amts =
    match (accts, amts) with
    | (acct :: accts', amt :: amts') →
        set arr acct (get arr acct cap + amt) ▷ cap;
        loop accts' amts'
    | _ → release ◁ cap in
  loop accts0 amts0
```

The capability is bound to the implicitly threaded variable *cap*, which then appears free in the body of function *loop*. The transformation adds *cap* as a

[2]The implemented transformation generates several administrative redexes omitted here.

parameter to *loop* and threads it through calls to *loop*:

```
let deposits arr accts0 amts0 =
  let cap = acquire arr in
  let rec loop accts amts cap =
    match (accts, amts) with
    | (acct :: accts', amt :: amts') →
        let (r1, cap) =
          let (r2, cap) = get arr acct cap in
          set arr acct (r2 + amt) ▷ cap in
        loop accts' amts' cap
    | _ → release ◁ cap in
  loop accts0 amts0 cap
```

Several more examples of the implicit threading syntax appear in chapter 4.

3.2.2 Arrow Qualifier Inference

Dependent kinds eliminate the need for almost all qualifiers from the syntax of types, with one major exception: function types. In the code examples in this chapter thus far, I have often omitted writing qualifiers on function types when the qualifier is U (though sometimes I included it for emphasis). The simple rule that U superscripts may be omitted means that most arrows do not require superscripts, because most functions, in practice, are unlimited. However, Alms actually uses a stronger heuristic for filling in missing function type qualifiers that allows omitting around two-thirds of the non-U arrow superscripts, while adding very few required U superscripts.[3]

Alms's rule for filling in missing function qualifiers relies on a simple observation about the common case in function types: For a curried function that uses all its arguments, each argument is part of the closure of the partial application through that argument (or equivalently, each argument is in the free variables of its λ's body), which means that by the principal promotion

[3] In order to select a qualifier inference rule, I tested five candidate rules against Alms's standard library and evaluated the rules for efficacy and predictability. Using the rule that I chose, there was exactly one case of a mandatory U superscript in the corpus of Alms types that I analyzed. All types written to this point actually conform to *both* Alms's actual rule and the simple missing-means-U rule, because the qualifier superscripts required by the latter rule have been a subset of the former.

rule, the kind of each argument becomes part of the qualifier of all arrows subsequent to the arrow immediately following the argument. That is, arrow types typically follow a pattern like this:

$$\text{`}a \to \text{`}b \xrightarrow{\text{`}a} \text{`}c \xrightarrow{\text{`}a \vee \text{`}b} \text{`}d \xrightarrow{\text{`}a \vee \text{`}b \vee \text{`}c} \ldots$$

Of course, not all arrow types follow such a pattern, but in practice, most do. The inferred type will follow such a pattern when the function has this form (where all the arguments are used in the body):

$$\boldsymbol{\lambda}\ w\ x\ y\ z \ldots.$$

Functions with more unusual forms have types that do not follow the pattern:

$$\boldsymbol{\lambda}\ w\ x.\ \mathsf{let}\ r = \mathsf{aref}\ w\ \mathsf{in}\ \boldsymbol{\lambda}\ y.\ \mathsf{let}\ w' = \mathit{swap}\ r\ ()\ \mathsf{in}\ \boldsymbol{\lambda}\ z \ldots.$$
$$: \text{`}a \to \text{`}b \xrightarrow{\mathsf{A}} \text{`}c \xrightarrow{\text{`}a \vee \text{`}b} \text{`}d \xrightarrow{\text{`}a \vee \text{`}b \vee \text{`}c} \ldots$$

However, the goal is to optimize the syntax for the common case, and the simple function $\boldsymbol{\lambda}\ w\ x\ y\ z \ldots.$ is the common case.

The rule. The actual rule for filling missing function type qualifiers in type annotations relies on inspecting the context. Given a sequence of superscript-free arrows (as in the type of a curried function), the qualifier of each arrow is the join of the kinds of the arguments before that arrow, as in the pattern above. When some arrow has an explicit superscript, that supersedes the kinds of prior arguments and is included in the join instead. To be precise, given an arrow type like

$$\mathsf{t}_1 \xrightarrow{\mathsf{q}_1} \mathsf{t}_2 \xrightarrow{\mathsf{q}_2} \cdots \xrightarrow{\mathsf{q}_{k-2}} \mathsf{t}_{k-1} \xrightarrow{\mathsf{q}_{k-1}} \mathsf{t}_k \xrightarrow{?} \cdots$$

where each q_i may be missing or present, the qualifier at ? is inferred as follows. Choose the largest i such that q_j is missing for all $j > i$; in case $i = 0$, consider q_0 to be U. Then the inferred qualifier is

$$\mathsf{q}_i \vee \langle \mathsf{t}_i \rangle \vee \cdots \vee \langle \mathsf{t}_{k-1} \rangle,$$

where $\langle \mathsf{t} \rangle$ stands for the kind of t.

Some examples. A comparison of several types written using two qualifier inference rules appears in figure 3.12. On the left, types are written using the simple rule that all missing superscripts are replaced with qualifier U. On the

Implicit U Rule	**Actual Rule**	
$'a$ AfArray.array → int $\xrightarrow{\mathsf{A}}$ $'a$ $\xrightarrow{\mathsf{A}}$ $'a$ AfArray.array	$'a$ AfArray.array → int → $'a$ → $'a$ AfArray.array	(1)
($`a$,$'b$) RWLock.array → int → $`a$ → $'b$@excl $\xrightarrow{`a}$ $'b$@excl	($`a$,$'b$) RWLock.array → int → $`a$ → $'b$@excl → $'b$@excl	(2)
(unit $\xrightarrow{\mathsf{A}}$ unit) → Thread.thread	(unit $\xrightarrow{\mathsf{A}}$ unit) → Thread.thread	(3)
($`a$ → $`b$ $\xrightarrow{\mathsf{A}}$ $`b$) → $`b$ → $`a$ list $\xrightarrow{`b}$ $`b$	($`a$ → $`b$ $\xrightarrow{\mathsf{A}}$ $`b$) → $`b$ → $`a$ list → $`b$	(4)
$`a$ → $`b$ $\xrightarrow{`a}$ $`c$ $\xrightarrow{`a \vee `b}$ $`d$ $\xrightarrow{`a \vee `b \vee `c}$ ⋯	$`a$ → $`b$ → $`c$ → $`d$ → ⋯	(5)
($`a$ $\xrightarrow{\mathsf{A}}$ $`b$) → $`a$ → $`b$	($`a$ $\xrightarrow{\mathsf{A}}$ $`b$) → $`a$ $\xrightarrow{\mathsf{U}}$ $`b$	(6)

Figure 3.12: Comparison of missing-means-U rule to actual rule

right, types are written using the actual rule in Alms, which takes advantage of the observation about the common patterns of qualifiers on function types. The first five rows contains types that appeared earlier in this chapter, and the sixth row contains an interesting type from the standard basis library.

In row (1), the two **A** superscripts on the left are not necessary on the right, because the kind of $'a$ AfArrayarray is **A**, and that propagates to subsequent arrows. Row (2) is similar, in that the superscript on the final arrow is $`a$ on the left but omitted on the right, because it is implied by the argument of type $`a$.

The types are the same on both sides of row (3), because the superscript **A** is not inferred by the rule—it is specific to the invariant that *Thread.fork* applies its argument only once. Row (4) has two superscripts on the left and one on the right. The first superscript, **A**, is again required, because the implied superscript at that point is $`a$, but *foldl* makes a stronger guarantee for how it uses its first argument. The second superscript is implied by the argument of type $`b$, so it is omitted on the right.

Row (5) shows the prototypical case that suggests the rule, which allows all superscripts to be omitted.

Finally, row (6) has the type of a function from the standard library that coerces a one-shot function to an unlimited function by adding a dynamic check (as described in chapter 7). This is an atypical case where the right

column requires a superscript that the left does not. The actual rule requires superscript U because the affine argument type would otherwise imply A on the final arrow as well. Of course, the library function of this type is doing something unusual, so it is reasonable that the type should appear unusual as well.

CHAPTER 4

Expressiveness of Alms

THE STATEFUL TYPE systems introduced in chapter 2 are similar to one another in that they all rely on some notion of linear or affine resources. They are not, however, generally interchangeable or interexpressible. Despite the similarity between session types and typestate, session types are not cleanly expressible in Vault, and Vault-style typestate protocols become awkward when encoded using session types. Similarly, while Vault offers region-based memory management, it does not provide the flexibility with respect to aliasing possible in the Calculus of Capabilities.

The language Vault (DeLine and Fähndrich 2001) provides one, fixed notion of typestate; the language Sing# (Fähndrich et al. 2006) provides one, fixed notion of session types. Alms has no built-in typestate or session types mechanisms, but both can be implemented—in multiple ways. In this chapter, I show how Alms can elegantly express the examples from chapter 2.

4.1 Typestate

Several of the Alms code examples in the previous chapter are reminiscent of typestate. In this section, I describe a more extended typestate example implementing the Berkeley sockets API introduced in §2.2, which ensures that socket setup operations follow the correct protocol. In the Alms version of the interface, I include both the client and server protocols for setting up a socket, and I show how to handle error states.

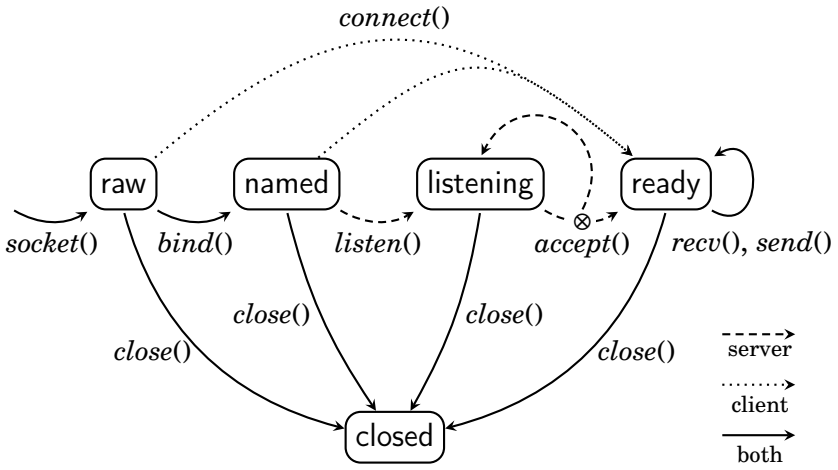


Figure 4.1: States and transitions for Berkeley sockets TCP

```
module type SOCKET_CAP = sig
  type 'a socket
  type 'a @ 'c : A
  type raw
  type named
  type listening
  type ready

  val socket  : unit → ∃'a. 'a socket × 'a@raw
  val bind    : 'a socket → int → 'a@raw → 'a@named
  val connect : 'a socket → string → string →
                  'a@raw + 'a@named → 'a@ready
  val listen  : 'a socket → 'a@named → 'a@listening
  val accept  : 'a socket → 'a@listening →
                  (∃'b. 'b socket × 'b@ready) × 'a@listening
  val send    : 'a socket → string → 'a@ready → 'a@ready
  val recv    : 'a socket → int → 'a@ready → string × 'a@ready
  val close   : 'a socket → 'a@'c → unit

  ⟨error handling in figure 4.3⟩
end
```

Figure 4.2: Alms interface to Berkeley sockets TCP (i): basic operations

The interface. A diagram of states and transitions for Berkeley sockets appears in figure 4.1 (reprinted from §2.2), and the corresponding Alms signature (deferring details of error handling) appears in figure 4.2. We might compare the Alms signature to the same interface written in Vault (figure 2.3 on p. 18) and λ^{URAL} (figure 2.5 on p. 23). For example, here are the types of *accept* as written in the three languages:

(Vault)

$$\textsf{tracked}(N)\ \textsf{sock}\ \mathit{accept}(\textsf{tracked}(S)\ \textsf{sock})\ \ [S@\textsf{listening}, \textsf{new}\ N@\textsf{ready}];$$

(λ^{URAL})

$$\mathit{accept} : {}^{\mathsf{U}}\forall\alpha{:}\textsc{type}.\,\textsf{sock}\ \alpha \xrightarrow{\mathsf{U}} {}^{\mathsf{L}}(\alpha@\textsf{listening}) \multimap^{\mathsf{U}} {}^{\mathsf{L}}({}^{\mathsf{L}}(\exists\beta{:}\textsc{type}.\ {}^{\mathsf{L}}({}^{\mathsf{U}}(\textsf{sock}\ \beta) \otimes {}^{\mathsf{L}}(\beta@\textsf{ready}))) \otimes {}^{\mathsf{L}}(\alpha@\textsf{listening}))$$

(Alms)

$$\textsf{val}\ \mathit{accept} : {'a}\ \textsf{socket} \to {'a}@\textsf{listening} \to (\exists {'b}.\ {'b}\ \textsf{socket} \times {'b}@\textsf{ready}) \times {'a}@\textsf{listening}$$

In Vault, *accept* takes a socket and returns a new socket, clearly different from the argument socket because it is tracked by a different key. The state change appears as a side condition, which indicates the (unchanged) state of the old socket and the initial state of the new socket. Neither λ^{URAL} nor Alms has a built-in notion of tracked keys and states, so instead the states of the sockets are witnessed by linear or affine values whose types reflect the state of the socket. Unlike Vault, where @ is part of the primitive syntax for states and keys, in λ^{URAL} and Alms @ is an ordinary infix, binary type constructor.[1] Both λ^{URAL} and Alms use essentially the same encoding of typestate, whereby a socket is associated with its capability using an existentially quantified type variable as a parameter to both the socket value and the capability. Between λ^{URAL} and Alms the major difference is one of legibility, since the type in Alms does not require qualifier annotations as it does in λ^{URAL}.

Error handling. DeLine and Fähndrich (2001) point out that in a realistic Vault interface to Berkeley sockets, operations should return a variant value

[1] λ^{URAL}, being a core model, includes no facility for defining a type within the language, but I introduce @ extralinguistically as an abstract type constructor by giving its kind. Thus λ^{URAL} does not treat it at all specially.

```
type 'a dynamicCap = Raw       of 'a@raw
                   | Named     of 'a@named
                   | Listening of 'a@listening
                   | Ready     of 'a@ready
exception Socket of (∃'a. 'a socket × 'a dynamicCap) option × string

val isSame       : 'a socket → 'b socket → ('a@'c → 'b@'c) option

val catchRaw     : (unit -A-> 'r) → 'a socket × ('a@raw -A-> 'r) → 'r
val catchNamed   : (unit -A-> 'r) → 'a socket × ('a@named -A-> 'r) → 'r
val catchListening : (unit -A-> 'r) → 'a socket × ('a@listening -A-> 'r) → 'r
val catchReady   : (unit -A-> 'r) → 'a socket × ('a@ready -A-> 'r) → 'r
```

Figure 4.3: Alms interface for TCP (ii): error handling

indicating success or failure. This means every socket operation must be followed by an explicit pattern match to check for an error. Such an interface is easily expressible in Alms as well, but I choose a different approach: Berkeley sockets operations in this Alms interface signal errors by raising exceptions, which contain the resources necessary to recover the prior state of the socket.

The portion of signature SOCKET_CAP for error handling appears in figure 4.3. The signature begins with type 'a dynamicCap and exception *Socket*, which are used by socket operations to signal errors. In particular, type 'a dynamicCap is a four-way sum over the four possible capability states, which allows dynamically checking which state a socket is in. When a socket operation fails, it raises a *Socket* exception. In cases where no socket yet (or still) exists, the exception contains only an error message, but in other cases, the exception carries an existentially quantified pair of a socket and its dynamic state, which can be used to recover from the failed operation.

Because the socket and capability are existentially quantified in the exception, this means that the association of the capability with any other references to the socket in the program is lost. In order to re-associate the capability with an existing socket type, the signature provides an operation *isSame*, which dynamically checks whether two sockets with apparently different type tags,

'a and *'b*, are in fact identical. If the sockets are the same, then *isSame* returns a coercion to tie a capability typed for socket *'a* to work instead with (the same socket) *'b*.

The basic error-handling interface described thus far can be inconvenient, because when catching an exception it requires checking both which socket it goes with and what state the socket is in. The last four functions in the signature specify a higher-level error-handling interface that supports catching exceptions containing a specified socket in a specified state. For example, *catchRaw* can be used as follows:

$$\begin{array}{l} \mathit{catchRaw} \\ \quad (\boldsymbol{\lambda}\ \_ \rightarrow e_{\text{body}}) \\ \quad (\mathit{sock}, \boldsymbol{\lambda}\ \mathit{cap} \rightarrow e_{\text{handler}}) \end{array}$$

Function *catchRaw* evaluates expression e_{body} in the context of an exception handler that catches exceptions for socket *sock* in state raw, in which case it invokes the handler with *cap* as the raw socket capability. There are three additional functions for catching exceptions in the three other socket states.

The implementation. An implementation of signature SOCKET_CAP appears in figure 4.4. It begins with type declarations for sockets, capabilities, and states. Sockets are represented by an underlying, non-typestate socket type *S*.socket; capabilities, as usual, are represented by type unit, since they need not represent any information at run time. The four types representing socket states, raw, named, listening, and ready, are purely abstract. Because the parameters to the capability type *'a@'c* are phantom, the states themselves need not be represented at all.

Type dynamicCap and exception *Socket*, both used for error handling, are described above.

The first operation, *socket*, uses function *S.socket* provided by the underlying, non-typestate sockets structure, to create a socket, and if successful returns it in a pair along with value () representing the capability. If *S.socket* fails, it raises an *IOError* exception carrying an error message; *socket* catches that exception and throws instead a *Socket* exception with the same message. The exception also contains *None*, because there is no socket yet to include as the first component of the pair.

```
module SocketCap : SOCKET_CAP = struct
  module S = Socket

  type 'a socket        = S.socket
  type 'a @ 'c          = unit
  type raw and named and listening and ready

  type 'a dynamicCap = Raw       of 'a@raw
                     | Named     of 'a@named
                     | Listening of 'a@listening
                     | Ready     of 'a@ready

  exception Socket of (∃'a. 'a socket × 'a dynamicCap) option × string

  let socket _ : ∃'a. 'a socket × 'a@raw =
    try (S.socket (), ())
    with IOError msg → raise (Socket (None, msg))

  let lift thunk sock mkcap =
    try thunk ()
    with IOError msg → raise (Socket (Some (sock, mkcap ()), msg))

  let bind sock port _   = lift (λ _ → S.bind sock port) sock Raw
  let listen sock _      = lift (λ _ → S.listen sock) sock Named
  let accept sock _      =
    lift (λ _ → ((S.accept sock, ()) : ∃'a. 'a socket × 'a@ready, ()))
         sock Listening

  let connect sock host port cap =
    lift (λ _ → S.connect sock host port)
         sock (match cap with Left _ → Raw | Right _ → Named)

  let send sock data _   = lift (λ _ → S.send sock data; ()) sock Ready
  let recv sock len _    = lift (λ _ → (S.recv sock len, ())) sock Ready
  let close sock _       =
    try S.close sock
    with IOError msg → raise (Socket (None, msg))

  ⟨error handling in figure 4.5⟩
end
```

Figure 4.4: Alms implementation of TCP (i): basic operations

The next several operations follow a similar pattern, catching any *IOError* exception thrown by the underlying operation and throwing a *Socket* exception containing the socket and its capability instead. This pattern is abstracted into helper function *lift*, which takes three arguments: first, a thunk for the desired operation; second, the socket to include in an exception, should one be raised; and third, a data constructor for the dynamic capability in the correct state should the operation fail.

The next six functions, starting with *bind*, demonstrate a simple use of *lift*. Each passes *lift* a function that performs the underlying socket operation, the socket itself, and the data constructor for the current state of the socket. Function *bind*, if successful, takes a socket in the raw state and transitions it to the named state. If *bind* fails, the socket remains in the raw state, so the third argument to *lift* in the definition of *bind* is data constructor *Raw*. Thus, if *S.bind* fails, then *bind* raises a *Socket* exception containing the socket and a dynamicCap value with a raw capability. Functions *listen*, *accept*, *send*, and *recv* all follow the same pattern. Function *connect* is slightly more complicated, because the socket given to *connect* may be in either state raw or named; it determines which state the capability is in by pattern matching on the sum of capabilities given as its final parameter, and passes *lift* the appropriate constructor. The final function in the figure, *close*, does not use *lift*, but instead raises an exception with *None* on failure.

The implementation of module *SocketCap* continues in figure 4.5 with the error-handling functions whose interface appeared in figure 4.3. Function *isSame* checks whether two sockets are identical; if so, it returns a witness function, and if not it returns *None*. The four high-level error-handling functions share a similar structure, abstracted into helper function *catchBy*. Function *catchBy* adds a second argument to the state-specific functions to specify which state to catch, represented (as in *lift*) as a data constructor for type dynamicCap. It invokes the body computation with an exception handler catching *Socket* exceptions. It then checks whether the caught dynamic capability is in the expected state and whether the caught socket matches the expected socket. If so, then it invokes the handler, but otherwise it re-raises the exception. The four state-specific error-handling functions are defined

```
let isSame sock sock' =
  if sock == sock'
    then Some (λ _ → ())
    else None

let catchBy body state (sock', handler) =
  try body ()
  with Socket ((Some (sock, dyncap), msg) as se) →
    if dyncap == state () && sock == sock'
      then handler ()
      else raise (Socket se)

let catchRaw body       = catchBy body Raw
let catchNamed body     = catchBy body Named
let catchListening body = catchBy body Listening
let catchReady body     = catchBy body Ready
```

Figure 4.5: Alms implementation of TCP (ii): error handling

simply as *catchBy* instantiated with the correct state constructor for each.

In general, implementing a typestate interface requires a trusted kernel that is not checked to obey the desired abstraction. Sometimes, it is possible to reduce the size of the trusted code by providing the right operations to untrusted code. For example, note that making *catchBy* available to clients of the sockets library would not be safe, as it trusts the caller that the given data constructor matches the type of the given exception handler. Thus, the implementation of figure 4.5 relies on the type of each of the four state-specific functions in the interface matching the data constructors passed by each in the implementation. There is an alternate design in which *catchBy* and the four state-specific functions are kept outside the trusted kernel, as they are expressible using *isSame*. This implementation of the error-handling functions, which is more complicated than the trusted version, appears in figure 4.6.

4.1.1 A Socket Example

```
let catchBy body
        (prj : ∀'a. 'a dynamicCap →
                 'a dynamicCap + ('a@'c -A-> 'a dynamicCap) × 'a@'c)
        (sock', handler) =
  try body ()
  with Socket (Some (sock, dyncap), msg) →
    match prj dyncap with
    | Left dyncap → raise (Socket (Some (sock, dyncap), msg))
    | Right (uncap, cap) →
        match isSame sock sock' with
        | None → raise (Socket (Some (sock, uncap cap), msg))
        | Some witness → handler (witness cap)

let catchRaw body =
  catchBy body (function Raw cap → Right (Raw, cap)
                       | dyncap  → Left dyncap)
let catchNamed body =
  catchBy body (function Named cap → Right (Named, cap)
                       | dyncap    → Left dyncap)
let catchListening body =
  catchBy body (function Listening cap → Right (Listening, cap)
                       | dyncap        → Left dyncap)
let catchReady body =
  catchBy body (function Ready cap → Right (Ready, cap)
                       | dyncap    → Left dyncap)
```

Figure 4.6: Alternate, untrusted implementation of error handling

```
module EchoServer = struct
  open SocketCap

  let bufSize = 1024

  let rec clientLoop sock ↯cap =
    let str = recv sock bufSize cap in
      send sock str ▷ cap;
      clientLoop sock cap

  let rec acceptLoop sock ↯cap =
    let (clientsock, clientcap) = accept sock cap in
      putStrLn "Opened␣connection";
      Thread.fork (λ _ →
        catchReady
          (λ _ → clientLoop clientsock clientcap)
          (clientsock, λ clientcap →
            close clientsock clientcap;
            putStrLn "Closed␣connection"));
      acceptLoop sock cap

  let serve port =
    let (sock, ↯cap) = socket () in
      bind sock port ▷ cap;
      listen sock ▷ cap;
      acceptLoop sock cap
end

let main = function
  | [port] → EchoServer.serve (int_of_string port)
  | _ → failwith "Usage:␣echoServer.alms␣PORT"

in main (getArgs ())
```

Figure 4.7: An echo server using *SocketCap*

Figure 4.7 contains an example program that uses the *SocketCap* structure to implement a simple echo server. The echo server accepts incoming connections, and for each client echoes back whatever data the client sends until the client disconnects. The program is structured as two loops—one for accepting incoming connections and one for handling each client—and some initialization code.

Function *clientLoop* defines the loop for interacting with each client, which repeatedly reads a string from the socket and writes the string back. The function binds the capability using the implicit threading syntax (§3.2.1); it does not need to explicitly thread the capability because that is handled by the implicit threading transformation.

Function *acceptLoop* defines the outer loop that waits for incoming connections and calls *clientLoop* in a new thread for each; this function too binds *cap* as an implicitly threaded variable. After a new connection is accepted, a new thread is started to service the client. The client loop is run in the context of an exception handler that catches *Socket* exceptions for socket *clientsock* in the ready state. When the client disconnects, then either *send* or *recv* (in the client loop) raises a *Socket* exception containing a socket and capability in state ready. The exception handler catches this, uses the capability to close the socket, and prints a message that the connection was closed.

The initial setup for the server happens in function *serve*, which creates a socket (binding the capability to an implicitly threaded variable, as usual), then binds the socket to a port and begins queueing incoming connections, calling *acceptLoop* to accept clients.

Finally, function *main* parses the command line arguments. Given a single argument, it starts the server on that port; otherwise, it prints an error message and exits.

4.2 Regions

As a high-level language with affine rather than linear types, Alms assumes that the run-time system uses tracing garbage collection to reclaim memory. However, regardless of run-time support for manual memory management,

```
module type SIMPLE_REGION = sig
  type 'r region
  type 'r regcap : A
  type ('r, 'a) ref

  val create : unit → ∃'r. 'r region × 'r regcap
  val delete : 'r region → 'r regcap → unit

  val new   : 'a → 'r region → 'r regcap → ('r, 'a) ref × 'r regcap
  val swap  : ('r, 'a) ref → 'a → 'r regcap → 'a × 'r regcap
  val write : ('r, 'a) ref → 'a → 'r regcap → 'r regcap
  val read  : ('r, 'a) ref → 'r regcap → 'a × 'r regcap
end
```

Figure 4.8: Simple, Vault-style regions

Alms's type system can express both simple and more advanced interfaces for regions. Throughout this section, we assume that we are allocating only mutable references in regions, rather than allocating arbitrary heap values in regions in the style of Walker et al. (2000).

Vault-style regions. A simple interface to regions similar those provided in Vault (see figures 2.6 and 2.7 on page 27) appears in figure 4.8. In this interface, regions are unlimited values and capabilities to access regions are affine. Because operations to access or free a region require an affine capability, this interface ensures that a region cannot be accessed after it is freed.

Regions are identified by type variables, which correspond to keys in Vault. That is, a region value of type `'r region` corresponds to a region value in Vault of type `tracked('r) region`; the capability `'r regcap` corresponds to having `'r` in the held key set. The type `('r, 'a) ref` of a value of type `'a` allocated in region `'r` corresponds to the Vault guarded type `'r:'a`.

Temporary aliasing. It is not surprising that Alms can support Vault-style regions, since they are a simple application of typestate. However, Alms can support more flexible kinds of regions as well. In §2.3.2, I discussed how Walker et al. (2000) use uniqueness subtyping and bounded quantification to

```
module type FRAC_REGION = sig
  type 'r region
  type ('r, 'c) regcap : A

  type 1
  type 2
  type 'n/'d

  val create : unit → ∃'r. 'r region × ('r, 1) regcap
  val delete : 'r region → ('r, 1) regcap → unit

  val split  : ('r, 'c) regcap → ('r, 'c/2) regcap × ('r, 'c/2) regcap
  val join   : ('r, 'c/2) regcap × ('r, 'c/2) regcap → ('r, 'c) regcap

  type ('r, 'a) ref

  val new    : 'a → 'r region → ('r, 'c) regcap → ('r, 'a) ref × ('r, 'c) regcap
  val swap   : ('r, 'a) ref → 'a → ('r, 'c) regcap → 'a × ('r, 'c) regcap
  val write  : ('r, 'a) ref → 'a → ('r, 'c) regcap → ('r, 'c) regcap
  val read   : ('r, 'a) ref → ('r, 'c) regcap → 'a × ('r, 'c) regcap
end
```

Figure 4.9: Regions with fractional capabilities

support temporarily aliasing regions. For example, here is the type of a CL function that takes a capability granting shared access to two regions and passes the same capability to its continuation:

$$\forall \alpha_1{:}\mathrm{RGN}, \alpha_2{:}\mathrm{RGN}.\, \forall \beta \leq {}^{\times}\mathsf{cap}\,\alpha_1 \otimes {}^{\times}\mathsf{cap}\,\alpha_2.(\beta, \ldots, (\beta, \ldots) \to 0) \to 0.$$

This type allows passing a unique capability to a single region, temporarily aliasing it as α_1 and α_2, such that the continuation then recovers the original, unique capability.

While Alms does not support uniqueness subtyping and bounded quantification, the same concept can be expressed in Alms using fractional capabilities (as in figure 3.11 on p. 51). The idea is that the type of a region capability includes a fraction that specifies "how much" of the capability is held. The full capability grants exclusive access and all that entails, whereas a partial capability grants only shared access. A signature for fractional regional

capabilities appears in figure 4.9. Region capabilities now have a second type parameter, which represents the fraction of the capability using types 1 and 2 and binary type constructor $\cdot/\cdot$. Function *create* returns a full capability (fraction 1), and *delete* likewise requires a full capability in order to delete the region. Functions *split* and *join* are used to manage capabilities by splitting a capability of fraction $'c$ into two of fraction $'c/2$ each and rejoining a pair of the latter back into the former. Operations on references are polymorphic in the fraction of the capability that they require and return, because none of the reference operations requires exclusive access.

Using this interface, the example from Walker et al. (2000) may be written in Alms as a function that takes two potentially-shared region capabilities, having this type:

$$\cdots \to (\text{'}r_1, \text{'}c_1)\ \mathsf{regcap} \times (\text{'}r_2, \text{'}c_2)\ \mathsf{regcap} \to \cdots \times (\text{'}r_1, \text{'}c_1)\ \mathsf{regcap} \times (\text{'}r_2, \text{'}c_2)\ \mathsf{regcap}$$

To call such a function by aliasing a single region capability, the caller must split the capability, and the caller can recover the original capability by joining the capabilities in the result. This split-join pattern is easily abstracted into a function that performs the splitting and joining:

```
let withSplit f cap =
  let (result, cap1, cap2) = f (split cap) in
    (result, join (cap1, cap2))
```

Adoption and focus. In the region examples presented thus far, the types of values in references are tracked by the reference type itself, which means that regions contain values of multiple types. Another approach is *homogeneous regions*, where the region capability tracks the type of references to the region, and thus all values in a region have the same type (Walker et al. 2000; Charguéraud and Pottier 2008). A signature for simple homogeneous regions appears in figure 4.10.

Fähndrich and DeLine (2002) propose a pair of operations, *adoption* and *focus*, for dealing with unlimited objects that point to linear objects; Pottier (2007) reinterprets these operations as acting on homogeneous regions. Pottier distinguishes singleton regions, which contain a single reference, from group

```
module type HOMOGENEOUS_REGION = sig
  type 'r region
  type ('r, 'a) regcap : A
  type 'r ref

  val create : unit → ∃'r. 'r region × ('r, 'a) regcap
  val delete : 'r region → ('r, 'a) regcap → unit

  val new   : 'a → 'r region → ('r, 'a) regcap → 'r ref × ('r, 'a) regcap
  val swap  : 'r ref → 'a → ('r, 'a) regcap → 'a × ('r, 'a) regcap
  val write : 'r ref → 'a → ('r, 'a) regcap → ('r, 'a) regcap
  val read  : ('r, 'a) ref → ('r, 'a) regcap → 'a × ('r, 'a) regcap
end
```

Figure 4.10: Simple homogeneous regions

regions, which may contain a number of references. References in both group and singleton regions support the usual reference operations such as reading, writing, and swapping. References in singleton regions also support *strong updates,* whereby writing a reference changes its type. Group regions do not support strong updates because the same region capability tracks the type of all references to the region. A signature for singleton and group regions appears in figure 4.11.

The adoption and focus operations support managing singleton and group regions. Each new reference is allocated in a new, singleton region. A group region may permanently *adopt* a singleton region, at which point the capability for the singleton region is lost and the type of the reference is updated to indicate that it now belongs to the group region. Group regions do not support strong updates, but *focus* allows carving out one reference from a group region and temporarily assigning it to a singleton region. Since the type of the reference may then be changed by a strong update, the old group capability must be temporarily disabled to avoid accessing the reference at its old type. So, *focus* does not return the old group capability, but instead returns a defocus function, which converts the new singleton region capability back into the old group region capability.

```
module type FOCUS_REGION = sig
  type ('r, 'k, 'a) rgn : A
  type singleton
  type group
  type ('r, 'a) rgn1 = ('r, singleton, 'a) rgn
  type ('r, 'a) rgnG = ('r, group, 'a) rgn
  type 'r ref

  val new       : 'a → ∃'r. 'r ref × ('r, 'a) rgn1
  val newGroup  : unit → ∃'r. ('r, 'a) rgnG
  val delete    : ('r, 'k, 'a) rgn → unit

  val read      : 'r ref → ('r, 'k, 'a) rgn → 'a × ('r, 'k, 'a) rgn
  val write     : 'r ref → 'a → ('r, 'k, 'a) rgn → ('r, 'k, 'a) rgn
  val swap      : 'r ref → 'a → ('r, 'k, 'a) rgn → 'a × ('r, 'k, 'a) rgn
  val strongWrite : 'r ref → 'b → ('r, 'a) rgn1 → ('r, 'b) rgn1
  val strongSwap : 'r ref → 'b → ('r, 'a) rgn1 → 'a × ('r, 'b) rgn1

  type ('r, 'a, 's) defocus = ('s, 'a) rgn1 → ('r, 'a) rgnG

  val adopt     : 'r ref → ('r, 'a) rgn1 → ('s, 'a) rgnG →
                    's ref × ('s, 'a) rgnG
  val focus     : 'r ref → ('r, 'a) rgnG →
                    ∃'s. 's ref × ('s, 'a) rgn1 × ('r, 'a, 's) defocus
end
```

Figure 4.11: Homogeneous regions with adoption and focus

None of the region interfaces in this section are especially useful in a high-level language like Alms, though they do demonstrate the flexibility of Alms's type system and its potential applicability for managing memory in a lower-level language. However, regions are not limited to managing memory. I return to regions with adoption and focus at the end of this chapter (§4.3.4), where I discuss an application of regions to session types.

```
type 1
type +'a ; +'s rec 's
type ↑ -'a
type ↓ +'a
type +'s ⊕ +'t
type +'s & +'t
```

Figure 4.12: Binary session types

4.3 Session Types

In this section, I describe three designs for session types in Alms. The first provides binary, anonymous branches—in other words, every protocol branch goes two ways, with labels amounting to *left* and *right*. The second take on session types provides k-way, labeled branches, though unlike the first it relies on explicit protocol declarations. Finally, the third subsection combines session types with regions. In each case, the value that tracks the state of the session, whether the channel itself or a capability, is affine, which ensures that as each step in a protocol is taken, the ability to erroneously repeat that step is lost.

4.3.1 Anonymous, Binary Session Types

Type definitions for binary session types appear in figure 4.12. The session type for a finished session is 1. Sequencing is written using the binary type constructor $\cdot;\cdot$. Both parameters are covariant, and the annotation "**rec** 's" specifies that equirecursion is allowed through the second parameter, in order to represent recursive session types.[2] The next two type constructors, $\uparrow\cdot$ and $\downarrow\cdot$, are used in the first parameter of $\cdot;\cdot$ to indicate sending and receiving, respectively. (There is no need for special cases for sending and receiving sessions, since session-typed channels are ordinary values.) Thus, $\uparrow$int; $\downarrow$bool; 1 is the protocol to send an int and then receive a bool.

[2]Alms has equirecursive types, but in order to prevent the spurious typings that equirecursion introduces, it infers equirecursive types only when cycles are guarded by a type constructor that is declared to allow recursion.

```
type 1          dual = 1
   | (↑'a; 's) dual = ↓'a; 's dual
   | (↓'a; 's) dual = ↑'a; 's dual
   | ('s & 't) dual = 's dual ⊕ 't dual
   | ('s ⊕ 't) dual = 's dual & 't dual
```

Figure 4.13: Duality for binary session types

Binary choice is written using type constructors $\cdot \underline{\oplus} \cdot$ for internal choice and $\cdot \underline{\&} \cdot$ for external choice. For example, μ'a. ↑int; 'a $\underline{\&}$ 1 is the type of a protocol to send any number of ints, where the length of the sequence is determined by the receiver.

Figure 4.13 defines a *type function* dual, which computes session type duality. Alms supports the definition of type functions defined by recursion and pattern matching on types. Type patterns are built out of type variables and type constructors (which cannot be type functions or type synonyms themselves to ensure coherence). Type function applications are evaluated by checking the patterns in each clause in order until one matches, then substituting in the right-hand side of the matching clause. Type function application is non-strict, so that recursive functions such as dual may be applied to recursive types. (Evaluation is forced when it is necessary to check subtyping between applications of different type constructors.) Type function dual matches five potential patterns, in each case transforming a session type to its dual (defined in figure 2.12 on p. 32).

The remainder of the interface for binary session types appears in figure 4.14. Type 's rendezvous is an unlimited rendezvous object for protocol 's, which servers and clients can synchronize on to start sessions; function *newRendezvous* creates a new rendezvous object for any protocol. To start a new session, a client uses function *request* on a rendezvous object for some protocol 's to get an affine channel of type 's channel. The client blocks until paired with a server calling function *accept* on the same rendezvous object, at which point *accept* returns a channel for the dual protocol 's dual.

The last five functions in signature BINARY_SESSION implement the

```
module type BINARY_SESSION = sig
  ⟨session types in figure 4.12⟩
  ⟨duality in figure 4.13⟩

  type 's rendezvous
  type +'s channel : A

  val newRendezvous : unit → 's rendezvous

  val request       : 's rendezvous → 's channel
  val accept        : 's rendezvous → 's dual channel

  val send          : 'a → (↑'a; 's) channel → 's channel
  val recv          : (↓'a; 's) channel → 'a × 's channel
  val sel1          : ('s ⊕ 't) channel → 's channel
  val sel2          : ('s ⊕ 't) channel → 't channel
  val follow        : ('s & 't) channel → 's channel + 't channel
end
```

Figure 4.14: Interface for binary session types

operations for communicating in a session. Functions *send* and *recv* send and receive values, in each case taking a channel where the protocol says to send (or receive) a value, and returning a channel for the remaining protocol. Internal choice is done with functions sel_1 and sel_2, which given a channel whose protocol is an internal choice, return a channel for the left or right protocol of the choice, respectively. Function *follow* implements external choice by turning a channel with an external choice protocol into either a channel for the left protocol or a channel for the right protocol, depending on the selection made on the other end of the channel.

Implementation. My implementation of the binary session types interface uses monomorphic, synchronous channels (figure 4.15). Because these channels allow communicating values of only one type, implementing session-typed channels on top of monomorphic channels requires some way to send multiple types. One possibility is channel-passing style, whereby each message includes a fresh channel to use for the next message, but this is complicated and likely inefficient. Instead, I use a coercion, *Unsafe.unsafeCoerce*, which has

```
module Channel : sig
  type 'a channel

  val new  : unit → 'a channel
  val send : 'a channel → 'a → unit
  val recv : 'a channel → 'a
end
```

Figure 4.15: Monomorphic, synchronous channels

```
module BinarySession : BINARY_SESSION = struct
  ⟨session types in figure 4.12⟩
  ⟨duality in figure 4.13⟩

  module C = Channel

  type 's channel = bool C.channel
  type 's rendezvous = 's channel C.channel

  let newRendezvous = C.new
  let request = C.recv
  let accept rv = let c = C.new () in C.send rv c; c

  let send c a = C.send (Unsafe.unsafeCoerce c) a; c
  let recv c = (C.recv (Unsafe.unsafeCoerce c), c)
  let sel1 c = C.send c true; c
  let sel2 c = C.send c false; c
  let follow c = if C.recv c then Left c else Right c
end
```

Figure 4.16: Implementation of binary session types

type $\forall\,\text{'}a\,\text{'}b.\ \text{'}a \to \text{'}b$. Using an unsafe coercion can easily violate type safety; thus, safety relies on the correctness of both the signature and the structure implementing it. (I showed elsewhere (Pucella and Tov 2008) how session types implemented using unsafe underlying channels protected by the right signature can be proved type safe.)

Signature BINARY_SESSION is implemented in figure 4.16. Session channels are represented by type bool *C*.channel; the type parameter is not im-

portant, since we are using unsafe coercions, but bool simplifies the code somewhat. A rendezvous object for protocol *'s* is represented by a monomorphic channel for sending channels. Creating a new rendezvous object merely requires creating a new channel, and requesting a session is done by receiving the channel for the session over a rendezvous channel. Thus, at the other end of a request, *accept* needs to create a new session channel, send it over the rendezvous channel, and return it.

Session communication is straightforward except for the coercions. To send or receive on a session-typed channel, we coerce the channel to the right type for the value to be sent or received, send or receive the value, and return the channel. For choice, the internal choice side sends a boolean to the external choice side, where true indicates the left branch and false indicates the right. Thus, sel_1 and sel_2 send true and false, respectively. Function *follow* receives a bool is returns the channel injected to the left or right depending on the branch chosen by the other side.

4.3.2 Labeled, k-ary Session Types

The changes from binary to k-ary session types are small (figure 4.17). The complete session 1, sequencing, sending, and receiving, are written as before; only the treatment of branching changes. Whereas before we declared branching as part of the syntax of session types, instead branching will be done using Alms's built-in algebraic data types. Internal choice is now written with type $\sim\underline{\oplus}$`c, which is merely a type synonym for sending a value of type `c and completing the session; external choice, written $\sim\underline{\&}$`c is done by receiving the value of type `c. Type variable `c will be instantiated with an algebraic data type representing the available choices in some branch. The branching operations thus work as follows. Function *choose* takes a constructor to wrap a channel for some protocol *'s* in the algebraic data type `c, which *choose* then sends over the $\sim\underline{\oplus}$`c channel; *choose* returns the other end of the sent channel as an *'s* dual channel. On the other end, *follow* receives the value of the algebraic data type, which the receiver must pattern match to get a channel with the protocol of the sender's choice. Recursive sessions are written using algebraic

```
module type SESSION_TYPE = sig
  type 1
  type +'a ; +'s
  type ↑ −'a
  type ↓ +'a

  type 1          dual = 1
     | (↑'a; 's) dual = ↓'a; 's dual
     | (↓'a; 's) dual = ↑'a; 's dual

  type 's rendezvous
  type +'s channel : A

  val newRendezvous : unit → 's rendezvous

  val request : 's rendezvous → 's channel
  val accept  : 's rendezvous → 's dual channel

  val send    : 'a → (↑'a; 's) channel → unit × 's channel
  val recv    : (↓'a; 's) channel → 'a × 's channel

  type ~⊕ 'c = ↑'c; 1
  type ~& 'c = ↓'c; 1

  val choose  : ('s channel -A-> 'c) → ~⊕'c channel → unit × 's dual channel
  val follow  : ~&'c channel → unit × 'c
end
```

Figure 4.17: Interface for k-ary session types

data types as well.

If this approach to branching seems mystifyingly general, consider the following example. The ATM–bank protocol (originally from figure 2.9 on p. 30) appears in figure 4.18, and a sample client using the protocol appears in figure 4.19. The protocol starts with an internal choice, whereby the ATM sends the bank a value of type choose_trans, which is a three-way labeled sum over channels for the protocols for each choice. From the ATM's perspective, the sent channels are for a protocol dual to its own, since they will be used by the bank side of the transaction. Within the *Withdraw* branch, there is an external choice, which indicates where the bank selects *Success* or *Failure* by

```
type atm_prot      = ~⊕choose_trans
 and choose_trans = Deposit of (↑int; ↓int; 1) dual channel
                  | Balance of (↓int; 1) dual channel
                  | Withdraw of (↓int; ~&status) dual channel
 and status       = Success of (↓int; 1) channel
                  | Failure of (↓string; 1) channel
```

Figure 4.18: ATM–bank protocol in k-ary session types

```
let performWithdrawal amt (rv : atm_prot rendezvous) =
  let ↯c = request rv in
    choose Withdraw c;
    send amt c;
    follow c;
    match c with
    | Success c → recv c
    | Failure c → failwith (recv c)
```

Figure 4.19: Client for ATM–bank protocol

sending a channel for the remaining protocol in one of those data constructors. The client implements the ATM side of the protocol for making a withdrawal in function *performWithdrawal*. The arguments to *performWithdrawal* are the amount to withdraw and a rendezvous object for connecting to the bank. The client connects, chooses the *Withdraw* branch, sends the amount, and then must follow whichever branch of type *status* chosen by the other side. On *Success*, the client receives and returns the transaction number; on *Failure*, the client receives the reason for failure and throws it as an exception.

Implementation. An implementation of labeled, k-ary session types appears in figure 4.20. The implementation is very similar to the implementation of binary session types in figure 4.16, except that there is less code for implementing choice because choice is handled by clients to the library using algebraic data types and pattern matching. Function *choose* simply takes the

```
module SessionType : SESSION_TYPE = struct
  ⟨session types and duality in figure 4.17⟩
  module C = Channel
  type 's channel       = bool C.channel
  type 's rendezvous    = 's channel C.channel
  let newRendezvous = C.new
  let request       = C.recv
  let accept rv     =
    let c = C.new () in
      C.send rv c;
      c
  let send a c      = (C.send (Unsafe.unsafeCoerce c) a, c)
  let recv c        = (C.recv (Unsafe.unsafeCoerce c), c)
  let choose ctor c = send (ctor c) c
  let follow c      = ((), fst (recv c))
end
```

Figure 4.20: Implementation of k-ary session types

constructor for the chosen branch and the channel, and sends the constructor applied to the channel over the channel; *follow* receives the chosen branch over the channel and returns it in place of the channel for pattern matching by the client code.

4.3.3 Example: Parallel Polygon Clipping

As an example, I give an implementation of Sutherland and Hodgman's (1974) reentrant polygon clipping algorithm using session types. A sample input and output for the algorithm appears in figure 4.21. (The actual algorithm works with planes and points in $\mathbb{R}^3$, but the figure is limited to two dimensions for readability.) The input to the algorithm is a set of planes, which represent a convex, plane-faced volume, and a sequence of points, which represent a polygon. In the diagram, the input appears on the left: there are two planes (perpendicular to the page), representing the volume to their right; and four

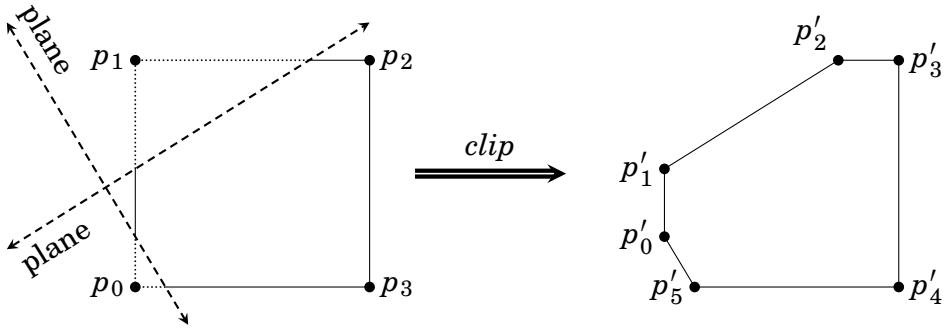


Figure 4.21: Example of polygon clipping

points, representing a rectangle. The output is the polygon clipped within the plane-faced volume, as appears on the right in figure 4.21.

Sutherland and Hodgman's algorithm clips a polygon against a single plane independently of the other planes, which means that it is easily parallelized with a separate process for each plane. The original sequence of points is sent to a process for the first plane, which clips against that plane and sends the modified sequence of points to a process for the next plane, and so on, where the process for the final plane outputs the final, clipped polygon.

In the example of figure 4.21, the first plane runs from the lower left to the upper right, labeled in the lower left, and the second plane runs from the upper left to lower center. The original polygon is represented by the sequence of points p_0, p_1, p_2, p_3. When sent to the process for the first plane, the result is the sequence $p_0, p'_1, p'_2, p_2, p_3$. That sequence is sent to the clipper process for the second plane, which yields $p'_0, p'_1, p'_2, p'_3, p'_4, p'_5$. Each clipper process holds onto at most two points at a time and begins producing points right away, which means that the algorithm is highly parallel.

A geometry library. A signature for a simple 3-D geometry library appears in figure 4.22. The library defines types representing points and planes. The record $\{ x, y, z \}$ represents the Cartesian coordinate (x, y, z); the record $\{ a, b, c, d \}$ represents the either the plane $\{(x, y, z) \mid ax + by + cz + d = 0\}$ or the volume $\{(x, y, z) \mid ax + by + cz + d > 0\}$, depending on context. A line segment is represented as a pair of points.

```
module Geometry : sig
  type point    = { x, y, z : float }
  type plane    = { a, b, c, d : float }
  type segment = point × point

  val string_of_point  : point → string
  val string_of_plane : plane → string

  val point_of_string  : string → point
  val plane_of_string : string → plane

  val is_point_above_plane
                       : point → plane → bool
  val intersect        : segment → plane → point option
end
```

Figure 4.22: Interface for a simple 3-D geometry library

The *Geometry* structure provides functions for printing and parsing points and planes, and two additional functions for doing geometry. The first function, *is_point_above_plane*, takes a point and a plane representing the volume "above" the plane, and returns whether the point is in the volume. Function *intersect* takes a plane and a line segment and returns the point where the segment intersects the plane, or *None* if they do not intersect.

The protocol. The protocol used by the polygon clipping program is defined in figure 4.23. The first protocol, ‘a stream, specifies receiving a sequence of values of type ‘a, where the length of the sequence is determined by the sender. This is the protocol used to communicate sequences of points between clipping processes. The second protocol, main_prot, is used to communicate the whole input to the algorithm, as it specifies receiving a sequence of planes (whose length is determined by the sender) followed by a sequence of points (protocol point stream).

Parsing and printing. The remaining code in figure 4.23 implements the program's communication with the outside world.

The result of the algorithm is printed to the standard output by function

```
open Geometry

(* Protocol for receiving a sequence of 'a: *)
type 'a stream   = ~&('a step)
 and 'a step     = Done of 1 channel
                 | Next of (↓'a; 'a stream) channel

(* The main protocol: receive a sequence of planes followed by a
 * sequence of points: *)
type main_prot = ~&choice
 and choice     = Planes of (↓plane; main_prot) channel
                | Points of point stream channel

(* Print sequence of points read from ic: *)
let rec printer ↯(ic : point streamchannel) =
  follow ic;
  match ic with
  | Done ic → ()
  | Next ic → putStrLn (string_of_point (recv ic)); printer ic

(* Parse the input planes and points; send to a channel: *)
let parser ↯(oc : main_protdual channel) =
  let rec plane_loop () = match getLine () with
        | "" →  choose Points oc;
                point_loop ()
        | s  →  choose Planes oc;
                send (plane_of_string s) oc;
                plane_loop ()
    and point_loop () = match getLine () with
        | "" →  choose Done oc
        | s  →  choose Next oc;
                send (point_of_string s) oc;
                point_loop ()
  in plane_loop ()
```

Figure 4.23: Implementation of polygon clipping (part 1 of 3)

(continued in figure 4.24)

printer, which receives a sequence of points from a point stream channel and prints them, in their external form, as they are read.

The input to the algorithm is read from the standard input and parsed by function *parser*, which sends the sequence of planes and sequence of points over a main_prot dual channel. The external form of the input is a sequence of planes on separate lines, then a blank line, and then a sequence of points, one per line, and finally terminating with a blank line. The parser reads this external input format and sends values of types plane and point over the channel provided as its argument.

Clipping. The polygon clipping program is structured as a chain of stream transducers, with a parser as the source, a clipper process for each plane, and a printer as the sink. Function *clipper* (figure 4.24) implements a clipper process for a single plane. The function takes a plane representing the space to clip within and two channels: a point stream channel for input and a point stream dual channel for output.

The definition of *clipper* begins with several helper functions. Helper function *put* informs the next clipper (or the printer) on the output channel that it will be sending another point by selecting branch *Next*, and then sends the point over the same channel. It does not take channel *oc* as an explicit argument, because the outer binding of *oc* as an implicitly-threaded variable is in scope; thus, the implicit threading transformation takes care of threading the output channel through *put*. The next helper, *putCross*, takes a line segment and sends the intersection of the segment with the clipping plane, if they intersect. The final helper, *putVisible*, takes a point and sends it only if the point is within the space delimited by the clipping plane.

The main body of *clipper* works as follows. First, if the input sequence of points is empty then so is the output; otherwise, it reads the first point p_0, which it needs to retain, as the first vertex of a closed polygon is also the last. It then loops through the remaining input points, outputting those points or their intersections with the plane as appropriate. It finishes by connecting the last point received to p_0.

(continued from figure 4.23)

```
(* Read a polygon from ic and send it, clipped by plane, to oc: *)
let clipper plane
        ↯(ic : point stream channel, oc : point stream dual channel) =
  (* Send point p on channel oc: *)
  let put p = choose Next oc; send p oc in
  (* If segment intersects plane then send the point of intersection: *)
  let putCross segment =
    match intersect segment plane with
    | Some p → put p
    | None → () in
  (* Send point p if in semi-space "above" the plane: *)
  let putVisible p =
    if is_point_above_plane p plane
      then put p
      else ()
  in
  follow ic;
  match ic with
  | Done ic → choose Done oc
  | Next ic →
    let p0 = recv ic in
    let rec loop p =
      putVisible p;
      follow ic;
      match ic with
      | Done ic → putCross (p, p0); choose Done oc
      | Next ic → let p' = recv ic in putCross p p'; loop p'
    in
      loop p0
```

Figure 4.24: Implementation of polygon clipping (part 2 of 3)

(continued in figure 4.25)

(continued from figure 4.24)

```
let main () =
  let rec get_planes (acc : plane list) ↯(ic : main_prot channel) =
        follow ic;
        match ic with
        | Points ic → rev acc
        | Planes ic → get_planes (recv ic :: acc) ic in
  let rec connect plane (ic : point stream channel) =
        let outrv = newRendezvous () in
          Thread.fork (λ _ → clipper plane (ic, accept outrv); ());
          request outrv in
  let rv = newRendezvous () in
  Thread.fork (λ _ → parser (accept rv); ());
  let (planes, ic) = get_planes [] (request rv) in
  let ic = foldl connect ic planes
  in
    printer ic
```

Figure 4.25: Implementation of polygon clipping (part 3 of 3)

Putting it all together. Function *main* (figure 4.25) sets up the chain of stream transducers, creating the necessary threads and connecting them with session-typed channels. The definition of *main* begins with two helper functions. Helper function *get_planes* accumulates a list of planes read from a main_prot channel, returning the list of planes and the channel, which has by that point become a point stream channel. Helper function *connect* takes a plane and a point stream channel, and creates a clipper for that plane in a new thread, connecting the given channel as the clipper's input and creating a new channel for its output; it returns the other end of the new channel, from which the next transducer in the chain can read the sequence of points as clipped by the new clipper.

Then the whole chain of stream transducers is set up as follows. First, a rendezvous object is created for communicating with the parser, which is started in a new thread. The parser accepts a connection on the new rendezvous object, and the main thread requests a connection on the same rendezvous and passes the resulting channel to *get_planes* in order to obtain

the list of planes that represent the clipping space. Then, folding *connect* over the list of planes starts a clipper for each plane, yielding a channel connected to the final clipper, which is passed to the printer for printing the result.

4.3.4 Session Type Regions

As I observed at the end of §4.2, while Alms is capable of expressing interfaces for regions, region-based memory management is not especially useful in Alms. Charguéraud and Pottier (2008) suggest understanding regions not as sets of memory locations but as sets of *values*. From that perspective, regions become more widely applicable. In this section, I show how regions may be combined with session types, in order to allow a single region capability to track the state of any number of channels.

A signature for binary session types with regions appears in figure 4.26. The main idea is to separate channels from their sessions, tracking the current session with a capability. Thus, the type parameter on the channel type *'r* channel is not a session, but a region variable tying the channel to a capability of either type *'r@'s* or *'r@@'s*, where *'s* is the protocol for channel *'r*. Regions here are homogeneous, and as in the previous example with adoption and focus, there are singleton regions with capabilities of type *'r@'s*, and group regions with capabilities of type *'r@@'s*. Channels are initially created paired with a singleton region, and all channel operations require a singleton region for the channel as well.

Group regions of type *'r@@'s*, created by function *newGroup*, allow associating multiple channels having the same session with a single region, which means that they are all tracked by the same capability. There are no direct channel operations for channels belonging to group regions, so to use such a channel requires first focusing on the channel to get a temporary singleton region, then performing the desired channel operations, and finally defocusing to regain access to the group region that owns the channel.

Implementation. The implementation of signature SESSION_REGION is straightforward (figure 4.27), and mostly similar to the implementation of BINARY_SESSION. Channels are represented by bool *C*.channel, which means

```
module type SESSION_REGION = sig
  type 1
  type +'a ; +'s rec 's
  type ↑−'a
  type ↓+'a
  type +'s ⊕ +'t
  type +'s & +'t

  type 1          dual = 1
     | (↑'a; 's) dual = ↓'a; 's dual
     | (↓'a; 's) dual = ↑'a; 's dual
     | ('s & 't) dual = 's dual ⊕ 't dual
     | ('s ⊕ 't) dual = 's dual & 't dual

  type 's rendezvous
  type 'r channel
  type 'r @ 's   : A
  type 'r @@ 's : A

  val newRendezvous : unit → 's rendezvous

  val request       : 's rendezvous → ∃'r. 'r channel × 'r@'s
  val accept        : 's rendezvous → ∃'r. 'r channel × 'r@'s dual

  val send          : 'a → 'r channel → 'r@(↑'a; 's) → 'r@'s
  val recv          : 'r channel → 'r@(↓'a; 's) → 'a × 'r@'s

  val sel1          : 'r channel → 'r@('s ⊕ 't) → 'r@'s
  val sel2          : 'r channel → 'r@('s ⊕ 't) → 'r@'t
  val follow        : 'r channel → 'r@('s & 't) → 'r@'s + 'r@'t

  type ('r1, 's, 'r2) defocus = 'r2@'s → 'r1@@'s

  val newGroup      : unit → ∃'r. 'r@@'s
  val adopt         : 'r1 channel → 'r1@'s → 'r2@@'s →
                        'r2 channel × 'r2@@'s
  val focus         : 'r1 channel → 'r1@@'s →
                        ∃'r2. 'r2 channel × ('r1, 's, 'r2) defocus × 'r2@'s
end
```

Figure 4.26: Interface to session types with regions

```
module SessionRegion : SESSION_REGION = struct
  ⟨session types in figure 4.12⟩
  ⟨duality in figure 4.13⟩

  module C = Channel

  type 's rendezvous = bool C.channel C.channel
  type 'r channel = bool C.channel
  type 'r @ 's = unit
  type 'r @@ 's = unit

  let newRendezvous = C.new
  let request rv : ∃'r. 'r channel × unit = (C.recv rv, ())
  let accept rv : ∃'r. 'r channel × unit =
    let c = C.new () in
      C.send rv c;
      (c, ())

  let send a c _ = C.send (Unsafe.unsafeCoerce c) a
  let recv c _ = (C.recv (Unsafe.unsafeCoerce c), ())

  let sel1 c _ = C.send c true
  let sel2 c _ = C.send c false
  let follow c _ = if C.recv c then Left () else Right ()

  type ('r1, 's, 'r2) defocus = 'r2@'s → 'r1@@'s

  let newGroup _ : ∃'r. 'r@@'s = ()

  let adopt c _ _ = (c, ())
  let focus c _ : ∃'r2. 'r2 channel × (unit → unit) × unit = (c, id, ())
end
```

Figure 4.27: Implementation of session types with regions

that communication requires an unsafe coercion. Region capabilities are represented by type unit. All channel operations now take an additional parameter representing the region capability, and return value () as appropriate to represent the region capability in the result.

The new operations are straightforward as well. Function *newGroup* returns the unit value to represent a new group region capability. Function *adopt* takes a channel and ignores its two capability arguments, and returns a pair of the channel (which, by the signature, has a different region tag on its type) and () for the capability. Finally, *focus* take a channel and capability, and returns a triple of the channel, the identity function to represent the defocus operation, and () as the new region capability.

A session type region example. A program using session type regions appears in figures 4.28 and 4.29.

The server side. Figure 4.28 begins with two protocols for a server that broadcasts messages of some type *'a* to multiple subscribers. The server receives two kinds of commands over a control channel with protocol *'a* control. In the left branch of the protocol, the server receives a value of type *'a* to broadcast to all of its subscribers. In the left branch, the server receives a new subscriber, represented as a pair of a channel and a singleton capability. The subscriber communicates with the server by protocol *'a* subscription: first the subscriber receives a welcome message from the server, and then the subscriber can choose the left branch to disconnect, or the right branch to subscribe to the sequence of broadcast messages.

Because the server needs to maintain an arbitrary number of subscribers, all of which speak the same protocol, it uses a group region to track all subscribers with a single capability. Thus, to communicate with a subscriber, the server must focus that subscriber's channel. Before the definition of the server itself, two functions that help the server communicate with subscribers are defined. Function *deliver* takes a message, the channel for a subscriber, and the group region capability for the channel. It focuses the channel, sends the message, and then defocuses in order to return the group capability. Function *broadcast* takes a message, a list of subscriber channels, and the group region

```
open SessionRegion

type 'a subscription = ↓string; (1 ⊕ μ's. ↓'a; 's)

type 'a control =
    ↓'a; 'a control
  &
    ↓(∃'r. 'r channel × 'r@'a subscription dual); 'a control

let deliver msg subChan ↯(subRgn : 'rs@@(μ's. ↑'a; 's)) =
  let (subChan, defocus) = focus subChan subRgn in
    send msg subChan ▷ subRgn;
    defocus subRgn

let broadcast msg subChans (subRgn : 'rs@@(μ's. ↑'a; 's)) =
  foldl (deliver msg) subRgn subChans

let server (ctlChan, ctlRgn) =
  let rec loop subChans
            ↯(ctlRgn : 'r@'a control, subsRgn : 'q@@(μ's. ↑'a; 's)) =
    follow ctlChan ▷ ctlRgn;
    let subChans = match ctlRgn with
        | Left ctlRgn →
            let msg = recv ctlChan ctlRgn in
            broadcast msg subChans ▷ subsRgn;
            subChans
        | Right ctlRgn → fst $
            let (subChan, ↯subRgn) = recv ctlChan ctlRgn in
            send "Hello." subChan ▷ subRgn;
            follow subChan ▷ subRgn;
            match subRgn with
            | Left _ → subChans
            | Right subRgn →
              (adopt subChan ◁ subRgn) subsRgn :: subChans
    in loop subChans (ctlRgn, subsRgn)
  in loop [] (ctlRgn, newGroup ())
```

Figure 4.28: Broadcasting using session type regions (part 1 of 2)

(continued in figure 4.29)

capability for all the subscribers; it uses *deliver* to send the message to every subscriber in the list.

The server is defined by function *server*, which takes as arguments a control channel and the singleton region capability for the channel. Before entering its loop for handling commands, the server creates a new group region for subscribers, and starts the loop with an empty list of subscriber channels. The server then reads commands from the control channel. Given a message command (left), the server broadcasts the message to all the subscribers using function *broadcast*. Given a subscriber command (right), the server first announces itself to the potential subscriber, and then follows whether the potential subscriber elects to subscribe. If not, then the server is done with that command; if so, then it adds the subscriber to the subscriber group region and adds the subscriber channel to the list of subscribers.

The client side. Function *subscriber*, in figure 4.29, implements a subscriber. It takes two parameters: *each* is a function to apply to each received message, and *rv* is a rendezvous value on which to request a subscription. After requesting a subscription, the client prints the welcome message read from the channel, elects to subscribe by choosing the right branch of the '*a* subscribtion protocol, and enters a loop to read messages from the channel, applying *each* to each.

Function *client* implements a line-based user interface with three possible commands. If the user types +, then the client creates a new subscriber in a new thread and connects it to the server. If the user types q, then the client exits the program. Anything else typed by the user is sent to the server as a message to broadcast to its subscribers.

Finally, function *main* ties it all together. It starts the server in a new thread and connects it to the client started in the main thread.

4.4 Discussion

In this chapter, I have demonstrated approaches to typestate, regions, and session types in Alms. Several notable benefits of Alms should stand out.

(continued from figure 4.28)

```
let subscriber each (rv : 'a subscription rendezvous) =
  let (subChan, ↯subRgn) = request rv in
  let rec loop () =
    each (recv subChan subRgn);
    loop () in
  putStrLn (recv subChan subRgn);
  sel2 subChan ▷ subRgn;
  loop ()

let rec client ctlChan ↯(ctlRgn : 'r@string control dual) =
  match getLine () with
  | "+" →
      let rv = newRendezvous () in
      Thread.fork (λ _ → subscriber putStrLn rv);
      sel2 ctlChan ▷ ctlRgn;
      send (accept rv) ctlChan ▷ ctlRgn;
      client ctlChan ctlRgn
  | "q" →
      exit 0
  | msg →
      sel1 ctlChan ▷ ctlRgn;
      send msg ctlChan ▷ ctlRgn;
      client ctlChan ctlRgn

let main () =
  let rv = newRendezvous () in
  Thread.fork (λ _ → server (request rv));
  uncurry client (accept rv)
```

Figure 4.29: Broadcasting using session type regions (part 2 of 2)

First, the interfaces are reasonably concise and the implementations simple. None of the examples in this chapter relies on an elaborate encoding. They do, however, rely on the module system's hiding of implementation details, which forces clients to abide by usage restrictions; for example, clients are prevented from duplicating affine capabilities, even though these capabilities are usually represented by an unlimited nonce value.

Second, the sample programs written using the libraries defined in this chapter are not significantly more complicated than the same programs would be if written in languages with special-purpose stateful type systems; for example, the echo server written with the typestate Berkeley sockets library looks similar to how it would in Vault.

Third, the examples in this chapter demonstrate how affine types combine with other language features to support defining and using abstractions in new ways. For example, the use of exceptions in the Berkeley sockets library simplifies client code that uses the interface because it eliminates the need for explicit error checks; yet it maintains the possibility of handling errors if desired. As another example, algebraic data types combine with session types to support k-ary, labeled session types.

Fourth, and perhaps most importantly, this chapter shows off Alms's flexibility. Because none of these stateful type systems is built in to Alms, each can be defined in a variety of different ways to suit different needs. Furthermore, they can be combined in ways that were not anticipated when the language was designed, such as in the final example of session types with regions.

CHAPTER 5

A Model of Alms

ALMS HAS SEVERAL new type system features to make programming with affine types practical, such as dereliction subtyping, automatic selection of function usage qualifiers, dependent kinds, and abstract affine types. To validate the soundness of this approach, I have constructed a core model for Alms and established two results demonstrating its beneficial properties: a standard, syntactic type soundness theorem and a theorem that Alms always selects the best usage qualifier for function types. In this chapter, I describe the syntax, semantics, and theory of the model.

The model, ${}^{a}\lambda_{ms}$, is based on System $F^{\omega}_{<:}$, the higher-order polymorphic λ calculus with subtyping (Pierce 2002), with affine types and several other changes to make it a more faithful model of Alms. Like $F^{\omega}_{<:}$, it features subtyping and type operators. Subtyping models Alms's dereliction subtyping, whereby a function whose qualifier is unlimited may be used where a one-shot function is expected; indeed, all of ${}^{a}\lambda_{ms}$'s subtyping relation arises from function qualifier subtyping. Like $F^{\omega}_{<:}$, ${}^{a}\lambda_{ms}$ has type operators, but unlike $F^{\omega}_{<:}$, type operators are restricted to first-order kinds, which means that the arguments to type operators cannot be type operators themselves. The kind system of ${}^{a}\lambda_{ms}$ is enriched with dependent kinds, which succinctly describe the relationship between the qualifier of a type operator and the qualifiers of its parameters. Additionally, kinds in ${}^{a}\lambda_{ms}$ carry variance information (Steffen 1997), which allows abstract type constructors to specify how their results vary in relation to their parameters.

The calculus ${}^{a}\lambda_{ms}$ also includes more types and expressions than a minimal presentation of $F^{\omega}_{<:}$. Because I am interested in practical issues, I believe it is important for the model to include products,[1] sums, mutable references, and non-termination.

While ${}^{a}\lambda_{ms}$ does not model Alms's module system directly, the combination of universal types, type operators, and subkinding is sufficient to support abstract affine types in the style of Alms. These features make ${}^{a}\lambda_{ms}$ a suitable target for the first-order fragment of Rossberg et al.'s (2010) "F-ing modules" translation, which compiles a language with modules to System F_{ω}.

5.1 Syntax and Semantics of ${}^{a}\lambda_{ms}$

The syntax of ${}^{a}\lambda_{ms}$ begins in figure 5.1. Expressions (e) include the usual expressions from System F (variables, abstractions, applications, type abstractions, and type applications), several forms for data construction and elimination (nil, sum injections, sum elimination, pairs, and pair elimination), recursion ($\mathsf{fix}\, e$), and several operations on reference cells (allocation, linear swap, and deallocation). Location names (ℓ) appear at run time but are not present in source terms.

Types (τ, figure 5.2) include type variables, type-level abstraction and application, universal quantification, function types, and type constructor constants for sums, products, unit, and references. As in Alms, the function arrow carries a *usage qualifier* (ξ), which specifies whether the function is unlimited or one-shot.

The two constant usage qualifiers (q), U for unlimited and A for affine, are the bottom and top of the two-element lattice in figure 5.3. To see why the constant usage qualifiers are insufficient for describing the most general usage constraints of functions, consider the K combinator $\Lambda\alpha{:}\langle\alpha\rangle.\, \Lambda\beta{:}\langle\beta\rangle.\, \lambda x{:}\alpha.\, \lambda y{:}\beta.\, x$

[1] In linear logic terms, my calculus supplies multiplicative conjunction ($\otimes$) and additive disjunction ($\oplus$) directly. Additive conjunction ($\&$) is easily encoded by

$$\tau_1 \,\&\, \tau_2 \triangleq \forall\alpha{:}\mathsf{A}.(\tau_1 \xrightarrow{\mathsf{A}}_{\circ} \alpha) \oplus (\tau_2 \xrightarrow{\mathsf{A}}_{\circ} \alpha) \xrightarrow{\xi_1 \sqcup \xi_2}_{\circ} \alpha$$

$$[v_1, v_2] \triangleq \Lambda\alpha{:}\mathsf{A}.\, \lambda x{:}(\tau_1 \xrightarrow{\mathsf{A}}_{\circ} \alpha) \oplus (\tau_2 \xrightarrow{\mathsf{A}}_{\circ} \alpha).\, \mathsf{case}\, x\, \mathsf{of}\, \mathsf{inl}\, y_1 \to y_1\, v_1; \mathsf{inr}\, y_2 \to y_2\, v_2,$$

where ξ_1 and ξ_2 are the kinds of τ_1 and τ_2.

α, β	$\in$	$TVar$	type variables
x, y	$\in$	Var	variables
ℓ	$\in$	Loc	locations (run time only)
e	::=		expressions
	$\mid$	x	variable
	$\mid$	$\lambda x{:}\tau.\, e$	abstraction
	$\mid$	$e_1\, e_2$	application
	$\mid$	$\Lambda\alpha{:}\kappa.\, v$	type abstraction
	$\mid$	$e\,\tau$	type application
	$\mid$	$\mathsf{fix}\, e$	recursion
	$\mid$	$\langle\rangle$	nil value
	$\mid$	$\mathsf{inl}\, e$	left sum injection
	$\mid$	$\mathsf{inr}\, e$	right sum injection
	$\mid$	$\mathsf{case}\, e\, \mathsf{of}\, \mathsf{inl}\, x \to e_1; \mathsf{inr}\, y \to e_2$	sum elimination
	$\mid$	$\langle e_1, e_2\rangle$	pair construction
	$\mid$	$\mathsf{let}\, \langle x, y\rangle = e_1\, \mathsf{in}\, e_2$	pair elimination
	$\mid$	$\mathsf{new}\, e$	reference allocation
	$\mid$	$\mathsf{swap}\, e_1\, e_2$	reference access
	$\mid$	$\mathsf{delete}\, e$	reference deallocation
	$\mid$	ℓ	location (run time only)

Figure 5.1: Syntax (i): expressions

τ, σ	::=		types
	$\mid$	α	type variable
	$\mid$	$\lambda\alpha.\, \tau$	type-level abstraction
	$\mid$	$\tau_1\, \tau_2$	type-level application
	$\mid$	$\forall\alpha{:}\kappa.\, \tau$	universal type
	$\mid$	$\tau_1 \overset{\xi}{\multimap} \tau_2$	function type
	$\mid$	χ	type constructor constant
χ	::=		type constructor constants
	$\mid$	$(\oplus)$	additive disjunction
	$\mid$	$(\otimes)$	multiplicative conjunction
	$\mid$	1	unit of $(\otimes)$
	$\mid$	ref	mutable reference

Figure 5.2: Syntax (ii): types

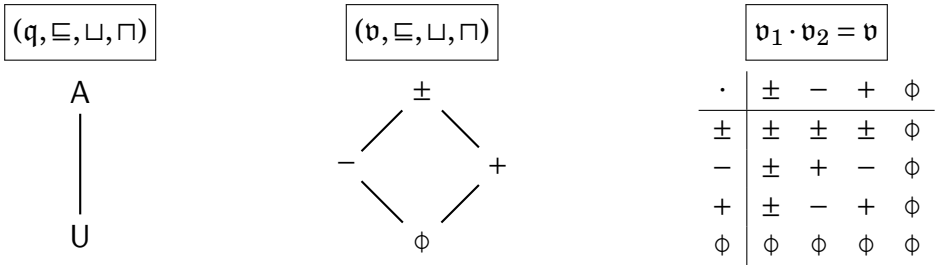


Figure 5.3: Syntax (iii): Qualifier constants, variances, and variance composition

ξ	::=		usage qualifier expressions
	|	$\mathfrak{q}$	qualifier constant
	|	$\langle\alpha\rangle$	qualifier of type variable
	|	$\xi_1 \sqcup \xi_2$	least upper bound
	|	$\xi_1 \sqcap \xi_2$	greatest lower bound
κ	::=		kinds
	|	ξ	kind of proper type
	|	$\Pi\alpha^{\mathfrak{v}}.\kappa$	kind of type operator

Figure 5.4: Syntax (iv): kinds

instantiated with two types and partially applied to a value: $\mathrm{K}\tau_1\tau_2 v$. Whether it is safe to duplicate this expression depends on whether it is safe to duplicate v, and this is reflected in the instantiation of α by τ_1. To express this relationship, I introduce usage qualifier expressions (ξ), which form a bounded, distributive lattice over type variables—qualifier $\langle\alpha\rangle$ is the kind of type α—with U and A as bottom and top. We can thus give K type $\forall\alpha{:}\langle\alpha\rangle.\,\forall\beta{:}\langle\beta\rangle.\,\alpha \xrightarrow{\mathsf{U}}\circ\, \beta \xrightarrow{\langle\alpha\rangle}\circ\, \alpha$.

Qualifier expressions (figure 5.4) are the base kinds of ${}^{a}\lambda_{ms}$—that is, the kinds that classify proper types that may in turn classify values. To classify type operators, kinds (κ) also include dependent product kinds, written $\Pi\alpha^{\mathfrak{v}}.\kappa$. This is the kind of a type operator that, when applied to a type with kind ξ, gives a type with kind $\{\xi/\alpha\}\kappa$. For example, the kind of the Alms list type constructor is $\Pi\alpha^{+}.\langle\alpha\rangle$, which means that a list has the same usage qualifier

as its elements.

The superscript + in kind $\Pi\alpha^{+}.\langle\alpha\rangle$ means that the list type constructor is covariant (or monotone): if τ_1 is a subtype of τ_2 then a list of τ_1 is a subtype of a list of τ_2. Variances ($\mathfrak{v}$) form a four-point lattice (figure 5.3). A type operator may also be contravariant ($-$), where the result varies inversely with the argument; omnivariant (ϕ), where argument may vary freely without affecting the result; or invariant ($\pm$), where the argument may not vary at all without producing a subtyping-unrelated result. We define a composition operation ($\cdot$) on variances, where $\mathfrak{v}_1 \cdot \mathfrak{v}_2$ is the variance of the composition of type operators having variances $\mathfrak{v}_1$ and $\mathfrak{v}_2$.

The kinds of the type constructors for sums and references may aid understanding. The sum type constructor ($\oplus$) has kind $\Pi\alpha^{+}.\Pi\beta^{+}.\langle\alpha\rangle \sqcup \langle\beta\rangle$. This means that the kind of a sum type is at least as restrictive as the kinds of its disjuncts. It is covariant in both arguments, which means that $\tau_1 \oplus \tau_2$ is a subtype of $\tau_1' \oplus \tau_2'$ if τ_1 is a subtype of τ_1' and τ_2 is a subtype of τ_2'. The reference type constructor, on the other hand, has kind $\Pi\alpha^{\pm}.\mathsf{A}$. This means that reference cells are always affine and that their types do not support subtyping in either direction.

5.1.1 Operational Semantics

The operational semantics of ${}^{a}\lambda_{ms}$ is mostly a standard call-by-value reduction semantics. The reduction rules are given in figure 5.5. The reduction relation ($\longmapsto$) relates configurations, (s, e), comprising a store and an expression. A store maps locations (ℓ) to values (v). Stores are taken to be unordered and do not repeat location names.

The rules for reference operations are worth noting. In store s, $\mathsf{new}\, v$ chooses a fresh location ℓ, adding v to the store at location ℓ and reducing to the reference ℓ. The operation $\mathsf{swap}\, \ell\, v_2$ requires that the store have location ℓ holding some value v_1. It swaps v_2 for v_1 in the store, returning a pair of a reference to ℓ and value v_1. Finally, $\mathsf{delete}\, \ell$ also requires that the store contain ℓ, which it then removes from the store. This means that freeing a location can result in a dangling pointer, which would cause subsequent attempts to

$$v ::= \lambda x{:}\tau.\, e \mid \Lambda\alpha{:}\kappa.\, v \mid \mathsf{fix}\, v \mid \langle\rangle \mid \mathsf{inl}\, v \mid \mathsf{inr}\, v \mid \langle v_1, v_2\rangle \mid \ell \qquad \text{values}$$

$$s ::= \{\} \mid \{\ell \mapsto v\} \mid s_1 \uplus s_2 \qquad \text{stores}$$

$$\boxed{(s,e) \longmapsto (s',e')} \qquad \textit{(reduction)}$$

$$(\beta_v) \qquad (s, (\lambda x{:}\tau.\, e)\, v) \longmapsto (s, \{v/x\}\, e)$$

$$(\mathrm{B}_\tau) \qquad (s, (\Lambda\alpha{:}\kappa.\, v)\, \tau) \longmapsto (s, \{\tau/\alpha\}\, v)$$

$$(\textsc{Fix}_v) \qquad (s, \mathsf{fix}\, v_1\, v_2) \longmapsto (s, v_1\, (\mathsf{fix}\, v_1)\, v_2)$$

$$(\textsc{CaseL}) \qquad (s, \mathsf{case}\ \mathsf{inl}\, v\ \mathsf{of}\ \mathsf{inl}\, x \to e_1; \mathsf{inr}\, y \to e_2) \longmapsto (s, \{v/x\}\, e_1)$$

$$(\textsc{CaseR}) \qquad (s, \mathsf{case}\ \mathsf{inr}\, v\ \mathsf{of}\ \mathsf{inl}\, x \to e_1; \mathsf{inr}\, y \to e_2) \longmapsto (s, \{v/y\}\, e_2)$$

$$(\textsc{LetPair}) \qquad (s, \mathsf{let}\, \langle x, y\rangle = \langle v_1, v_2\rangle\ \mathsf{in}\ e) \longmapsto (s, \{v_1/x\}\{v_2/y\}\, e)$$

$$(\textsc{New}) \qquad (s, \mathsf{new}\, v) \longmapsto (s \uplus \{\ell \mapsto v\}, \ell)$$

$$(\textsc{Swap}) \qquad (s \uplus \{\ell \mapsto v_1\}, \mathsf{swap}\, \ell\, v_2) \longmapsto (s \uplus \{\ell \mapsto v_2\}, \langle \ell, v_1\rangle)$$

$$(\textsc{Delete}) \qquad (s \uplus \{\ell \mapsto v\}, \mathsf{delete}\, \ell) \longmapsto (s, \langle\rangle)$$

$$(\textsc{Cxt}) \qquad \frac{(s,e) \longmapsto (s',e')}{(s, E[e]) \longmapsto (s', E[e'])}$$

$$\begin{aligned} \text{where } E ::= {} & [\,] \mid E\, e \mid v\, E \mid E\, \tau \mid \mathsf{fix}\, E \\ & \mid \mathsf{inl}\, E \mid \mathsf{inr}\, E \mid \mathsf{case}\, E\ \mathsf{of}\ \mathsf{inl}\, x \to e_1; \mathsf{inr}\, y \to e_2 \\ & \mid \langle E, e\rangle \mid \langle v, E\rangle \mid \mathsf{let}\, \langle x, y\rangle = E\ \mathsf{in}\ e \\ & \mid \mathsf{new}\, E \mid \mathsf{swap}\, E\, e \mid \mathsf{swap}\, v\, E \mid \mathsf{delete}\, E \end{aligned}$$

Figure 5.5: Operational semantics

access that location to get stuck. Because ${}^a\lambda_{ms}$ references are affine, the type system prevents this.

5.1.2 Static Semantics

The type system of ${}^a\lambda_{ms}$ involves a large number of judgments, which I summarize in figure 5.6.

Typing contexts (Γ or Σ; figure 5.7) associate type variables with their kinds, variables with their types, and locations with the types of their contents. By convention, I use Γ for typing contexts that include neither affine variables

$\Gamma \vdash \kappa$	kind κ is well formed	(fig. 5.8)
$\vdash \Gamma$	kinds in context Γ are well formed	(fig. 5.8)
$\Gamma \models \xi_1 \sqsubseteq \xi_2$	qualifier ξ_1 subsumes ξ_2	(def. 5.2)
$\Gamma \vdash \kappa_1 <: \kappa_2$	kind κ_1 subsumes κ_2	(fig. 5.8)
$\Gamma \vdash \alpha \in \tau \updownarrow \mathfrak{v}$	type τ varies $\mathfrak{v}$-ly when α increases	(fig. 5.9)
$\Gamma \vdash \tau : \kappa$	type τ has kind κ	(fig. 5.9)
$\tau_1 \equiv \tau_2$	types τ_1 and τ_2 are β-equivalent	(fig. 5.10)
$\Gamma \vdash \tau_1 <:^{\mathfrak{v}} \tau_2$	type τ_1 is $\mathfrak{v}$-related to type τ_2	(fig. 5.10)
$\Gamma \vdash \Sigma \preceq \xi$	context Σ is bounded by qualifier ξ	(fig. 5.11)
$\vdash \Gamma; \Sigma$	dual contexts $\Gamma; \Sigma$ are well formed	(fig. 5.11)
$\vdash (\Gamma_0; \Sigma_0), \Sigma' \rightsquigarrow \Gamma; \Sigma$	extending $\Gamma_0; \Sigma_0$ with Σ' gives $\Gamma; \Sigma$	(fig. 5.11)
$\Gamma; \Sigma \rhd e : \tau$	expression e has type τ	(fig. 5.12)
$\Sigma_1 \rhd s : \Sigma_2$	store s has type Σ_2	(fig. 5.13)
$\rhd (s, e) : \tau$	configuration (s, e) has type τ	(fig. 5.13)

Figure 5.6: Type system judgments

$\Gamma,\ \Sigma$	::=		typing contexts
	|	$\bullet$	empty
	|	Γ_1, Γ_2	concatenation
	|	$\alpha{:}\kappa$	kind of type variable
	|	$x{:}\tau$	type of variable
	|	$\ell{:}\tau$	type of location

Figure 5.7: Syntax of typing contexts

nor locations, and I use Σ for typing contexts that may include locations and affine (or indeterminate) variables.

Kind judgments. Judgments on kinds appear in figure 5.8. The first judgment, $\Gamma \vdash \kappa$, determines whether a kind κ is well formed in context Γ. A base kind (*i.e.,* a usage qualifier expression) is well formed if it is closed. A dependent product kind $\Pi\alpha^{\mathfrak{v}}.\kappa$ is well formed if whenever the bound type variable α is free in κ—that is, when the kind is truly dependent—then variance $\mathfrak{v}$ must be $+$ or $\pm$. This rules out incoherent kinds such as $\Pi\alpha^{-}.\langle\alpha\rangle$ that classify no useful type operator but whose presence breaks the kinding relation's monotonicity property (see lemma 5.4).

$\boxed{\Gamma \vdash \kappa}$ *(kind well-formedness)*

$$\textsc{OK-Qual}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathfrak{q}} \qquad \textsc{OK-Var}\ \frac{\alpha{:}\xi \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash \langle \alpha \rangle} \qquad \textsc{OK-Join}\ \frac{\Gamma \vdash \xi_1 \qquad \Gamma \vdash \xi_2}{\Gamma \vdash \xi_1 \sqcup \xi_2} \qquad \textsc{OK-Meet}\ \frac{\Gamma \vdash \xi_1 \qquad \Gamma \vdash \xi_2}{\Gamma \vdash \xi_1 \sqcap \xi_2}$$

$$\textsc{OK-Oper}\ \frac{\text{if } \alpha \in \mathrm{FTV}(\kappa) \text{ then } + \sqsubseteq \mathfrak{v} \qquad \Gamma, \alpha{:}\langle \alpha \rangle \vdash \kappa}{\Gamma \vdash \Pi \alpha^{\mathfrak{v}}.\kappa}$$

$\boxed{\vdash \Gamma}$ *(kind context well-formedness)*

$$\textsc{Wf-Nil}\ \frac{}{\vdash \bullet} \qquad \textsc{Wf-ConsA}\ \frac{\vdash \Gamma \qquad \Gamma \vdash \kappa}{\vdash \Gamma, \alpha{:}\kappa} \qquad \textsc{Wf-ConsARec}\ \frac{\vdash \Gamma}{\vdash \Gamma, \alpha{:}\langle \alpha \rangle} \qquad \textsc{Wf-ConsX}\ \frac{\vdash \Gamma}{\vdash \Gamma, x{:}\tau} \qquad \textsc{Wf-ConsL}\ \frac{\vdash \Gamma}{\vdash \Gamma, \ell{:}\tau}$$

$\boxed{\Gamma \vdash \kappa_1 <: \kappa_2}$ *(subkinding)*

$$\textsc{KSub-Qual}\ \frac{\Gamma \models \xi_1 \sqsubseteq \xi_2 \qquad \Gamma \vdash \xi_1 \qquad \Gamma \vdash \xi_2}{\Gamma \vdash \xi_1 <: \xi_2} \qquad \textsc{KSub-Oper}\ \frac{\mathfrak{v}_1 \sqsubseteq \mathfrak{v}_2 \qquad \Gamma, \alpha{:}\langle \alpha \rangle \vdash \kappa_1 <: \kappa_2}{\Gamma \vdash \Pi \alpha^{\mathfrak{v}_1}.\kappa_1 <: \Pi \alpha^{\mathfrak{v}_2}.\kappa_2}$$

Figure 5.8: Statics (i): kinds

The second judgment is well-formedness for kinding contexts, which ensures that all kinds in a context Γ are well-formed. Note that there are two rules for type variable bindings: rule WF-CONSA handles general bindings of the form $\alpha{:}\kappa$ for well formed kinds κ that do not refer to α, whereas rule WF-CONSAREC handles recursive bindings of the form $\alpha{:}\langle \alpha \rangle$. This judgment does not check the well-formedness of types in the context.

The third judgment is subkinding: $\Gamma \vdash \kappa_1 <: \kappa_2$. As we will see, if a type has kind κ_1, then it may be used where κ_1 or any greater kind is expected. For dependent product kinds the subkinding order is merely the product order on the variance and the result kind, but for base kinds the relation relies on an interpretation of qualifier expressions, which clarifies the meaning of free type

variables in base kinds.

We interpret qualifier expressions via a valuation $\mathcal{V}$, which is a map from type variables to qualifier constants. We extend $\mathcal{V}$'s domain to qualifier expressions:

$$\mathcal{V}(\mathsf{q}) = \mathsf{q} \qquad \mathcal{V}(\xi_1 \sqcup \xi_2) = \mathcal{V}(\xi_1) \sqcup \mathcal{V}(\xi_2)$$
$$\mathcal{V}(\langle\alpha\rangle) = \mathcal{V}(\alpha) \qquad \mathcal{V}(\xi_1 \sqcap \xi_2) = \mathcal{V}(\xi_1) \sqcap \mathcal{V}(\xi_2)$$

We need to interpret qualifier expressions under a typing context:

DEFINITION 5.1 (Consistent valuations)**.**

*A valuation $\mathcal{V}$ is **consistent with** a typing context Γ if for all $\alpha{:}\xi \in \Gamma$, $\mathcal{V}(\alpha) \sqsubseteq \mathcal{V}(\xi)$.*

Thus, a valuation is consistent with a context if it corresponds to a potential instantiation of the type variables, given that context.

DEFINITION 5.2 (Qualifier subsumption)**.**

*A qualifier expression ξ_1 **subsumes** ξ_2 in Γ, written $\Gamma \models \xi_1 \sqsubseteq \xi_2$, if for all valuations $\mathcal{V}$ consistent with Γ, $\mathcal{V}(\xi_1) \sqsubseteq \mathcal{V}(\xi_2)$.*

In other words, in all possible instantiations of the type variables in Γ, qualifier ξ_1 being A implies that ξ_2 is A.

Kinding and variance. The two judgments in figure 5.9, for computing variances and giving kinds to types, are defined by mutual induction. It should be clear on inspection that the definitions are well founded. Judgment $\Gamma \vdash \alpha \in \tau \updownarrow \mathfrak{v}$ means that type variable α appears in type τ at variance $\mathfrak{v}$, or in other words, that type operator $\lambda\alpha.\tau$ has variance $\mathfrak{v}$. Rules V-VARPRE, V-VARABS, and V-CON say that type variables appear positively with respect to themselves and omnivariantly with respect to types in which they are not free. Rule V-ABS says that a type variable appears in a type operator $\lambda\beta.\tau$ at the same variance that it appears in the body τ. The remaining three rules are more involved:

$\boxed{\Gamma \vdash \alpha \in \tau \updownarrow \mathfrak{v}}$ *(variance of type variables with respect to types)*

$$\textsc{V-VarPre}\ \frac{\Gamma \vdash \alpha : \kappa}{\Gamma \vdash \alpha \in \alpha \updownarrow +} \qquad \textsc{V-VarAbs}\ \frac{\Gamma \vdash \beta : \kappa \qquad \alpha \neq \beta}{\Gamma \vdash \alpha \in \beta \updownarrow \phi} \qquad \textsc{V-Con}\ \frac{\vdash \Gamma}{\Gamma \vdash \alpha \in \chi \updownarrow \phi} \qquad \textsc{V-Abs}\ \frac{\Gamma, \beta{:}\langle\beta\rangle \vdash \alpha \in \tau \updownarrow \mathfrak{v}}{\Gamma \vdash \alpha \in \lambda\beta.\tau \updownarrow \mathfrak{v}}$$

$$\textsc{V-App}\ \frac{\Gamma \vdash \alpha \in \tau_1 \updownarrow \mathfrak{v}_1 \qquad \Gamma \vdash \alpha \in \tau_2 \updownarrow \mathfrak{v}_2 \qquad \Gamma \vdash \tau_1 : \Pi\beta^{\mathfrak{v}_3}.\kappa_3}{\Gamma \vdash \alpha \in \tau_1\,\tau_2 \updownarrow \mathfrak{v}_1 \sqcup (\mathfrak{v}_2 \cdot \mathfrak{v}_3)}$$

$$\textsc{V-All}\ \frac{\Gamma, \beta{:}\kappa \vdash \alpha \in \tau \updownarrow \mathfrak{v}_1 \qquad \mathfrak{v}_2 = \begin{cases} \pm & \alpha \in \mathrm{FTV}(\kappa) \\ \phi & \alpha \notin \mathrm{FTV}(\kappa) \end{cases}}{\Gamma \vdash \alpha \in \forall\beta{:}\kappa.\tau \updownarrow \mathfrak{v}_1 \sqcup \mathfrak{v}_2} \qquad \textsc{V-Arr}\ \frac{\Gamma \vdash \alpha \in \tau_1 \updownarrow \mathfrak{v}_1 \qquad \Gamma \vdash \alpha \in \tau_2 \updownarrow \mathfrak{v}_2 \qquad \mathfrak{v}_3 = \begin{cases} + & \alpha \in \mathrm{FTV}(\xi) \\ \phi & \alpha \notin \mathrm{FTV}(\xi) \end{cases}}{\Gamma \vdash \alpha \in \tau_1 \xrightarrow{\xi}_{\circ} \tau_2 \updownarrow -\mathfrak{v}_1 \sqcup \mathfrak{v}_2 \sqcup \mathfrak{v}_3}$$

$\boxed{\Gamma \vdash \tau : \kappa}$ *(kinding of types)*

$$\textsc{K-Var}\ \frac{\alpha{:}\kappa \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \textsc{K-Abs}\ \frac{\Gamma, \alpha{:}\langle\alpha\rangle \vdash \tau : \kappa}{\Gamma, \alpha{:}\langle\alpha\rangle \vdash \alpha \in \tau \updownarrow \mathfrak{v}}{\Gamma \vdash \lambda\alpha.\tau : \Pi\alpha^{\mathfrak{v}}.\kappa} \qquad \textsc{K-App}\ \frac{\Gamma \vdash \tau_1 : \Pi\alpha^{\mathfrak{v}}.\kappa \qquad \Gamma \vdash \tau_2 : \xi}{\Gamma \vdash \tau_1\,\tau_2 : \{\xi/\alpha\}\kappa} \qquad \textsc{K-All}\ \frac{\Gamma, \alpha{:}\kappa \vdash \tau : \xi}{\Gamma \vdash \forall\alpha{:}\kappa.\tau : \{\mathsf{A}/\alpha\}\xi}$$

$$\textsc{K-Arr}\ \frac{\Gamma \vdash \tau_1 : \xi_1 \qquad \Gamma \vdash \tau_2 : \xi_2 \qquad \Gamma \vdash \xi}{\Gamma \vdash \tau_1 \xrightarrow{\xi}_{\circ} \tau_2 : \xi} \qquad \textsc{K-Sum}\ \frac{\vdash \Gamma}{\Gamma \vdash (\oplus) : \Pi\alpha^{+}.\Pi\beta^{+}.\langle\alpha\rangle \sqcup \langle\beta\rangle}$$

$$\textsc{K-Prod}\ \frac{\vdash \Gamma}{\Gamma \vdash (\otimes) : \Pi\alpha^{+}.\Pi\beta^{+}.\langle\alpha\rangle \sqcup \langle\beta\rangle} \qquad \textsc{K-Unit}\ \frac{\vdash \Gamma}{\Gamma \vdash 1 : \mathsf{U}} \qquad \textsc{K-Ref}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{ref} : \Pi\alpha^{\pm}.\mathsf{A}}$$

Figure 5.9: Statics (ii): types

- By rule V-APP, the variance of a type variable in a type application comes from both the operator and the operand. The variance of α in $\tau_1\,\tau_2$ is at least the variance of α in τ_1 and at least the variance of α in τ_2 composed with the variance of operator τ_1. This makes sense: if τ is a contravariant type operator, then α appears negatively in $\tau\,\alpha$ but positively in $\tau\,(\tau\,\alpha)$.

- By rule V-ALL, the variance of α in $\forall\beta{:}\kappa.\,\tau$ is at least its variance in τ. However, if α appears in κ then it is invariant in $\forall\beta{:}\kappa.\,\tau$. This reflects the fact that universally quantified types are related only if their bounds (κ) match exactly, so changing a type variable that appears in κ produces an unrelated type. (This means that ${}^{a}\lambda_{ms}$ is based on the *kernel* variant of $\mathrm{F}^{\omega}_{<:}$ (Pierce 2002).)

- By rule V-ARR, the variance of α in a function type $\tau_1 \xrightarrow{\xi} \tau_2$ is at least its variance in the codomain τ_2 and at least the opposite (composition with $-$) of its variance in the domain τ_1. This reflects function argument contravariance. The variance of α is at least $+$ if it appears in the qualifier expression ξ.

The second judgment, $\Gamma \vdash \tau : \kappa$, assigns kinds to well-formed types. Rule K-VAR merely looks up the kind of a type variable in the context. Rules K-ABS and K-APP are the usual rules for dependent abstraction and application, with two small changes in rule K-ABS. First, it associates α with *itself* in the context, as $\alpha{:}\langle\alpha\rangle$, which ensures that occurrences of α in τ can be reflected in κ. Second, it appeals to the variance judgment to determine the variance of the type operator. Rule K-ALL assigns a universal type the same kind as its body, but with A replacing α. This is necessary because the resulting kind is outside the scope of α. Qualifier A is a safe bound for any instantiation of α, and no terms have types that lose precision by this choice. The kind of an arrow type, in rule K-ARR, is just the qualifier expression attached to the arrow. The remaining rules give kinds for type constructor constants, where $(\oplus)$ and ref are as discussed above and $(\otimes)$ has the same kind as $(\oplus)$.

Type equivalence and dereliction subtyping. The next two judgments in figure 5.10 are type β equivalence and subtyping. The subtyping relation

$\boxed{\tau_1 \equiv \tau_2}$ *(type equivalence)*

$$\textsc{E-Refl}\ \frac{}{\tau \equiv \tau} \qquad \textsc{E-Sym}\ \frac{\tau_1 \equiv \tau_2}{\tau_2 \equiv \tau_1} \qquad \textsc{E-Trans}\ \frac{\tau_1 \equiv \tau_2 \qquad \tau_2 \equiv \tau_3}{\tau_1 \equiv \tau_3} \qquad \textsc{E-Arr}\ \frac{\tau_{11} \equiv \tau_{21} \qquad \tau_{12} \equiv \tau_{22}}{\tau_{11} \xrightarrow{\xi}\circ\ \tau_{12} \equiv \tau_{21} \xrightarrow{\xi}\circ\ \tau_{22}}$$

$$\textsc{E-All}\ \frac{\tau_1 \equiv \tau_2}{\forall \alpha{:}\kappa.\, \tau_1 \equiv \forall \alpha{:}\kappa.\, \tau_2} \qquad \textsc{E-Abs}\ \frac{\tau_1 \equiv \tau_2}{\lambda \alpha.\, \tau_1 \equiv \lambda \alpha.\, \tau_2} \qquad \textsc{E-App}\ \frac{\tau_{11} \equiv \tau_{21} \qquad \tau_{12} \equiv \tau_{22}}{\tau_{11}\, \tau_{12} \equiv \tau_{21}\, \tau_{22}}$$

$$\textsc{E-Beta}\ \frac{}{(\lambda \alpha.\, \tau_1)\, \tau_2 \equiv \{\tau_2/\alpha\}\, \tau_1}$$

$\boxed{\Gamma \vdash \tau_1 <:^{\mathfrak{v}} \tau_2}$ *(subtyping)*

$$\textsc{TSub-Eq}\ \frac{\Gamma \vdash \tau_1 : \kappa \qquad \Gamma \vdash \tau_2 : \kappa \qquad \tau_1 \equiv \tau_2}{\Gamma \vdash \tau_1 <:^{\mathfrak{v}} \tau_2} \qquad \textsc{TSub-Omni}\ \frac{\Gamma \vdash \tau_1 : \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \tau_1 <:^{\Phi} \tau_2}$$

$$\textsc{TSub-Trans}\ \frac{\Gamma \vdash \tau_1 <:^{\mathfrak{v}} \tau_2 \qquad \Gamma \vdash \tau_2 <:^{\mathfrak{v}} \tau_3 \qquad \Gamma \vdash \tau_2 : \kappa}{\Gamma \vdash \tau_1 <:^{\mathfrak{v}} \tau_3} \qquad \textsc{TSub-Contra}\ \frac{\Gamma \vdash \tau_2 <:^{-\mathfrak{v}} \tau_1}{\Gamma \vdash \tau_1 <:^{\mathfrak{v}} \tau_2}$$

$$\textsc{TSub-Abs}\ \frac{\Gamma, \alpha{:}\langle\alpha\rangle \vdash \tau_1 <:^{\mathfrak{v}} \tau_2}{\Gamma \vdash \lambda \alpha.\, \tau_1 <:^{\mathfrak{v}} \lambda \alpha.\, \tau_2} \qquad \textsc{TSub-App}\ \frac{\Gamma \vdash \tau_{11} : \Pi \alpha^{\mathfrak{v}_1}.\, \kappa_1 \qquad \Gamma \vdash \tau_{21} : \Pi \alpha^{\mathfrak{v}_2}.\, \kappa_2 \qquad \Gamma \vdash \tau_{11} <:^{\mathfrak{v}} \tau_{21} \qquad \Gamma \vdash \tau_{12} <:^{\mathfrak{v} \cdot (\mathfrak{v}_1 \sqcup \mathfrak{v}_2)} \tau_{22}}{\Gamma \vdash \tau_{11}\, \tau_{12} <:^{\mathfrak{v}} \tau_{21}\, \tau_{22}}$$

$$\textsc{TSub-All}\ \frac{\Gamma, \alpha{:}\kappa \vdash \tau_1 <:^{\mathfrak{v}} \tau_2}{\Gamma \vdash \forall \alpha{:}\kappa.\, \tau_1 <:^{\mathfrak{v}} \forall \alpha{:}\kappa.\, \tau_2}$$

$$\textsc{TSub-Arr}\ \frac{\Gamma \vdash \tau_{11} <:^{-\mathfrak{v}} \tau_{21} \qquad \Gamma \vdash \tau_{12} <:^{\mathfrak{v}} \tau_{22} \qquad \Gamma \vdash \xi_1 <:^{\mathfrak{v}} \xi_2}{\Gamma \vdash \tau_{11} \xrightarrow{\xi_1}\circ\ \tau_{12} <:^{\mathfrak{v}} \tau_{21} \xrightarrow{\xi_2}\circ\ \tau_{22}}$$

Figure 5.10: Statics (iii): subtyping

is parametrized by a variance $\mathfrak{v}$, which gives the direction of the subtyping: $\Gamma \vdash \tau_1 <:^+ \tau_2$ is the usual direction, judging τ_1 a subtype of τ_2. In terms of subsumption, this means that values of type τ_1 may be used where values of type τ_2 are expected. The other variances are useful in defining the relation in the presence of $\mathfrak{v}$-variant type operators: $(<:^-)$ gives the inverse of the subtyping relation, $(<:^\pm)$ relates only equivalent types, and $(<:^\phi)$ relates all types. We can see how this works in rule TSUB-APP. To determine whether $\tau_{11}\,\tau_{12}$ is a subtype of $\tau_{21}\,\tau_{22}$, we take $\mathfrak{v}$ to be $+$, yielding

$$\frac{\Gamma \vdash \tau_{11} : \Pi\alpha^{\mathfrak{v}_1}.\kappa_1 \qquad \Gamma \vdash \tau_{21} : \Pi\alpha^{\mathfrak{v}_2}.\kappa_2 \qquad \Gamma \vdash \tau_{11} <:^+ \tau_{21} \qquad \Gamma \vdash \tau_{12} <:^{\mathfrak{v}_1 \sqcup \mathfrak{v}_2} \tau_{22}}{\Gamma \vdash \tau_{11}\,\tau_{12} <:^+ \tau_{21}\,\tau_{22}}.$$

This means that for the subtyping relation to hold:

- The operators must be related in the same direction, so that τ_{11} is a subtype of τ_{21}.
- The operands must be related in the direction given by the variances of the operators. For example, if both operators are covariant, then the operands must vary in the same direction, so that τ_{12} is a subtype of τ_{22}. If both operators are contravariant, then the operands must vary in the opposite direction. If the operators are invariant then the operands cannot vary at all, but if they are omnivariant then $\tau_{11}\,\tau'_{12}$ is a subtype of $\tau_{21}\,\tau'_{22}$ for any τ'_{12} and τ'_{22}.

Rule TSUB-EQ says that subtyping includes type equivalence ($\tau_1 \equiv \tau_2$), which is merely β equivalence on types. Rule TSUB-OMNI allows any pair of types to be related by ϕ-variant subtyping, and rule TSUB-CONTRA says that the opposite variance sign gives the inverse relation. Rules TSUB-ABS and TSUB-ALL specify that type operators and universally quantified types are related if their bodies are.

Rule TSUB-ARR is more than the usual arrow subtyping rule. Beyond the usual contravariance for arguments and covariance for results, it requires that qualifiers ξ_1 and ξ_2 relate in the same direction. This rule is the source

of non-trivial subtyping in ${}^a\lambda_{ms}$, without which subtyping would relate only equivalent types. The rule has two important implications.

First, an unlimited-use function can always be used where a one-shot function is expected. This corresponds to linear logic's usual dereliction rule, which says that the ! ("of course!") modality may always be removed. Intuitionistic linear logic (Bierman 1993), for example, has this rule:

$$\text{ILL-DERELICTION}\quad \frac{\Delta \vdash e : !\tau}{\Delta \vdash \mathsf{derelict}\ e : \tau}.$$

Dereliction is syntax-directed in this rule, but for practical programming I consider that as too inconvenient. Thus, ${}^a\lambda_{ms}$'s subtyping relation supports dereliction as needed.

For example, the function for creating a new thread in Alms, *Thread.fork*, has type $(\mathsf{unit} \xrightarrow{\mathsf{A}} \mathsf{unit}) \xrightarrow{\mathsf{U}} \mathsf{thread}$, which means that *Thread.fork* will not call its argument more than once. However, this should not stop us from passing an unlimited-use function to *Thread.fork*, and indeed we can. Dereliction subtyping allows us to use a value of type $\mathsf{unit} \xrightarrow{\mathsf{U}} \mathsf{unit}$ where a value of type $\mathsf{unit} \xrightarrow{\mathsf{A}} \mathsf{unit}$ is expected. Alternatively, by domain contravariance, we can use *Thread.fork* where a value of type $(\mathsf{unit} \xrightarrow{\mathsf{U}} \mathsf{unit}) \xrightarrow{\mathsf{U}} \mathsf{thread}$ is expected. In this case subsumption allows us to forget *Thread.fork*'s promise not to reuse its argument.

The second important implication of dereliction subtyping will be clearer in light of how qualifier expressions are assigned to function types. Subsumption makes it reasonable to always assign functions the most permissive safe usage qualifier, because subsumption then allows us to use them in a less permissive context.

Dereliction subtyping applies only to function types because only function types carry qualifiers, in both the ${}^a\lambda_{ms}$ calculus and Alms language. For instance, Alms has no separate types $\mathsf{U}\mathsf{int}$ for unlimited integers and $\mathsf{A}\mathsf{int}$ for affine integers. Rather, integers are always unlimited. A programmer who wants an affine integer type can define it in Alms using the module system.

$\boxed{\Gamma \vdash \Sigma \preceq \xi}$ *(bound of typing context)*

$$\textsc{B-Nil}\ \frac{\vdash \Gamma}{\Gamma \vdash \bullet \preceq \mathsf{U}} \qquad \textsc{B-ConsX}\ \frac{\Gamma \vdash \Sigma \preceq \xi_1 \qquad \Gamma \vdash \tau : \xi_2}{\Gamma \vdash \Sigma, x{:}\tau \preceq \xi_1 \sqcup \xi_2} \qquad \textsc{B-ConsL}\ \frac{\Gamma \vdash \Sigma \preceq \xi_1 \qquad \Gamma \vdash \tau : \xi_2}{\Gamma \vdash \Sigma, \ell{:}\tau \preceq \mathsf{A}}$$

$$\textsc{B-ConsA}\ \frac{\Gamma \vdash \Sigma \preceq \xi \qquad \Gamma \vdash \kappa}{\Gamma \vdash \Sigma, \alpha{:}\kappa \preceq \xi}$$

$\boxed{\vdash \Gamma; \Sigma}$ *(dual context well-formedness)*

$$\textsc{Wf}\ \frac{\Gamma \vdash \Gamma \preceq \mathsf{U} \qquad \Gamma \vdash \Sigma \preceq \xi}{\vdash \Gamma; \Sigma}$$

$\boxed{\vdash (\Gamma_0; \Sigma_0), \Sigma' \rightsquigarrow \Gamma_1; \Sigma_1}$ *(environment extension)*

$$\textsc{X-Nil}\ \frac{\vdash \Gamma; \Sigma}{\vdash (\Gamma; \Sigma), \bullet \rightsquigarrow \Gamma; \Sigma} \qquad \textsc{X-ConsU}\ \frac{\Gamma_0 \vdash \tau : \mathsf{U} \qquad \vdash (\Gamma_0, x{:}\tau; \Sigma_0), \Sigma' \rightsquigarrow \Gamma_1; \Sigma_1}{\vdash (\Gamma_0; \Sigma_0), x{:}\tau, \Sigma' \rightsquigarrow \Gamma_1; \Sigma_1}$$

$$\textsc{X-ConsA}\ \frac{\Gamma_0 \vdash \tau : \xi \qquad \vdash (\Gamma_0; \Sigma_0, x{:}\tau), \Sigma' \rightsquigarrow \Gamma_1; \Sigma_1}{\vdash (\Gamma_0; \Sigma_0), x{:}\tau, \Sigma' \rightsquigarrow \Gamma_1; \Sigma_1}$$

Figure 5.11: Statics (iv): typing contexts

Context judgments. Figure 5.11 defines three judgments on contexts. Judgment $\Gamma \vdash \Sigma \preceq \xi$, which will be important in typing functions, computes an upper bound ξ on the qualifiers of all the types in context Σ. If a context contains any locations, it is bounded by A; otherwise, its bound is the least upper bound of the qualifiers of all the types of variables in the context.

The typing judgment for terms will use two typing contexts in the style of Dual Intuitionistic Linear Logic (Barber 1996): Γ holds environment information that may be safely duplicated, such as type variables and variables of unlimited type, whereas Σ holds information, such as location types and affine variables, that disallows duplication. The other two judgments in figure 5.11 deal with such pairs of contexts $\Gamma;\Sigma$. Judgment $\vdash \Gamma;\Sigma$ checks the well-formedness of a pair of contexts: Context Γ must be bounded by U, and context Σ must be bounded by some qualifier, which ensures that its contents are well formed. The third judgment shows how environments are extended by variable bindings. Given contexts Γ_0 and Σ_0, judgment $\vdash (\Gamma_0;\Sigma_0), \Sigma' \rightsquigarrow \Gamma;\Sigma$ extends them by the variables and types in Σ' to get Γ and Σ. Any variables may be placed in Σ, but only variables whose types are known to be unlimited may be placed in Γ, since Γ may be duplicated.

Expression judgment. The typing judgment for expressions appears in figure 5.12. The judgment, $\Gamma;\Sigma \rhd e : \tau$, uses two typing contexts as discussed above: the unlimited environment Γ and the affine environment Σ. When typing multiplicative expression forms such as application, the typing rules distribute Γ to both subexpressions but partition Σ between the two:

$$\frac{\Gamma;\Sigma_1 \rhd e_1 : \tau_1 \xrightarrow{\xi}_{\circ} \tau_2 \qquad \Gamma;\Sigma_2 \rhd e_2 : \tau_1}{\Gamma;\Sigma_1,\Sigma_2 \rhd e_1\, e_2 : \tau_2}\ \text{T-App.}$$

Unlike DILL, not all types in Σ are necessarily affine. Since types whose usage qualifier involves type variables are not known to be unlimited, those are placed in Σ to ensure that we do not duplicate values that *might* turn out to be affine once universally quantified types are instantiated.

The other multiplicative rules are T-Pair for product introduction, T-Unpair for product elimination, and T-Swap for reference updates. Note that

$\boxed{\Gamma;\Sigma \rhd e : \tau}$ *(typing of expressions)*

T-Subsume
$$\frac{\Gamma;\Sigma \rhd e : \tau' \qquad \Gamma \vdash \tau' <:^+ \tau \qquad \Gamma \vdash \tau : \xi}{\Gamma;\Sigma \rhd e : \tau}$$

T-Weak
$$\frac{\Gamma;\Sigma \rhd e : \tau \qquad \vdash \Gamma;\Sigma,\Sigma'}{\Gamma;\Sigma,\Sigma' \rhd e : \tau}$$

T-Var
$$\frac{x{:}\tau \in \Gamma,\Sigma \qquad \Gamma \vdash \tau : \xi \qquad \vdash \Gamma;\Sigma}{\Gamma;\Sigma \rhd x : \tau}$$

T-Ptr
$$\frac{\ell{:}\tau \in \Sigma \qquad \bullet \vdash \tau : \xi \qquad \vdash \Gamma;\Sigma}{\Gamma;\Sigma \rhd \ell : \mathsf{ref}\ \tau}$$

T-Abs
$$\frac{\vdash (\Gamma;\Sigma), x{:}\tau_1 \rightsquigarrow \Gamma';\Sigma' \qquad \Gamma';\Sigma' \rhd e : \tau_2 \qquad \Gamma \vdash \Sigma \preceq \xi \qquad \Gamma \vdash \tau_1 : \xi_1}{\Gamma;\Sigma \rhd \lambda x{:}\tau_1.e : \tau_1 \xrightarrow{\xi}_{\circ} \tau_2}$$

T-TAbs
$$\frac{\Gamma,\alpha{:}\kappa;\Sigma \rhd v : \tau \qquad \Gamma \vdash \kappa}{\Gamma;\Sigma \rhd \Lambda\alpha{:}\kappa.v : \forall\alpha{:}\kappa.\tau}$$

T-App
$$\frac{\Gamma;\Sigma_1 \rhd e_1 : \tau_1 \xrightarrow{\xi}_{\circ} \tau_2 \qquad \Gamma;\Sigma_2 \rhd e_2 : \tau_1}{\Gamma;\Sigma_1,\Sigma_2 \rhd e_1\, e_2 : \tau_2}$$

T-TApp
$$\frac{\Gamma;\Sigma \rhd e : \forall\alpha{:}\kappa.\tau \qquad \Gamma \vdash \tau' : \kappa' \qquad \Gamma \vdash \kappa' <: \kappa}{\Gamma;\Sigma \rhd e\,\tau' : \{\tau'/\alpha\}\tau}$$

T-Fix
$$\frac{\Gamma;\Sigma \rhd e : \tau \xrightarrow{\cup}_{\circ} \tau}{\Gamma;\Sigma \rhd \mathsf{fix}\ e : \tau}$$

T-Unit
$$\frac{\vdash \Gamma;\Sigma}{\Gamma;\Sigma \rhd \langle\rangle : 1}$$

T-Inl
$$\frac{\Gamma;\Sigma \rhd e : \tau_1 \qquad \Gamma \vdash \tau_2 : \xi}{\Gamma;\Sigma \rhd \mathsf{inl}\ e : \tau_1 \oplus \tau_2}$$

T-Inr
$$\frac{\Gamma;\Sigma \rhd e : \tau_2 \qquad \Gamma \vdash \tau_1 : \xi}{\Gamma;\Sigma \rhd \mathsf{inr}\ e : \tau_1 \oplus \tau_2}$$

T-Pair
$$\frac{\Gamma;\Sigma_1 \rhd e_1 : \tau_1 \qquad \Gamma;\Sigma_2 \rhd e_2 : \tau_2}{\Gamma;\Sigma_1,\Sigma_2 \rhd \langle e_1, e_2\rangle : \tau_1 \otimes \tau_2}$$

T-Case
$$\frac{\begin{array}{c}\Gamma;\Sigma \rhd e : \tau_1 \oplus \tau_2 \\ \vdash (\Gamma;\Sigma'), x_1{:}\tau_1 \rightsquigarrow \Gamma_1;\Sigma_1 \qquad \Gamma_1;\Sigma_1 \rhd e_1 : \tau \\ \vdash (\Gamma;\Sigma'), x_2{:}\tau_2 \rightsquigarrow \Gamma_2;\Sigma_2 \qquad \Gamma_2;\Sigma_2 \rhd e_2 : \tau\end{array}}{\Gamma;\Sigma,\Sigma' \rhd \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x_1 \to e_1; \mathsf{inr}\ x_2 \to e_2 : \tau}$$

T-Unpair
$$\frac{\Gamma;\Sigma_1 \rhd e : \tau_1 \otimes \tau_2 \qquad \vdash (\Gamma;\Sigma_2), x_1{:}\tau_1, x_2{:}\tau_2 \rightsquigarrow \Gamma';\Sigma' \qquad \Gamma';\Sigma' \rhd e_1 : \tau}{\Gamma;\Sigma_1,\Sigma_2 \rhd \mathsf{let}\ \langle x_1, x_2\rangle = e\ \mathsf{in}\ e_1 : \tau}$$

T-New
$$\frac{\Gamma;\Sigma \rhd e : \tau}{\Gamma;\Sigma \rhd \mathsf{new}\ e : \mathsf{ref}\ \tau}$$

T-Swap
$$\frac{\Gamma;\Sigma_1 \rhd e_1 : \mathsf{ref}\ \tau_1 \qquad \Gamma;\Sigma_2 \rhd e_2 : \tau_2}{\Gamma;\Sigma_1,\Sigma_2 \rhd \mathsf{swap}\ e_1\, e_2 : \mathsf{ref}\ \tau_2 \otimes \tau_1}$$

T-Delete
$$\frac{\Gamma;\Sigma \rhd e : \mathsf{ref}\ \tau}{\Gamma;\Sigma \rhd \mathsf{delete}\ e : 1}$$

Figure 5.12: Statics (v): expressions

T-SWAP does not require that the type of the reference in its first parameter match the type of the value in its second—in other words, swap performs a *strong update*. To type the term $\mathsf{let}\,\langle x, y\rangle = e_1\,\mathsf{in}\,e_2$, rule T-UNPAIR first splits the affine environment into Σ_1 for typing subterm e_1 and Σ_2 for subterm e_2. It invokes the context extension relation (figure 5.11) to extend Γ and Σ_2 with bindings for x and y in order to type e_2. The context extension relation requires that variables not known to be unlimited be added to Σ_2.

The rule for sum elimination, T-CASE, is both multiplicative and additive: The affine context is split between the term being destructed and the branches of the case expression. However, the portion of the context given to the branches is shared between them, because only one or the other will be evaluated. Rule T-CASE also uses the context extension relation to bind the pattern variables for the branches.

Rules T-NEW and T-DELETE introduce and eliminate reference types in the usual way. Likewise, the sum introduction rules T-INL and T-INR and type abstraction rule T-TABS are standard. Rules T-VAR, T-PTR, and T-UNIT are standard for an affine calculus but not a linear one, as they implicitly support weakening by allowing Σ to contain unused bindings. Rule T-FIX is also standard, up to the reasonable constraint that its parameter function be unlimited, since the reduction rule for fix makes a copy of the parameter.

The type application rule T-TAPP supports subkinding, because it requires only that the kind of the actual type parameter be a subkind of that of the formal parameter. This is the rule that supports the sort of type abstraction that I used in the examples of chapter 3 to construct affine capabilities. For example, the rule allows instantiating affine type variable α with unlimited unit type 1:

$$\frac{\Gamma;\Sigma \rhd (\Lambda\alpha{:}\mathsf{A}.\,\lambda x{:}\alpha.\,e) : \forall\alpha{:}\mathsf{A}.\,\alpha \xrightarrow{\xi}_{\circ} \tau \qquad \Gamma \vdash 1 : \mathsf{U} \qquad \Gamma \vdash \mathsf{U} <: \mathsf{A}}{\Gamma;\Sigma \rhd (\Lambda\alpha{:}\mathsf{A}.\,\lambda x{:}\alpha.\,e)\,1 : 1 \xrightarrow{\xi}_{\circ} \tau}\ \text{T-TAPP}.$$

Within its scope, α is considered *a priori* affine, regardless of how it may eventually be instantiated. This expression types only if x appears in affine fashion in e.

This brings us finally to T-ABS, the rule for typing expression-level λ abstractions. To type an expression $\lambda x{:}\tau_1.\, e$, rule T-ABS uses the context extension relation to add $x{:}\tau_1$ to its contexts and types the body e in the extended contexts. It also must determine the qualifier ξ that decorates the arrow. Because abstractions close over their free variables, duplicating a function also duplicates the values of its free variables. Therefore, the qualifier of a function type should be at least as restrictive as the qualifiers of the abstraction's free variables. To do this, rule T-ABS appeals to the context bounding judgment (figure 5.11) to find the least upper bound of the usage qualifiers of variables in the affine environment, and it requires that the function type's qualifier be equally restrictive.

This refines linear logic's usual promotion rule, which says that the ! modality may be added to propositions that in turn depend only on !-ed resources. In ILL, we have

$$\textsc{Promotion}\quad \frac{!\Delta \vdash e : \tau}{!\Delta \vdash \mathsf{promote}\ e : !\tau},$$

where $!\Delta$ is a context in which all assumptions are of the form $x : !\sigma$. As with dereliction, in ${}^{a}\lambda_{ms}$ it only makes sense to apply promotion to function types.

My treatment of promotion indicates why ${}^{a}\lambda_{ms}$ needs the explicit weakening rule T-WEAK, which allows discarding unused portions of the affine environment. In order to give a function type the best qualifier possible, we need to remove from Σ any unused variables or locations that might otherwise raise the bound on Σ, and the algorithmic version of the type system as implemented in Alms does just that (chapter 6). In §5.2, I show that this implicit promotion mechanism selects the best usage qualifier for function types.

Store and configuration judgments. In order to prove my type soundness theorem, I need to lift the typing judgments to stores and run-time configurations.

The type of a store is a typing context containing the names of the store's locations and the types of their contents. The store typing judgment $\Sigma_1 \rhd s : \Sigma_2$ gives store s type Σ_2 in the context of Σ_1, which is necessary because values in

$\boxed{\Sigma_1 \rhd s : \Sigma_2}$ *(store typing)*

$$\text{S-NIL}\ \frac{}{\Sigma \rhd \{\} : \bullet} \qquad \text{S-CONS}\ \frac{\Sigma_1 \rhd s : \Sigma' \qquad \bullet; \Sigma_2 \rhd v : \tau}{\Sigma_1, \Sigma_2 \rhd s \uplus \{\ell \mapsto v\} : \Sigma', \ell{:}\tau}$$

$\boxed{\rhd (s, e) : \tau}$ *(configuration typing)*

$$\text{CONF}\ \frac{\Sigma_1 \rhd s : \Sigma_1, \Sigma_2 \qquad \bullet; \Sigma_2 \rhd e : \tau}{\rhd (s, e) : \tau}$$

Figure 5.13: Statics (vi): stores and configurations

the store may refer to other values in the store. Rule S-CONS shows that the resources represented by context Σ_1 (*i.e.,* Σ_{11}, Σ_{12}) are split between the values in s.

The preservation lemma in the next section concerns typing judgments on configurations, $\rhd (s, e) : \tau$, which means that e has type τ in the context of store s. To type the configuration by rule CONF, we type the store, splitting its type into Σ_1, which contains locations referenced from the store, and Σ_2, which contains locations referenced from expression e.

5.2 Theoretical Results

In this section I state and prove two main theorems about ${}^a\lambda_{ms}$: its principal qualifiers property and syntactic type soundness.

5.2.1 Principal Qualifiers

Alms and ${}^a\lambda_{ms}$ go to a lot of trouble to find the best usage qualifier expressions for function types. To make programming with affine types as convenient as possible, I want to maximize polymorphism between one-shot and unlimited versions of functions. While writing the Alms standard library, I found that

usage qualifier constants **A** and **U**, even with dereliction subtyping, were insufficient to give some terms a principal type.

For example, consider an Alms function *default*, an eliminator for option types:

```
let default def opt =
  match opt with
  | Some x → x
  | None → def
```

Without usage qualifier expressions, *default* has at least two types:

$$default_1 : \forall \text{`}a.\ \text{`}a \xrightarrow{\mathsf{U}} \text{`}a\ \mathsf{option} \xrightarrow{\mathsf{A}} \text{`}a$$

versus the incomparable

$$default_2 : \forall \text{'}a.\ \text{'}a \xrightarrow{\mathsf{U}} \text{'}a\ \mathsf{option} \xrightarrow{\mathsf{U}} \text{'}a.$$

In the first case, because `a *might* be affine, the partial application of $default_1$ must be a one-shot function, but in the second case we know that 'a is unlimited so partially applying $default_2$ and reusing the result is safe. Formally, these types are incomparable because the universally quantified type variable `a in the former has a different kind than 'a in the latter, and Alms uses the *kernel* variant of rule TSUB-ALL. However, even were we to replace rule TSUB-ALL with a rule analogous to $F^{\omega}_{<:}$'s *full* variant,

$$\frac{\Gamma, \alpha{:}\kappa \vdash \tau_1 <:^{\mathfrak{v}} \tau_2 \qquad \Gamma, \alpha{:}\kappa \vdash \kappa_1 <:^{-\mathfrak{v}} \kappa_2}{\Gamma \vdash \forall \alpha{:}\kappa_1.\, \tau_1 <:^{\mathfrak{v}} \forall \alpha{:}\kappa_2.\, \tau_2}\ \textsc{Tsub-All}_{\textsc{full}},$$

the types would not be related by the subtyping order. More importantly, neither type is preferable in an informal sense. The type of $default_1$ allows `a to be instantiated to an affine or unlimited type, but the result of partially applying it is a one-shot function even if `a is known to be unlimited:

$default_1\ 5$: $\mathsf{int\ option} \xrightarrow{\mathsf{A}} \mathsf{int}$
$default_1\ (aref\ 5)$: $\mathsf{int\ aref\ option} \xrightarrow{\mathsf{A}} \mathsf{int\ aref}$

If we choose $default_2$, the result of partial application is unlimited, but attempting to instantiate 'a to an affine type is a type error:

$default_2\ 5$: int option $\xrightarrow{\mathsf{U}}$ int
$default_2\ (aref\ 5)$: **TYPE ERROR!**

Alms avoids both problems and instead discovers that the best usage qualifier for the arrow is the kind of the type variable:

$default$: $\forall\, \text{`}a.\ \text{`}a \xrightarrow{\mathsf{U}} \text{`}a$ option $\xrightarrow{\text{`}a} \text{`}a$
$default\ 5$: int option $\xrightarrow{\mathsf{U}}$ int
$default\ (aref\ 5)$: int aref option $\xrightarrow{\mathsf{A}}$ int aref

Because this is an important property, I prove a theorem that every typeable ${}^{a}\lambda_{ms}$ function has a principal usage qualifier.

THEOREM 5.3 (Principal qualifiers)**.**

If $\Gamma;\Sigma \rhd \lambda x{:}\tau.e : \tau_1 \xrightarrow{\xi}\circ\ \tau_2$*, then it has a least qualifier expression* ξ_0*; that is,*

- $\Gamma;\Sigma \rhd \lambda x{:}\tau.e : \tau_1 \xrightarrow{\xi_0}\circ\ \tau_2$ *and*
- $\Gamma \vdash \xi_0 <: \xi'$ *for all* ξ' *such that* $\Gamma;\Sigma \rhd \lambda x{:}\tau.e : \tau_1 \xrightarrow{\xi'}\circ\ \tau_2$.

Proof. By assumption, $\Gamma;\Sigma \rhd \lambda x{:}\tau_1.e : \tau_1' \xrightarrow{\xi'}\circ\ \tau_2'$. Three rules apply at the root of the typing derivation: the syntax-directed rule T-ABS, the subsumption rule T-SUBSUME, and the weakening rule T-WEAK. Without loss of generality, multiple subsumptions may be collapsed to one by transitivity of subtyping, or zero subsumptions expanded to one by reflexivity of subtyping. Likewise, multiple weakenings may be collapsed to one, and zero may be expanded to one, which weakens by the empty context. Noting that weakening and subsumption commute, it is only necessary to consider derivations of the form

$$\dfrac{\dfrac{\dfrac{\mathcal{A} : \vdash (\Gamma;\Sigma), x{:}\tau_1 \rightsquigarrow \Gamma';\Sigma' \quad \mathcal{B} : \Gamma';\Sigma' \rhd e : \tau_2 \quad \mathcal{C} : \Gamma \vdash \Sigma \preceq \xi \quad \mathcal{D} : \Gamma \vdash \tau_1 : \xi_1}{\Gamma;\Sigma \rhd \lambda x{:}\tau_1.e : \tau_1 \xrightarrow{\xi}\circ\ \tau_2}\ \text{T-ABS} \qquad \dfrac{\mathcal{J} : \Gamma \vdash \tau_1 <:^{-} \tau_1' \quad \mathcal{K} : \Gamma \vdash \xi <: \xi' \quad \mathcal{L} : \Gamma \vdash \tau_2 <:^{+} \tau_2'}{\Gamma \vdash \tau_1 \xrightarrow{\xi}\circ\ \tau_2 <:^{+} \tau_1' \xrightarrow{\xi'}\circ\ \tau_2'}\ \text{TSUB-ARR}}{\Gamma;\Sigma \rhd \lambda x{:}\tau_1.e : \tau_1' \xrightarrow{\xi'}\circ\ \tau_2'}\ \text{T-SUBSUME}}{\Gamma;\Sigma,\Sigma_w \rhd \lambda x{:}\tau_1.e : \tau_1' \xrightarrow{\xi'}\circ\ \tau_2'}\ \text{T-WEAK.}$$

Now let Σ_0 be Σ restricted to the free variables and locations of $\lambda x{:}\tau_1.e$. If $(x{:}\tau_1) \in \Sigma'$ then let $\Sigma'_0 = \Sigma_0, x{:}\tau_1$; otherwise let $\Sigma'_0 = \Sigma_0$. Thus, $\vdash (\Gamma;\Sigma_0), x{:}\tau_1 \rightsquigarrow \Gamma';\Sigma'_0$. By lemma A.8 (Unique context bounds), let ξ_0 be the unique qualifier expression such that $\Gamma \vdash \Sigma_0 \preceq \xi_0$. By inspection of the typing rules, we may strengthen $\mathcal{B}$ to $\Gamma';\Sigma'_0 \rhd e : \tau_2$, because we removed only irrelevant assumptions from Σ'_0. Thus, we can derive:

$$\dfrac{\dfrac{\dfrac{\mathcal{A}' : \vdash (\Gamma;\Sigma_0), x{:}\tau_1 \rightsquigarrow \Gamma';\Sigma'_0 \quad \mathcal{B}' : \Gamma';\Sigma'_0 \rhd e : \tau_2 \quad \mathcal{C}' : \Gamma \vdash \Sigma_0 \preceq \xi_0 \quad \mathcal{D} : \Gamma \vdash \tau_1 : \xi_1}{\Gamma;\Sigma' \rhd \lambda x{:}\tau_1.e : \tau_1 \xrightarrow{\xi_0\circ} \tau_2}\ \text{T-Abs} \qquad \dfrac{\mathcal{J} : \Gamma \vdash \tau_1 <:^{-} \tau'_1 \quad \mathcal{K}' : \Gamma \vdash \xi_0 <: \xi_0 \quad \mathcal{L} : \Gamma \vdash \tau_2 <:^{+} \tau'_2}{\Gamma \vdash \tau_1 \xrightarrow{\xi_0\circ} \tau_2 <:^{+} \tau'_1 \xrightarrow{\xi_0\circ} \tau'_2}\ \text{TSub-Arr}}{\Gamma;\Sigma' \rhd \lambda x{:}\tau_1.e : \tau'_1 \xrightarrow{\xi_0\circ} \tau'_2}\ \text{T-Subsume}}{\Gamma;\Sigma,\Sigma_w \rhd \lambda x{:}\tau_1.e : \tau'_1 \xrightarrow{\xi_0\circ} \tau'_2}\ \text{T-Weak.}$$

We now must show that ξ_0 is the least usage qualifier that can be given to $\lambda x{:}\tau_1.e$. Since ξ_0 is the least upper bound for Σ_0, the only way to get a lower qualifier would be to remove some variables from Σ_0, but we defined Σ_0 to contain only variables relevant to $\lambda x{:}\tau_1.e$, which means that nothing else can be removed. □

5.2.2 Type Soundness

The key obstacle in proving type soundness is establishing a substitution lemma, which in turn relies on showing that the kind of the type of any value accurately reflects the resources contained in that value, which itself comes as a corollary to the proposition that the kinds of subtypes are themselves subkinds.

Lemma 5.4 (Monotonicity of kinding)**.**

If $\Gamma \vdash \tau_1 <:^{+} \tau_2$ *where* $\Gamma \vdash \tau_1 : \xi_1$ *and* $\Gamma \vdash \tau_2 : \xi_2$*, then* $\Gamma \vdash \xi_1 <: \xi_2$.

This lemma is the reason for the premise in rule OK-Oper that for a kind $\Pi\alpha^{\mathfrak{v}}.\kappa$, variance $\mathfrak{v}$ must be at least + if $\alpha \in \mathrm{FTV}(\kappa)$. Otherwise, I could construct a counterexample to lemma 5.4:

- $\beta{:}\Pi\alpha^{-}.\langle\alpha\rangle \vdash \beta(1 \xrightarrow{\mathsf{A}} 1) <:^{+} \beta(1 \xrightarrow{\mathsf{U}} 1)$,
- $\beta{:}\Pi\alpha^{-}.\langle\alpha\rangle \vdash \beta(1 \xrightarrow{\mathsf{A}} 1) : \mathsf{A}$, and
- $\beta{:}\Pi\alpha^{-}.\langle\alpha\rangle \vdash \beta(1 \xrightarrow{\mathsf{U}} 1) : \mathsf{U}$,
- but $\beta{:}\Pi\alpha^{-}.\langle\alpha\rangle \vdash \mathsf{A} <: \mathsf{U}$ is not the case.

The kind well-formedness judgment rules out kinds like $\Pi\alpha^{-}.\langle\alpha\rangle$.

Proof of lemma 5.4. I define an extension of the subkinding relation, $\Gamma \vdash \kappa_1 \lessapprox \kappa_2$, which is insensitive to the variances decorating Π kinds. Observe that on qualifier expressions this new relation coincides with subkinding. Generalize the induction hypothesis—if $\Gamma \vdash \tau_1 <:^{+} \tau_2$ where $\Gamma \vdash \tau_1 : \kappa_1$ and $\Gamma \vdash \tau_2 : \kappa_2$, then $\Gamma \vdash \kappa_1 \lessapprox \kappa_2$—and the proof follows by induction on the structure of the subtyping derivation.

Please see p. 288 for details. ▷

LEMMA 5.5 (Kinding finds locations)**.**

Suppose that $\Gamma;\Sigma \rhd v : \tau$ *and* $\Gamma \vdash \tau : \xi$*. If any locations appear in value* v *then* $\Gamma \vdash \mathsf{A} <: \xi$.

Proof. By induction on the typing derivation, using the previous lemma in the case for the subsumption rule T-SUBSUME:

Case $\dfrac{\Gamma;\Sigma \rhd v : \tau' \qquad \Gamma \vdash \tau' <:^{+} \tau \qquad \Gamma \vdash \tau : \xi}{\Gamma;\Sigma \rhd v : \tau}$**.**

By the induction hypothesis, $\Gamma \vdash \tau' : \mathsf{A}$, and by lemma 5.4 (Monotonicity of kinding), $\Gamma \vdash \mathsf{A} <: \xi$.

Please see p. 291 for details. ▷

Lemma 5.5 lets me prove a substitution lemma.

LEMMA 5.6 (Substitution)**.**

If

- $\vdash (\Gamma; \Sigma_1), x{:}\tau' \rightsquigarrow \Gamma'; \Sigma_1'$,
- $\Gamma'; \Sigma_1' \rhd e : \tau$, *and*
- $\bullet; \Sigma_2 \rhd v : \tau'$, *where*
- *the domain of* Σ_2 *contains only locations,*

then $\Gamma; \Sigma_1, \Sigma_2 \rhd \{v/x\}\, e : \tau$.

Proof. By induction on the derivation of $\Gamma'; \Sigma_1' \rhd e : \tau$. Please see p. 293 for details. ▷

Now progress, preservation, and type soundness are standard.

LEMMA 5.7 (Progress)**.**

If $\rhd (s, e) : \tau$ *then either* e *is a value, or there exist some* s' *and* e' *such that* $(s, e) \longmapsto (s', e')$.

Proof. Since $\rhd (s, e) : \tau$, by lemma A.46 (Faulty expressions), (s, e) is not faulty (definition A.39 on p. 319). Furthermore, since it types, e must be closed. Then by lemma A.40 (Uniform evaluation), either e is a value or the configuration takes a step. □

LEMMA 5.8 (Preservation)**.**

If $\rhd (s, e) : \tau$ *and* $(s, e) \longmapsto (s', e')$ *then* $\rhd (s', e') : \tau$.

Proof. By cases on the reduction relation. Please see p. 304 for details. ▷

THEOREM 5.9 (Type soundness)**.**

If $\rhd (\{\}, e) : \tau$ *then either* e *diverges or there exists some store* s *and value* v *such that* $(\{\}, e) \overset{*}{\longmapsto} (s, v)$ *and* $\rhd (s, v) : \tau$.

Proof. By lemma 5.7 (Progress), lemma 5.8 (Preservation), and induction on the length of the reduction sequence. □

CHAPTER 6

Implementation of Alms

IN ORDER TO validate the practicality and expressiveness of Alms's design, I built a prototype implementation of the language and wrote libraries and programs using it. Some of these programs appear in chapter 4. In this chapter, I report on interesting aspects of the implementation.

Alms is implemented in around 23k lines of Haskell code, which I break down by function in figure 6.1. The majority of the code is straightforward and similar to any other language implementation, but one portion stands out as non-obvious: type inference with affine types and subtyping.

In the implementation of Alms, type inference deals with a wide variety of language features, including pattern matching, first-class polymorphism, equirecursive types, and row types. In this chapter, however, I consider inferring types for Core Alms, a simplified language of variables, applications, abstractions, and *let* expressions.

6.1 Core Alms

The syntax of Core Alms appears in figure 6.2.[1] Unlike the model of the previous chapter, which is intended to reflect several features of Alms, Core Alms is the smallest language suitable for explaining how type inference works

[1] A caveat: While I use mathematical notation throughout this chapter for clarity, the treatment is essentially informal. My goal is to communicate interesting aspects of the implementation sufficiently well that a reader could replicate it, but I have not proven anything here correct, with the exception of §6.3.3.

Purpose	**Line Count**	**% of Total**
Type checking	8,290	36.4%
Abstract syntax	3,456	15.2%
Parsing and pretty printing	3,282	14.4%
Miscellaneous support code	2,631	11.5%
Types and values of primitives	1,655	7.3%
Implicit threading syntax	1,141	5.0%
Error handling infrastructure	956	4.2%
Dynamics	882	3.9%
REPL and library loading	499	2.2%
Total	22,792	100 %

Figure 6.1: Alms source code size by function

e	::=		expressions
	|	x	variable
	|	$\lambda x.e$	abstraction
	|	$e_1\, e_2$	application
	|	$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$	*let* expression
τ	::=		types
	|	α	type variable
	|	$\tau_1 \xrightarrow{\xi} \tau_2$	function type
	|	$\chi\, \tau_1 \ldots \tau_k$	applied type constructor
σ	::=		type schemes
	|	τ	monotype
	|	$\forall \alpha{:}\mathfrak{q}.\sigma$	universal type
ξ	::=		usage qualifier expressions
	|	$\mathfrak{q}$	qualifier constant
	|	$\langle \alpha \rangle$	qualifier of type variable
	|	$\xi_1 \sqcup \xi_2$	least upper bound
κ	::=		kinds
	|	$\Pi(\alpha_1^{\mathfrak{v}_1}, \ldots, \alpha_k^{\mathfrak{v}_k}).\xi$	kind of arity-k type constructor

Figure 6.2: Syntax of Core Alms

C, D	::=		constraints
	|	$\top$	the trivial constraint
	|	$C \wedge D$	conjunction of constraints
	|	$\tau_1 \leq \tau_2$	τ_1 is a subtype of τ_2
	|	$\exists \alpha . C$	existential quantification (freshness)
	|	...	etc.

$\boxed{\theta \models C}$ *(constraint satisfaction)*

$$\text{C-TRUE}\quad \theta \models \top \qquad \text{C-SUBTYPE}\quad \frac{\vdash \theta\tau_1 \leq \theta\tau_2}{\theta \models \tau_1 \leq \tau_2} \qquad \text{C-CONJ}\quad \frac{\theta \models C \qquad \theta \models D}{\theta \models C \wedge D} \qquad \text{C-EXISTS}\quad \frac{\theta \circ \{\tau/\alpha\} \models C}{\theta \models \exists \alpha . C}$$

$\boxed{C \Vdash D}$ *(constraint entailment)*

$$\text{C-ENTAILS}\quad \frac{(\forall \theta)\, \theta \models C \Longrightarrow \theta \models D}{C \Vdash D}$$

Figure 6.3: Syntax and semantics of constraints

in Alms. As in ML, types (τ) do not contain quantifiers, but type schemes (σ) are universally quantified. Bound type variables are bounded above by qualifier constants ($\mathsf{q} \in \{\mathsf{U}, \mathsf{A}\}$), which limit which types may instantiate them to those whose qualifier is less than or equal to the upper bound. Type variables bounded by U correspond to unlimited type variables (’α) in Alms, whereas type variables bounded by A correspond to affine type variables (‘α). Like Alms, function types carry a qualifier expression (ξ), which controls how many times a function may be used. Types also include applications of (unspecified) type constructors (χ), which have dependent kinds (κ) as in Alms.

6.1.1 Background: HM(X)

Type inference for (Core) Alms is based on HM(X) (Odersky et al. 1999), which is the type system of Damas and Milner (1982) parametrized by a constraint

$\boxed{C;\Gamma \vdash e : \tau}$ *(expression typing)*

$$\text{HMX-VAR}\quad \frac{x{:}\forall\overline{\alpha_i}[D].\tau \in \Gamma \qquad C \Vdash \overline{\{\tau_i/\alpha_i\}D}}{C;\Gamma \vdash x : \overline{\{\tau_i/\alpha_i\}\tau}} \qquad \text{HMX-ABS}\quad \frac{C;\Gamma, x{:}\tau' \vdash e : \tau}{C;\Gamma \vdash \lambda x.e : \tau' \to \tau}$$

$$\text{HMX-APP}\quad \frac{C;\Gamma \vdash e_1 : \tau_1 \qquad C;\Gamma \vdash e_2 : \tau_2 \qquad C \Vdash \tau_1 \le \tau_2 \to \tau}{C;\Gamma \vdash e_1\, e_2 : \tau}$$

$$\text{HMX-LET}\quad \frac{C \wedge D;\Gamma \vdash e_1 : \tau' \qquad C;\Gamma, x{:}\forall\overline{\alpha_i}[D].\tau' \vdash e_2 : \tau \qquad \overline{\alpha_i} \notin \mathrm{FTV}(C) \cup \mathrm{FTV}(\Gamma)}{C \wedge \exists\overline{\alpha_i}.D;\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau}$$

Figure 6.4: HM(X) (syntax directed, with subtyping)

system. Before considering how to deal with affine types, I review the basics of HM(X) for a non-affine language with subtyping.

Constraints (figure 6.3) include at least the trivial constraint ($\top$), subtyping, conjunction of constraints, and existential quantification (which represents freshness). Constraints are considered up to equivalence, which includes commutativity and associativity of conjunction, absorption of $\top$ as the identity of conjunction, and α conversion and scope extrusion of existential quantifiers. Figure 6.3 also gives two judgments that define the semantics of constraints. Constraints are interpreted in the context of an idempotent substitution θ mapping type variables to types; when $\theta \models C$, we say that θ *satisfies* C. Constraint C *entails* constraint D (judgment $C \Vdash D$) when every substitution satisfying C also satisfies D. Type schemes in HM(X) contain constraints:

$$\sigma ::= \forall\overline{\alpha_i}[C].\tau \qquad \text{HM(X) type schemes}$$

The substitution that instantiates a type scheme must satisfy the constraint.

Figure 6.4 contains a presentation of HM(X) based on Pottier and Rémy (2005). Judgment $C;\Gamma \vdash e : \tau$ assigns type τ to expression e in the context of constraint C and environment Γ, which maps variables to type schemes. Rule HMX-VAR type checks a variable whose type scheme is $\forall\overline{\alpha_i}[D].\tau$. The

$$
\begin{array}{lcll}
C, D & ::= & \cdots & \text{additional constraints} \\
 & | & \xi_1 \sqsubseteq \xi_2 & \text{qualifier } \xi_1 \text{ is a subqualifier of } \xi_2
\end{array}
$$

$\boxed{\theta \models C}$ *(constraint satisfaction)*

$$
\cdots \qquad \frac{\vdash \theta\xi_1 \sqsubseteq \theta\xi_2}{\theta \models \xi_1 \sqsubseteq \xi_2}\ \text{C-SUBQUALIFIER}
$$

Figure 6.5: Constraints for Substructural HM(X)

type scheme is instantiated by some types $\overline{\tau_i}$, and the scheme's constraint D, with types $\overline{\tau_i}$ substituted for free type variables $\overline{\alpha_i}$, must be entailed by the context's constraint C. The abstraction rule, HMX-ABS, is ordinary; it requires the bound variable to have a monotype, not a universal type scheme. Rule HMX-APP requires the type of the operator to be a subtype of the function type whose domain is the type of the operand, rather than requiring type equality.

Rule HMX-LET performs generalization and binds a variable to a type scheme. In order to generalize some type variables $\overline{\alpha_i}$, the context from checking e_1 is split into conjuncts C and D, such that $\overline{\alpha_i}$ are not free in C. (The generalized type variables also must not be free in Γ.) Then the body expression e_2 is checked with variable x bound to a type scheme whose constraint is D, which means that each use of x must instantiate $\overline{\alpha_i}$ to some types that satisfy D. The presence of $\exists\overline{\alpha_i}.D$ in the constraint in the conclusion of the rule ensures that constraint D is satisfiable even if x is not used in e_2.

6.1.2 Substructural HM(X)

Before defining the type system for Core Alms, I extend HM(X) to infer substructural types. The extension is remarkably straightforward. Constraints are extended to include subsumption constraints between qualifiers, and the satisfaction relation is extended for the new constraint form (figure 6.5). Figure 6.6 defines the qualifier of a type and the qualifier of an environment.

$\boxed{\langle\tau\rangle = \xi}, \boxed{\langle\Gamma\rangle = \xi}$ *(qualifiers)*

$$\begin{aligned}
\langle\alpha\rangle &= \langle\alpha\rangle \\
\langle\tau_1 \xrightarrow{\xi} \tau_2\rangle &= \xi \\
\langle\chi\,\overline{\tau_i}\rangle &= \overline{\{\langle\tau_i\rangle/\alpha_i\}}\xi \quad \text{if } \mathrm{kind}(\chi) = \Pi(\overline{\alpha_i^{\mathfrak{v}_i}}).\,\xi \\
\langle\bullet\rangle &= \mathsf{U} \\
\langle\Gamma, x{:}\forall\overline{\alpha_i{:}\mathsf{q}_i}.\,\tau\rangle &= \langle\Gamma\rangle \sqcup \overline{\{\mathsf{U}/\alpha_i\}}\langle\tau\rangle
\end{aligned}$$

Figure 6.6: Qualifiers of types and environments

Substructural HM(X) defines a simple usage analysis to count the number of occurrences of bound variables in their scopes. Metafunction $\mathrm{occ}_x(e)$ counts the number of occurrences of variable x in expression e, yielding a usage count $\mathfrak{u}$. A definition of occ_x for affine types appears in figure 6.7. In this case, usage counts ($\mathfrak{u} \in \mathfrak{U}$) are an abstraction of the natural numbers, with all numbers greater than 1 collapsed to many; usage counts map to qualifiers in the obvious way. This approach extends smoothly to systems with more qualifiers and additive expressions, as in figure 6.8. In order to handle additive forms such as *if* expressions, usage counts in this case are subsets of $\mathfrak{U}$, and addition is lifted to add sets of counts. The qualifier of a usage count set is the greatest lower bound of the qualifiers of its elements. For Core Alms, the affine occurrence analysis of figure 6.7 suffices.

The typing rules for Substructural HM(X) appear in figure 6.9. The rules are changed from HM(X) to track occurrences of bound variables and to deal with qualifiers on function types:

- In rule SHMX-ABS, the new premise $C \Vdash \langle\tau'\rangle \sqsubseteq \langle\mathrm{occ}_x(e)\rangle$ ensures that the qualifier of the argument type reflects its usage in expression e. Likewise, in rule SHMX-LET, the new premise $C \Vdash \langle\tau'\rangle \sqsubseteq \langle\mathrm{occ}_x(e_2)\rangle$ bounds the qualifier of the type of bound variable x based on how many times it occurs in e_2.

- In rule SHMX-ABS, the qualifier of the function type is $\langle\Gamma|_{\mathrm{FV}(\lambda x.e)}\rangle$;

$\mathfrak{u} \in \mathfrak{U} = \{0, 1, \mathsf{many}\}$ usage counts

$\boxed{\mathrm{occ}_x(e) = \mathfrak{u}}$ *(occurrence analysis)*

$$\begin{aligned}
\mathrm{occ}_x(x) &= 1 \\
\mathrm{occ}_x(y) &= 0 && (x \neq y) \\
\mathrm{occ}_x(\lambda y.\, e) &= \mathrm{occ}_x(e) \\
\mathrm{occ}_x(e_1\, e_2) &= \mathrm{occ}_x(e_1) + \mathrm{occ}_x(e_2) \\
\mathrm{occ}_x(\mathsf{let}\ y = e_1\ \mathsf{in}\ e_2) &= \mathrm{occ}_x(e_1) + \mathrm{occ}_x(e_2)
\end{aligned}$$

$\boxed{\langle \mathfrak{u} \rangle = \mathfrak{q}}$ *(qualifiers of usage counts)*

$$\langle 0 \rangle = \mathsf{A} \qquad \langle 1 \rangle = \mathsf{A} \qquad \langle \mathsf{many} \rangle = \mathsf{U}$$

Figure 6.7: Occurrence analysis for affine types

$n \in \mathfrak{U} = \{0, 1, \mathsf{many}\}$ atomic usage counts

$\mathfrak{u} \in \wp(\mathfrak{U})$ usage counts

$\boxed{\mathfrak{u}_1 \hat{+} \mathfrak{u}_2 = \mathfrak{u}}$ *(usage count addition)*

$$\mathfrak{u}_1 \hat{+} \mathfrak{u}_2 = \{n_1 + n_2 \mid n_1 \in \mathfrak{u}_1, n_2 \in \mathfrak{u}_2\}$$

$\boxed{\mathrm{occ}_x(e) = \mathfrak{u}}$ *(occurrence analysis)*

$$\begin{aligned}
\mathrm{occ}_x(x) &= \{1\} \\
\mathrm{occ}_x(y) &= \{0\} && (x \neq y) \\
\mathrm{occ}_x(\lambda y.\, e) &= \mathrm{occ}_x(e) \\
\mathrm{occ}_x(e_1\, e_2) &= \mathrm{occ}_x(e_1) \hat{+} \mathrm{occ}_x(e_2) \\
\mathrm{occ}_x(\mathsf{let}\ y = e_1\ \mathsf{in}\ e_2) &= \mathrm{occ}_x(e_1) \hat{+} \mathrm{occ}_x(e_2) \\
\mathrm{occ}_x(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3) &= \mathrm{occ}_x(e_1) \hat{+} (\mathrm{occ}_x(e_2) \cup \mathrm{occ}_x(e_3))
\end{aligned}$$

$\boxed{\langle n \rangle = \mathfrak{q}}$, $\boxed{\langle \mathfrak{u} \rangle = \mathfrak{q}}$ *(qualifiers of usage counts)*

$$\langle 0 \rangle = \mathsf{A} \qquad \langle 1 \rangle = \mathsf{L} \qquad \langle \mathsf{many} \rangle = \mathsf{R} \qquad \langle \mathfrak{u} \rangle = \bigsqcap_{n \in \mathfrak{u}} \langle n \rangle$$

Figure 6.8: Occurrence analysis for URAL types, with additive *if* expression

$\boxed{C;\Gamma \vdash e : \tau}$ *(expression typing)*

$$\text{SHMX-VAR}\quad \frac{x{:}\forall\overline{\alpha_i}[D].\,\tau \in \Gamma \qquad C \Vdash \overline{\{\tau_i/\alpha_i\}D}}{C;\Gamma \vdash x : \overline{\{\tau_i/\alpha_i\}}\tau}$$

$$\text{SHMX-ABS}\quad \frac{C;\Gamma, x{:}\tau' \vdash e : \tau \qquad C \Vdash \langle\tau'\rangle \sqsubseteq \langle \mathrm{occ}_x(e)\rangle}{C;\Gamma \vdash \lambda x.e : \tau' \xrightarrow{\langle\Gamma|_{\mathrm{FV}(\lambda x.e)}\rangle}\!\circ\, \tau}$$

$$\text{SHMX-APP}\quad \frac{C;\Gamma \vdash e_1 : \tau_1 \qquad C;\Gamma \vdash e_2 : \tau_2 \qquad C \Vdash \tau_1 \le \tau_2 \xrightarrow{\top}\!\circ\, \tau}{C;\Gamma \vdash e_1\, e_2 : \tau}$$

$$\text{SHMX-LET}\quad \frac{\begin{array}{l} C \wedge D;\Gamma \vdash e_1 : \tau' \qquad C;\Gamma, x{:}\forall\overline{\alpha_i}[D].\,\tau' \vdash e_2 : \tau \\ C \Vdash \langle\tau'\rangle \sqsubseteq \langle \mathrm{occ}_x(e_2)\rangle \qquad \overline{\alpha_i} \notin \mathrm{FTV}(C) \cup \mathrm{FTV}(\Gamma) \end{array}}{C \wedge \exists\overline{\alpha_i}.D;\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau}$$

Figure 6.9: Substructural HM(X) (syntax directed)

that is, the qualifier is the least upper bound of the types of the free variables of the abstraction expression. In rule SHMX-APP, the type of the operator must be a subtype of a function type whose qualifier is the top of the qualifier lattice, because application expressions do not impose any constraint on the qualifier of the function to be applied.

Substructural HM(X) allows arbitrary constraints, including subtyping constraints and subqualifier constraints, in type schemes. In Alms, I chose not to have such constraints in type schemes, because the constraints can grow large and be difficult to read. Instead, Alms limits the constraints that appear in type schemes, as described in the next section.

6.1.3 Core Alms

The type system of Core Alms is derived from Substructural HM(X) by placing a syntactic restriction on the constraints that may appear in type schemes. Internally, Core Alms uses the same language of constraints as Substructural HM(X). However, in a type scheme that binds type variables $\overline{\alpha_i}$, the constraint is restricted to be a conjunction of subqualifier constraints of the form $\langle\alpha_i\rangle \sqsubseteq \mathrm{q}$,

$\boxed{C;\Gamma \vdash e : \tau}$ *(expression typing)*

$$\text{CA-VAR}\quad \frac{x{:}\forall\overline{\alpha_i{:}\mathrm{q}_i}.\,\tau \in \Gamma \qquad C \Vdash \overline{\langle \tau_i \rangle} \sqsubseteq \mathrm{q}_i}{C;\Gamma \vdash x : \overline{\{\tau_i/\alpha_i\}}\tau}$$

$$\text{CA-ABS}\quad \frac{C;\Gamma, x{:}\tau' \vdash e : \tau \qquad C \Vdash \langle \tau' \rangle \sqsubseteq \langle \mathrm{occ}_x(e) \rangle}{C;\Gamma \vdash \lambda x.\, e : \tau' \xrightarrow{\langle \Gamma |_{\mathrm{FV}(\lambda x.e)} \rangle}_{\circ} \tau}$$

$$\text{CA-APP}\quad \frac{C;\Gamma \vdash e_1 : \tau_1 \qquad C;\Gamma \vdash e_2 : \tau_2 \qquad C \Vdash \tau_1 \leq \tau_2 \xrightarrow{\mathsf{A}}_{\circ} \tau}{C;\Gamma \vdash e_1\, e_2 : \tau}$$

$$\text{CA-LET}\quad \frac{\begin{array}{c} C \wedge \overline{\langle \alpha_i \rangle \sqsubseteq \mathrm{q}_i}; \Gamma \vdash e_1 : \tau' \qquad C;\Gamma, x{:}\forall\overline{\alpha_i{:}\mathrm{q}_i}.\,\tau' \vdash e_2 : \tau \\ C \Vdash \langle \tau' \rangle \sqsubseteq \langle \mathrm{occ}_x(e_2) \rangle \qquad \overline{\alpha_i} \notin \mathrm{FTV}(C) \cup \mathrm{FTV}(\Gamma) \end{array}}{C;\Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau}$$

Figure 6.10: Core Alms (syntax directed)

where a universally quantified type variable is bounded above by a constant qualifier. In particular, Core Alms type schemes may be read as sugar for restricted Substructural HM(X) type schemes:

$$\forall\overline{\alpha_i{:}\mathrm{q}_i}.\,\tau \triangleq \forall\overline{\alpha_i}[\overline{\langle \alpha_i \rangle \sqsubseteq \mathrm{q}_i}].\,\tau.$$

Then the type rules for Core Alms (figure 6.10) are merely the rules for Substructural HM(X), with type schemes restricted as described. (Note that $\exists\overline{\alpha_i}.\,\overline{\langle \alpha_i \rangle \sqsubseteq \mathrm{q}_i}$ is not required in the conclusion of rule CA-LET because constraints of that form are always satisfiable.)

The restriction of external constraints to constant upper bounds on universally quantified type variables significantly simplifies the type schemes that are presented to users, but there is a downside: Unlike Substructural HM(X), Core Alms does not enjoy principal type schemes. This is because the principal constraint generated by checking a *let*-bound expression is not always of the restricted form that may be included in a type scheme. Instead, an algorithm for Core Alms type inference must sometimes guess when to unify a type variable with a type, in order to get constraints in the right form.

$\boxed{\Gamma \rhd e : \tau \,;\theta \,;C}$ *(expression typing)*

CA-VAR-ALG
$$\frac{x{:}\forall\overline{\alpha_i{:}\mathrm{q}_i}.\tau \in \Gamma \qquad \overline{\alpha_i} \text{ fresh}}{\Gamma \rhd x : \tau \,;\cdot\,;\overline{\langle\alpha_i\rangle \sqsubseteq \mathrm{q}_i}}$$

CA-ABS-ALG
$$\frac{\Gamma, x{:}\alpha \rhd e : \tau \,;\theta \,;C \qquad \alpha \text{ fresh}}{\Gamma \rhd \lambda x.\, e : \theta\alpha \xrightarrow{\langle\Gamma|_{\mathrm{FV}(\lambda x.\, e)}\rangle}_{\circ} \tau \,;\theta \,;C \wedge \langle\theta\alpha\rangle \sqsubseteq \langle \mathrm{occ}_x(e)\rangle}$$

CA-APP-ALG
$$\frac{\Gamma \rhd e_1 : \tau_1 \,;\theta_1 \,;C_1 \qquad \theta_1\Gamma \rhd e_2 : \tau_2 \,;\theta_2 \,;C_2 \qquad \alpha \text{ fresh}}{\Gamma \rhd e_1\, e_2 : \alpha \,;\theta_2 \circ \theta_1 \,;\theta_2 C_1 \wedge C_2 \wedge \theta_2\tau_1 \leq \tau_2 \xrightarrow{\mathsf{A}}_{\circ} \alpha}$$

CA-LET-ALG
$$\frac{\begin{array}{c}\Gamma \rhd e_1 : \tau_1 \,;\theta_1 \,;C_1 \\ \mathrm{gen}(C_1 \wedge \langle\tau_1\rangle \sqsubseteq \langle \mathrm{occ}_x(e_2)\rangle, \Gamma, \tau_1) \rightsquigarrow (\overline{\langle\alpha_i\rangle \sqsubseteq \mathrm{q}_i}, \theta_1', C_1') \\ \theta_1'\theta_1\Gamma, x{:}\forall\overline{\alpha_i{:}\mathrm{q}_i}.\theta_1'\tau_1 \rhd e_2 : \tau_2 \,;\theta_2 \,;C_2\end{array}}{\Gamma \rhd \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2 \,;\theta_2 \circ \theta_1' \circ \theta_1 \,;\theta_2 C_1' \wedge C_2}$$

Figure 6.11: Type inference algorithm for Core Alms

6.2 A Type Inference Algorithm

The type inference algorithm for Core Alms captures the essence of the algorithm used in the Alms implementation. While it does not infer principal type schemes—since (Core) Alms does not have principal type schemes—the algorithm works well in practice. By delaying constraint solving until generalization time, it types more expressions than an algorithm that greedily unifies subtyping constraints. Furthermore, compared to a hypothetical ML-like algorithm that uses type equality rather than subtyping (and which has principal type schemes), the types that my algorithm infers are, I conjecture, as good or better. The examples in chapters 3 and 4, most of which do not rely on explicit type annotations, support this conjecture.

Figure 6.11 defines the type inference algorithm. Judgment $\Gamma \rhd e : \tau \,;\theta \,;C$ has two input parameters, an environment Γ and an expression e; its three output parameters are a type τ, an idempotent substitution θ mapping type variables to types, and a constraint C.

The first three rules are straightforward: Rule CA-VAR-ALG instantiates

a type scheme with fresh type variables and returns a constraint that bounds each of them by their qualifier bounds in the type scheme. Rule CA-ABS-ALG binds the formal argument of a λ abstraction to a fresh type variable α, infers a type for the body, and bounds the type of x in the resulting constraint based on its occurrences in body expression e. Rule CA-APP-ALG infers types for the operator and operand in an application expression, threading substitutions in the standard way, and adds a subtyping constraint to relate the types of the two subexpressions, as in rule CA-APP.

Rule CA-LET-ALG delegates to an as-yet-unspecified constraint-solving and generalization procedure. First, it infers a type for expression e_1, yielding substitution θ_1 and constraint C_1. It then invokes the generalization procedure, giving it a constraint (including the constraint for e_1 and an upper bound for the qualifier of the type of e_1), the current environment (Γ), and the type to generalize (τ_1, henceforth the *generalizee*). The generalization procedure returns a constraint consisting of qualifier bounds for some type variables to generalize, a substitution recording any unification done during constraint solving, and a remaining constraint C_1'. Finally, the rule uses the type variable bounds computed by the generalization procedure to build a type scheme for variable x for checking body expression e_2.

6.2.1 Generalization

The core of the inference algorithm is the constraint-solving and generalization procedure. The actual generalization procedure used in Alms is complicated and relies heavily on heuristics; I do not attempt to formalize it completely, but merely sketch it in the remainder of this chapter.

The first step to understanding generalization is to specify soundness criteria that ensure that the algorithm implements the non-algorithmic type system. Given input constraint C, environment Γ, and generalizee τ, if generalization succeeds then the procedure produces a restricted constraint of the form $\overline{\langle\alpha_i\rangle \sqsubseteq \mathrm{q}_i}$, a substitution θ, and a constraint D:

$$\mathrm{gen}(C, \Gamma, \tau) \rightsquigarrow (\overline{\langle\alpha_i\rangle \sqsubseteq \mathrm{q}_i}, \theta, D)$$

$\boxed{\Gamma;\tau \vdash C_1 \overset{\theta}{\mapsto} C_2}$ *(reflexive, transitive constraint rewriting)*

$$\text{CR-Refl}\quad \Gamma;\tau \vdash C \overset{\cdot}{\mapsto} C \qquad \text{CR-Trans}\quad \frac{\Gamma;\tau \vdash C \overset{\theta}{\mapsto} C' \qquad \theta\Gamma;\theta\tau \vdash C' \overset{\theta'}{\mapsto} C''}{\Gamma;\tau \vdash C \overset{\theta'\circ\theta}{\longmapsto} C'}$$

$$\text{CR-Cxt}\quad \frac{\mathscr{C};\Gamma;\tau \vdash C \overset{\theta}{\mapsto} C'}{\Gamma;\tau \vdash \mathscr{C}[C] \overset{\theta}{\mapsto} \theta(\mathscr{C})[C']}$$

where $\mathscr{C} ::= [] \mid \mathscr{C} \wedge C \mid C \wedge \mathscr{C} \mid \exists\alpha.\mathscr{C}$

$\boxed{\text{gen}(C,\Gamma,\tau) \rightsquigarrow (\overline{\langle\alpha_i\rangle \sqsubseteq \mathfrak{q}_i}, \theta, C')}$ *(generalization procedure)*

$$\text{Gen}\quad \frac{\Gamma;\tau \vdash C \overset{\theta}{\mapsto} C' \wedge \overline{\langle\alpha_i\rangle \sqsubseteq \mathfrak{q}_i} \qquad \overline{\alpha_i} \notin \text{FTV}(C') \cup \text{FTV}(\theta\Gamma) \qquad \textit{unspecified heuristic stop condition}}{\text{gen}(C,\Gamma,\tau) \rightsquigarrow (\overline{\langle\alpha_i\rangle \sqsubseteq \mathfrak{q}_i}, \theta, C')}$$

Figure 6.12: Constraint rewriting and generalization

Generalization is sound with respect to the non-algorithmic Core Alms type system in figure 6.10 when

1. $\overline{\alpha_i} \notin \text{FTV}(D) \cup \text{FTV}(\theta\Gamma)$ and
2. $D \wedge \overline{\langle\alpha_i\rangle \sqsubseteq \mathfrak{q}_i} \Vdash \theta C$.

The generalization procedure is partially specified by a rewriting system for solving constraints. Judgment $\Gamma;\tau \vdash C_1 \overset{\theta}{\mapsto} C_2$ means that constraint C_1 rewrites to C_2, generating substitution θ, in the context of some environment and generalizee type. This relation is reflexive and transitive, and is defined in terms of a single-step rewriting relation $\mathscr{C};\Gamma;\tau \vdash C_1 \overset{\theta}{\mapsto} C_2$, which rewrites under a constraint rewriting context $\mathscr{C}$ (figure 6.12). The generalization procedure is defined in terms of constraint solving with an additional, unspecified stop condition that uses several heuristics; for example, we would like

to generalize all the type variables in $\mathrm{FTV}(\theta\tau) \setminus \mathrm{FTV}(\theta\Gamma)$, but some may be constrained in a way such that they are better left ungeneralized.

Does the definition of generalization in figure 6.12 satisfy the soundness criteria for generalization? Of the two criteria that I listed, the first appears as a premise to rule GEN, so it is satisfied. The second criterion must be preserved by each rewrite step: For each rule with conclusion $\mathscr{C};\Gamma;\tau \vdash C \overset{\theta}{\mapsto} C'$, we require that $\mathscr{C}[C'] \Vdash \theta(\mathscr{C})[C]$. If this is the case, then the second criterion is satisfied, by induction on the length of the rewriting sequence.

Constraint solving proceeds in two main phases. First, subtyping constraints are simplified and eliminated in order to make some type variables available for generalization. Second, subqualifier constraints are simplified in order to produce bounds for the type variables to be generalized. I describe these two phases in the next two sections.

6.3 Solving Subtype Constraints

Heuristics play a significant role in the implementation of constraint solving in Alms. As described here, constraint rewriting is non-deterministic, but in practice, the heuristics make it deterministic and terminating. One heuristic distinguishes "lossy" rules, which may guess to extend the substitution, from "non-lossy" rules, which do not guess and thus do not lose any generality. The constraint-solving algorithm prefers non-lossy rules to lossy rules whenever possible.

The first phase of constraint solving deals with subtype constraints, which are solved in five subphases:

1. *Decomposition* turns subtyping constraints into a graph whose nodes are type variables and whose edges represent subtyping between those type variables.

2. *Reduction* shrinks the subtyping graph by collapsing strongly connected components and coalescing some type variables based on their polarities.

$\boxed{\tau \leq^{\mathfrak{v}} \tau' = C}$ *(polarized subtyping constraints)*

$$\tau \leq^{+} \tau' \triangleq \tau \leq \tau' \qquad \tau \leq^{-} \tau' \triangleq \tau' \leq \tau$$
$$\tau \leq^{\pm} \tau' \triangleq \tau \leq \tau' \wedge \tau' \leq \tau \qquad \tau \leq^{\phi} \tau' \triangleq \top$$

$\boxed{\mathscr{C}; \Gamma; \tau \vdash C_1 \overset{\theta}{\mapsto} C_2}$ *(constraint rewriting)*

CT-DECOMPOSE

$$\frac{\mathrm{kind}(\chi) = \Pi(\alpha_1^{\mathfrak{v}_1}, \ldots, \alpha_k^{\mathfrak{v}_k}).\xi}{\mathscr{C}; \Gamma; \tau \vdash \chi\,\tau_1 \ldots \tau_k \leq \chi\,\tau'_1 \ldots \tau'_k \overset{\cdot}{\mapsto} \tau_1 \leq^{\mathfrak{v}_1} \tau'_1 \wedge \ldots \wedge \tau_k \leq^{\mathfrak{v}_k} \tau'_k}$$

CT-EXPAND

$$\frac{\overline{\beta_i} \notin \mathrm{FTV}(\mathscr{C}, \Gamma, \tau, \alpha, \overline{\tau_i}) \qquad \textit{occurs check}}{\mathscr{C}; \Gamma; \tau \vdash \alpha \leq^{\mathfrak{v}} \chi\,\overline{\tau_i} \overset{\{\chi\,\overline{\beta_i}/\alpha\}}{\longmapsto} \chi\,\overline{\beta_i} \leq^{\mathfrak{v}} \chi\,\overline{\tau_i}}$$

Figure 6.13: Some decomposition rules (non-lossy)

3. *Existential elimination* gets rid of graph nodes that do not appear in the environment or generalizee (which are thus existentially quantified in the constraint), and turns some subtype constraints into subqualifier constraints.

4. *Guessing* lossily unifies some type variables where a heuristic determines that this is beneficial.

5. *Candidate selection* determines which type variables to attempt to generalize.

6.3.1 Decomposition

Two non-lossy rewriting rules for the decomposition subphase appear in figure 6.13. This subphase repeatedly applies rules CT-DECOMPOSE and CT-EXPAND as much as possible, along with analogous rules for function types, until the subtyping constraint is decomposed into a subtyping graph on type variables. The implementation actually applies the decomposition

$\boxed{\mathrm{var}_\alpha(\xi) = \mathfrak{v}}$, $\boxed{\mathrm{var}_\alpha(\tau) = \mathfrak{v}}$, $\boxed{\mathrm{var}_\alpha(C) = \mathfrak{v}}$ *(variance)*

$$\mathrm{var}_\alpha(\xi) = \text{if } \alpha \in \mathrm{FTV}(\xi) \text{ then } + \text{ else } \phi$$

$$\mathrm{var}_\alpha(\beta) = \text{if } \alpha = \beta \text{ then } + \text{ else } \phi$$
$$\mathrm{var}_\alpha(\tau_1 \xrightarrow{\xi} \tau_2) = -\mathrm{var}_\alpha(\tau_1) \sqcup \mathrm{var}_\alpha(\xi) \sqcup \mathrm{var}_\alpha(\tau_2)$$
$$\mathrm{var}_\alpha(\chi\,\overline{\tau_i}) = \bigsqcup_i \mathfrak{v}_i \cdot \mathrm{var}_\alpha(\tau_i) \quad \text{where } \mathrm{kind}(\chi) = \Pi(\overline{\alpha_i^{\mathfrak{v}_i}}).\,\xi$$

$$\mathrm{var}_\alpha(\top) = \phi$$
$$\mathrm{var}_\alpha(\tau_1 \le \tau_2) = \mathrm{var}_\alpha(\tau_1) \sqcup -\mathrm{var}_\alpha(\tau_2)$$
$$\mathrm{var}_\alpha(\xi_1 \sqsubseteq \xi_2) = \mathrm{var}_\alpha(\xi_1) \sqcup -\mathrm{var}_\alpha(\xi_2)$$
$$\mathrm{var}_\alpha(C \wedge D) = \mathrm{var}_\alpha(C) \sqcup \mathrm{var}_\alpha(D)$$
$$\mathrm{var}_\alpha(\exists \beta.\, C) = \mathrm{var}_\alpha(C)$$

Figure 6.14: Variance of type variables

rules eagerly, so that internally the subtyping portion of a constraint uses an efficient graph representation.

Rule CT-DECOMPOSE implements structural subtyping by decomposing a subtyping constraint between two types with the same constructor χ into constraints on all the type parameters. The parameters are related in directions that depend on the variance of the type constructor. Rule CT-EXPAND rewrites a subtyping constraint between a type variable α and an applied type constructor χ by substituting χ applied to fresh type variables for α. The usual occurs check that $\alpha \notin \mathrm{FTV}(\chi\,\overline{\tau_i})$ is insufficient here because subtyping allows cycles between multiple atomic constraints. Consider, for example, repeatedly applying rules CT-DECOMPOSE and CT-EXPAND to constraint $\chi\,\alpha \le \beta \wedge \chi\,\beta \le \alpha$—the process diverges without ever failing the naïve occurs check. Instead, the implementation maintains equivalence classes of "same-sized" type variables and performs the occurs check on those.

$\boxed{\mathscr{C};\Gamma;\tau \vdash C_1 \overset{\theta}{\mapsto} C_2}$ *(constraint rewriting)*

CT-CYCLE

$$\mathscr{C};\Gamma;\tau \vdash \alpha_0 \le \alpha_1 \wedge \alpha_1 \le \alpha_2 \wedge \ldots \wedge \alpha_k \le \alpha_0 \overset{\overline{\{\alpha_0/\alpha_i\}}}{\longmapsto} \top$$

CT-POSPRED

$$\frac{\mathrm{var}_\beta(\tau) = + \qquad \mathrm{var}_\beta(\mathscr{C}) \sqsubseteq + \qquad \beta \notin \mathrm{FTV}(\Gamma)}{\mathscr{C};\Gamma;\tau \vdash \alpha \le \beta \overset{\{\alpha/\beta\}}{\longmapsto} \top}$$

CT-NEGSUCC

$$\frac{\mathrm{var}_\beta(\tau) = - \qquad \mathrm{var}_\beta(\mathscr{C}) \sqsubseteq - \qquad \beta \notin \mathrm{FTV}(\Gamma)}{\mathscr{C};\Gamma;\tau \vdash \beta \le \alpha \overset{\{\alpha/\beta\}}{\longmapsto} \top}$$

CT-POSSIB

$$\frac{\overline{\mathrm{var}_{\beta_j}(\tau) = +} \qquad \overline{\mathrm{var}_{\beta_j}(\mathscr{C}) \sqsubseteq +} \qquad \overline{\beta_j \notin \mathrm{FTV}(\Gamma)}}{\mathscr{C};\Gamma;\tau \vdash \overline{\overline{\alpha_i \le \beta_j}} \overset{\overline{\{\beta_1/\beta_j\}}}{\longmapsto} \overline{\alpha_i \le \beta_1}}$$

CT-NEGSIB

$$\frac{\overline{\mathrm{var}_{\beta_j}(\tau) = -} \qquad \overline{\mathrm{var}_{\beta_j}(\mathscr{C}) \sqsubseteq -} \qquad \overline{\beta_j \notin \mathrm{FTV}(\Gamma)}}{\mathscr{C};\Gamma;\tau \vdash \overline{\overline{\beta_j \le \alpha_i}} \overset{\overline{\{\beta_1/\beta_j\}}}{\longmapsto} \overline{\beta_1 \le \alpha_i}}$$

Figure 6.15: Some constraint reduction rules (non-lossy)

6.3.2 Reduction

The reduction subphase attempts to reduce the size of the subtyping graph built by the decomposition subphase. There are two ways that it does this: coalescing strongly connected components, and coalescing similarly situated nodes of a given variance in the generalizee. The variance of type variables with respect to qualifiers, types, and constraints is defined in figure 6.14.

Rule CT-CYCLE (figure 6.15) identifies a cycle in the subtyping graph and unifies all the type variables in the cycle. Applied repeatedly, this rule collapses each strongly connected component of the graph to a single type variable.

The remaining rules in figure 6.15 implement polarized constraint reduction, as described by Simonet (2003). Rule CT-POSPRED coalesces a type variable β with its unique predecessor α if (1) β appears covariantly in the

$\boxed{\mathscr{C};\Gamma;\tau \vdash C_1 \overset{\theta}{\mapsto} C_2}$ *(constraint rewriting)*

$$\text{CT-FRESH}\ \frac{\alpha \notin \mathrm{FTV}(\mathscr{C},\Gamma,\tau)}{\mathscr{C};\Gamma;\tau \vdash C \dot{\mapsto} \exists\alpha.C}$$

Figure 6.16: Existential introduction (non-lossy)

generalizee type, (2) β has no other predecessors in the constraint context, and (3) β does not appear in the environment; dually, rule CT-NEGSUCC coalesces a contravariant type variable with a unique successor. Rule CT-POSSIB coalesces several covariant type variables β_j if each has exactly the same set of predecessors α_i; rule CT-NEGSIB works dually for negative type variables with the same successors.

6.3.3 Existential Elimination

Type variables that appear only in the constraint, not in the environment nor the generalizee, are considered to be existentially quantified in the constraint. The existential elimination subphase attempts to remove these type variables from the subtyping constraint graph, potentially adding new subqualifier constraints for them if necessary. In the implementation, existential type variables are tracked by mutable state in unification variables, rather than having binders in the representation of constraints. For the purposes of specifying the constraint rewriting system, however, we add a rule that explicitly adds existential quantifiers to the constraint (figure 6.16).

Unlike the previous two subphases, existential elimination involves a subtle interaction between subtype and subqualifier constraints. To clarify the issues, I abstract from the concrete constraint-solving problem considered thus far.

The setup. Let there be:

- a partially ordered set $(T, \leq)$,

- a lattice $(Q, \sqsubseteq)$,
- and a set map $\langle\cdot\rangle : T \to Q$.

Let $\sim$ be the smallest equivalence relation containing $\leq$, and define the equivalence classes $[t] = \{t' \in T \mid t' \sim t\}$. For each $t \in T$, we call $[t] \in T/\sim$ the *shape* of t.

We require the shapes of T to have two additional properties:

- Each shape $([t], \leq)$ must be a lattice; further, each shape lattice must be *modular*: for all $t_1, t_2, t_3 \in [t]$, if $t_1 \leq t_3$ then $t_1 \sqcup (t_2 \sqcap t_3) = (t_1 \sqcup t_2) \sqcap t_3$.
- The map $\langle\cdot\rangle$ is not necessarily monotone, but when restricted to any shape $[t]$, then $\langle\cdot\rangle|_{[t]} : [t] \to Q$ is either monotone or antitone.

Constraints. Next, we define a language of constraints over elements of T and Q,

$$C, D ::= \top \mid C \wedge D \mid x_1 \leq x_2 \mid q_1 \sqsubseteq q_2 \mid \exists x.C,$$

where $q \in Q$ and variables x stand for elements of T. (Constraints never involve concrete elements of T.)

The machinery of constraints is essentially the same as for Substructural HM(X) constraints. Equivalence is defined up to the commutative monoid on $\wedge$ with identity $\top$, with scope extrusion and α conversion. The meaning of constraints is given by a relation $\vartheta \models C$, defined as for Substructural HM(X) in figure 6.3, where ϑ is a substitution mapping variables to elements of T. Single-step constraint rewriting is defined by a relation $C \overset{\vartheta}{\mapsto} D$, which is lifted to rewrite under constraint contexts. (Once existential quantifiers have been suitably introduced into a constraint by rule CT-FRESH, we no longer need to see the context, environment, or generalizee.)

Constraint rewriting. As before, a rewriting rule $C \overset{\vartheta}{\mapsto} D$ is sound if $D \Vdash \vartheta C$. However, the kind of simplification that I am concerned with here is existential elimination of the form $\exists y.C \overset{\cdot}{\mapsto} D$. To show that such a rule is sound, it suffices to show that for all ϑ such that $\vartheta \Vdash D$, we can find a $t_y \in T$ such that

$\vartheta \circ \{t_y/y\} \models C$. (This follows directly from the definition of constraint entailment and the meaning of existentially quantified constraints.)

We consider first a simple rewrite rule for eliminating existentials.

THEOREM 6.1 (Existential elimination (i))**.**

Rewrite rule $\exists y.x \leq y \wedge y \leq z \dot{\mapsto} x \leq z$ *is sound.*

Proof. Given that $\vartheta \models x \leq z$, we know that $\vartheta x \leq \vartheta z$. Let $t_y = \vartheta x$. Then,

$$\frac{\dfrac{\vartheta x \leq \vartheta x}{\vartheta \circ \{\vartheta x/y\} \models x \leq y\,.} \qquad \dfrac{\vartheta x \leq \vartheta z}{\vartheta \circ \{\vartheta x/y\} \models y \leq z\,.}}{\vartheta \circ \{\vartheta x/y\} \models x \leq y \wedge y \leq z}.$$

□

We can also admit a more general version of the previous rule:

THEOREM 6.2 (Existential elimination (ii))**.**

Rewrite rule

$$\exists y. \left(\bigwedge_i x_i \leq y\right) \wedge \left(\bigwedge_j y \leq z_j\right) \dot{\mapsto} \bigwedge_i \bigwedge_j x_i \leq z_j$$

is sound.

Proof. Given a substitution ϑ that satisfies the result of the rule, we must find a $t_y \in T$ such that $\vartheta x_i \leq t_y$ for all i and $t_y \leq \vartheta z_j$ for all j.

We know that $\vartheta x_i \leq \vartheta z_j$ for all i and j. This means that they all have the same shape: for all i and j, $[\vartheta x_i] = [\vartheta z_j]$. Thus, all the ϑx_i and ϑz_j are in a lattice $([\vartheta x_i], \leq)$, which has finite joins. Let $t_y = \bigsqcup_i \vartheta x_i$. Since each ϑz_j is an upper bound for all ϑx_i, and since t_y is the least upper bound of the latter, by the Riesz Interpolation Property, $t_y \leq \vartheta z_j$ for all j. □

The problem. When simplifying constraints, the goal is to get rid of $\leq$ constraints by taking advantage of existentially quantified variables; at this point, we do not mind if the number of $\sqsubseteq$ constraints increases. Now consider a constraints of the form

$$\exists y.x \leq y \wedge y \leq z \wedge C,$$

where C contains no $\leq$ constraints. We would like to eliminate y as in the above rules, but we cannot because C may contain some $\sqsubseteq$ constraints on $\langle y \rangle$. For example, consider this hypothetical constraint rewriting rule:

$$\frac{y \notin \mathrm{FV}(q_1, q_2)}{\exists y.\, x \leq y \wedge y \leq z \wedge q_1 \sqsubseteq \langle y \rangle \wedge \langle y \rangle \sqsubseteq q_2 \;\dot{\mapsto}\; x \leq z \wedge \exists y.\, q_1 \sqsubseteq \langle y \rangle \wedge \langle y \rangle \sqsubseteq q_2}.$$

The rule is unsound, because the eventual choice for y satisfying $q_1 \sqsubseteq \langle y \rangle$ and $\langle y \rangle \sqsubseteq q_2$ may not be between x and z. In fact, there is no way to reduce the number of $\leq$ constraints by adding $\sqsubseteq$ constraints, because we do not know enough about $\langle \cdot \rangle$.

An idea. A potential solution is to strengthen the properties of $\langle \cdot \rangle$, as follows. For each shape $[t]$ of T, we now require that

1. the lattice map $\langle \cdot \rangle|_{[t]} : [t] \to Q$ is continuous, and
2. the image $\langle [t] \rangle$ is either a single point in Q or all of Q.

We then need the following lemma about modular lattices:

LEMMA 6.3 (Intermediate value)**.**

Let $(A, \sqcup, \sqcap)$ be a modular lattice, $(B, \sqcup, \sqcap)$ be a lattice, and $g : A \to B$ be surjective and continuous. Then for all $a_1, a_2 \in A$ such that $a_1 \sqsubseteq a_2$, and for all $b \in B$ such that $g(a_1) \sqsubseteq b \sqsubseteq g(a_2)$, there exists some $a' \in A$ such that $g(a') = b$ and $a_1 \sqsubseteq a' \sqsubseteq a_2$.

Proof. Because map g is surjective, we know that b is in the image of g, so we can pull back b to its preimage $g^{-1}(b) \subseteq A$. Let $a' = a_1 \sqcup (a_b \sqcap a_2)$, where a_b is any element of $g^{-1}(b)$. Then:

$$\begin{aligned}
g(a') &= g(a_1 \sqcup (a_b \sqcap a_2)) && \text{definition of } a' \\
&= g(a_1) \sqcup (g(a_b) \sqcap g(a_2)) && \text{continuity of } g \\
&= g(a_1) \sqcup (b \sqcap g(a_2)) && a_b \in g^{-1}(b) \\
&= g(a_1) \sqcup b && b \sqsubseteq g(a_2) \\
&= b && g(a_1) \sqsubseteq b.
\end{aligned}$$

Because $a' = a_1 \sqcup (a_b \sqcap a_2)$, clearly $a_1 \sqsubseteq a'$. By modularity of A, and because $a_1 \sqsubseteq a_2$, we know that $a_1 \sqcup (a_b \sqcap a_2) = (a_1 \sqcup a_b) \sqcap a_2$, and furthermore, $(a_1 \sqcup a_b) \sqcap a_2 \sqsubseteq a_2$. Thus, we have that $a_1 \sqsubseteq a' \sqsubseteq a_2$. □

COROLLARY 6.4 (Intermediate value)**.**

If $t_1 \leq t_2$, then for all q such that $\langle t_1 \rangle \sqsubseteq q \sqsubseteq \langle t_2 \rangle$, there exists some t' such that $\langle t' \rangle = q$ and $t_1 \leq t' \leq t_2$.

Proof. Since $t_1 \leq t_2$, we know that t_1 and t_2 are in the same shape $[t_1]$, which is a modular lattice. According to the properties specified for $\langle\cdot\rangle$, the image $\langle [t_1] \rangle$ is either a single point or all of Q. If the image $\langle [t_1] \rangle$ is a single point, then $\langle t_1 \rangle = q = \langle t_2 \rangle$, so let $t' = t_1$. Otherwise, $\langle\cdot\rangle|_{[t_1]} : [t_1] \to Q$ is surjective, so the corollary holds by lemma 6.3. □

Now we can formulate a sound rewriting rule to replace the unsound rule proposed above.

THEOREM 6.5 (Existential elimination (iii))**.**

Rewrite rule

$$\frac{y, y' \notin \mathrm{FV}(q_1, q_2)}{\begin{array}{l}\exists y.\, x \leq y \wedge y \leq z \wedge q_1 \sqsubseteq \langle y \rangle \wedge \langle y \rangle \sqsubseteq q_2 \mathrel{\dot{\mapsto}} \\ \qquad x \leq z \wedge \exists y'.\, \langle x \rangle \sqsubseteq \langle y' \rangle \wedge \langle y' \rangle \sqsubseteq \langle z \rangle \wedge q_1 \sqsubseteq \langle y' \rangle \wedge \langle y' \rangle \sqsubseteq q_2\end{array}}$$

is sound.

Proof. If ϑ satisfies the result of the rule, then we know that $\vartheta x \leq \vartheta y$, and furthermore, there must be some $t_{y'}$ such that

$$\dfrac{\dfrac{\vartheta \models \exists y'.\, \langle x \rangle \sqsubseteq \langle y' \rangle \wedge \langle y' \rangle \sqsubseteq \langle z \rangle \wedge q_1 \sqsubseteq \langle y' \rangle \wedge \langle y' \rangle \sqsubseteq q_2}{\vartheta' \models \langle x \rangle \sqsubseteq \langle y' \rangle \wedge \langle y' \rangle \sqsubseteq \langle z \rangle \wedge q_1 \sqsubseteq \langle y' \rangle \wedge \langle y' \rangle \sqsubseteq q_2}}{\begin{array}{cccc} \dfrac{\vartheta' \models \langle x \rangle \sqsubseteq \langle y' \rangle}{\begin{array}{c}\langle \vartheta' x \rangle \sqsubseteq \langle \vartheta' y' \rangle \\ = \quad = \\ \langle \vartheta x \rangle \sqsubseteq \langle t_{y'} \rangle\end{array}} & \dfrac{\vartheta' \models \langle y' \rangle \sqsubseteq \langle z \rangle}{\begin{array}{c}\langle \vartheta' y' \rangle \sqsubseteq \langle \vartheta' z \rangle \\ = \quad = \\ \langle t_{y'} \rangle \sqsubseteq \langle \vartheta z \rangle\end{array}} & \dfrac{\vartheta' \models q_1 \sqsubseteq \langle y' \rangle}{\begin{array}{c}\vartheta' q_1 \sqsubseteq \langle \vartheta' y' \rangle \\ = \quad = \\ \vartheta q_1 \sqsubseteq \langle t_{y'} \rangle\end{array}} & \dfrac{\vartheta' \models \langle y' \rangle \sqsubseteq q_2}{\begin{array}{c}\langle \vartheta' y' \rangle \sqsubseteq \vartheta' q_2 \\ = \quad = \\ \langle t_{y'} \rangle \sqsubseteq \vartheta q_2\end{array}}\end{array}},$$

where $\vartheta' = \vartheta \circ \{t_{y'}/y'\}$. We assume that the rule respects scope, which means that $y, y' \notin \mathrm{FV}(x, z, q_1, q_2)$. Thus we know that $\vartheta\{t/y'\}x = \vartheta\{t/y\}x$ for any t, and likewise for z, q_1, and q_2.

Since $\vartheta x \le \vartheta z$ and $\langle \vartheta x \rangle \sqsubseteq \langle t_{y'} \rangle \sqsubseteq \langle \vartheta z \rangle$, we can apply corollary 6.4 with $t_1 = \vartheta x$, $t_2 = \vartheta z$, and $q = \langle t_{y'} \rangle$; then there is some t_y such that such that $\langle t_y \rangle = \langle t_{y'} \rangle$ and $\vartheta x \le t_y \le \vartheta y$. Let $\vartheta'' = \vartheta \circ \{t_y/y\}$. Note that $\vartheta'' x = \vartheta x$, $\vartheta'' y = \vartheta y$, $\vartheta'' q_1 = \vartheta q_1$, and $\vartheta'' q_2 = \vartheta q_2$ by our freshness assumptions. Then,

$$\dfrac{\dfrac{\dfrac{\vartheta x \le t_y}{\vartheta'' x \le \vartheta'' y}}{\vartheta'' \models x \le y} \quad \dfrac{\dfrac{t_y \le \vartheta z}{\vartheta'' y \le \vartheta'' z}}{\vartheta'' \models y \le z} \quad \dfrac{\dfrac{\vartheta q_1 \sqsubseteq \langle t_{y'} \rangle}{\vartheta'' q_1 \sqsubseteq \langle \vartheta'' y \rangle}}{\vartheta'' \models q_1 \sqsubseteq \langle y \rangle} \quad \dfrac{\dfrac{\langle t_{y'} \rangle \sqsubseteq \vartheta q_2}{\langle \vartheta'' y \rangle \sqsubseteq \vartheta'' q_2}}{\vartheta'' \models \langle y \rangle \sqsubseteq q_2}}{\vartheta'' \models x \le y \wedge y \le z \wedge q_1 \sqsubseteq \langle y \rangle \wedge \langle y \rangle \sqsubseteq q_2}. \qquad \square$$

Existential elimination in Alms. Having solved the problem of existential elimination in the abstract setting, it remains to apply the solution in Alms.

The condition on T, the set of types—in particular, that it partition to a family of modular lattices—holds in the context of atomic subtyping. In such a setting, types are related only if they differ at some leaves (atoms), which means that each shape represents a set of types with the same tree structure but different leaves. Modularity means that we can factor the potential subtyping in a shape into a product of the subtyping of the leaves.

The original condition on $\langle\cdot\rangle$ is also reasonable: It means that for any shape lattice of types, the qualifier property varies either with subtyping or against it. Where $\langle\cdot\rangle$ is monotone, this corresponds to a setting of linear or affine types, where the restrictions on a value may be strengthened by dereliction. Where $\langle\cdot\rangle$ is antitone, this gives uniqueness types, where subsuming a unique type to a non-unique type *increases* what we may do with it. When $\langle\cdot\rangle$ is constant on each shape, we have Wadler's (1991) steadfast types. Allowing $\langle\cdot\rangle$ to be monotone on some shapes and antitone on others would support both affine and uniqueness types in the same language; unfortunately, that property impedes type inference, because it makes the rewrite rule of theorem 6.5 unsound. Instead, we adopt the requirement that $\langle\cdot\rangle$ be continuous.

In this section, I treat $\langle\cdot\rangle$ as a fixed, abstract function with some known properties, but in Alms the map from types to qualifiers is extended each time a

new type is defined. Continuity of $\langle\cdot\rangle$ corresponds to monotonicity of dependent kinds in Alms. Because Alms relies on a generalization of theorem 6.5 for type inference, it requires the kinds of all defined types to be monotone. Assuming that the kinds of all primitive types are monotone, all new concrete types will have monotone kinds, by composition; however, Alms requires a check that abstract type constructors have monotone kinds as well. Monotonicity of kinds appears in the model ${}^{a}\lambda_{ms}$ as well, in lemma 5.4 on page 119.

6.3.4 Guessing and Candidate Selection

The final two subphases of the subtype-solving phase are guessing and candidate selection. Guessing extends the substitution based on a heuristic, with the goal of making more type variables generalizable in the candidate selection subphase.

To see why guessing is useful, consider two constraint-solving problems, where binary type constructor χ is invariant in both parameters:

1. $\mathscr{C}; \bullet; \chi\,\alpha\,\beta \vdash \alpha \le \beta \overset{?}{\mapsto} ?$
2. $\mathscr{C}; y{:}\beta; \chi\,\alpha\,\beta \vdash \alpha \le \beta \overset{?}{\mapsto} ?$

In both problems, because type variables α and β appear invariantly in the generalizee, none of the reduction rules apply and existential elimination is not an option. Furthermore, because both α and β appear in a subtyping constraint, neither is generalizable as the constraint stands. In problem 1, there are essentially two options: either unify α and β, thus making the coalesced type variable generalizable, or leave them ungeneralized. If we were constraint solving using equality, as in ML, then we would have unified them already, so we can safely unify them and do no worse than ML would. In problem 2, unifying the type variables will not let us generalize, because β appears free in the environment; thus, it is better not to unify. The guessing subphase therefore unifies α and β in problem 1 but does not in problem 2:

1. $\mathscr{C}; \bullet; \chi\,\alpha\,\beta \vdash \alpha \le \beta \overset{\{\alpha/\beta\}}{\longmapsto} \top$
2. $\mathscr{C}; y{:}\beta; \chi\,\alpha\,\beta \vdash \alpha \le \beta \overset{\cdot}{\mapsto} \alpha \le \beta$

$\boxed{\mathscr{C};\Gamma;\tau \vdash C_1 \overset{\theta}{\mapsto} C_2}$ *(constraint rewriting)*

$$\text{CT-GUESS}\ \frac{\text{subgraph } \overline{\alpha_i \leq \beta_i} \text{ is connected} \qquad \overline{\alpha_i}, \overline{\beta_i} \notin \text{FTV}(\mathscr{C}|_{\leq}, \Gamma)}{\mathscr{C};\Gamma;\tau \vdash \overline{\alpha_i \leq \beta_i} \xmapsto{\overline{\{\alpha_1/\alpha_i\}} \circ \overline{\{\alpha_1/\beta_i\}}} \top}$$

Figure 6.17: Guessing (lossy)

The two problems illustrate the main rule of the guessing subphase, which appears in figure 6.17. In rule CT-GUESS, $\mathscr{C}|_{\leq}$ stands for the portion of constraint rewriting context $\mathscr{C}$ that involves subtype constraints; or equivalently, $\mathscr{C}|_{\leq}$ is $\mathscr{C}$ with all subqualifier constraints replaced by $\top$. The rule identifies a connected component of the subtype constraint graph all of whose type variables are not free in environment Γ, and coalesces those type variables.

Finally, the candidate selection subphase identifies which type variables should be generalized based on the current constraint, environment, and generalizee:

$$\text{FTV}(\tau) \setminus (\text{FTV}(\Gamma) \cup \text{FTV}(C|_{\leq}))$$

That is, the generalization procedure will attempt to generalize the free type variables of the generalizee type, except for those that cannot be generalized because they appear in the environment or the subtype portion of the constraint. This selection does not happen by way of a rewrite rule, but serves to drive the second constraint solving phase, which attempts to find a constant qualifier bound for each generalization candidate type variable.

6.4 Solving Subqualifier Constraints

The subtype-solving phase completes by selecting a set of type variables to generalize. The goal of second phase, which solves subqualifier constraints, is to upper bound each generalization candidate by a single constant qualifier. That is, given the set of generalization candidates $\overline{\alpha_i} = \text{FTV}(\tau) \setminus (\text{FTV}(\Gamma) \cup$

$\mathrm{FTV}(C|_{\leq})$), the goal is to rewrite to a constraint of the form

$$\overline{\langle \alpha_i \rangle \sqsubseteq \mathsf{q}_i} \wedge D$$

where $\overline{\alpha_i} \notin \mathrm{FTV}(D)$.

In addition to keeping track of generalization candidates, qualifier solving distinguishes type variables that appear only in qualifiers (henceforth "flexible variables") from non-flexible type variables. This is important, because while non-flexible type variables stand for some type, flexible type variables are used only for their qualifier value. For example, consider type $\alpha \xrightarrow{\langle \beta \rangle} \gamma$ where α, β, and γ are unification variables. If we learn that $\langle \beta \rangle \sqsubseteq \mathsf{U}$ then we can safely replace $\langle \beta \rangle$ by U, since type variable β is being used only for its qualifier. By contrast, learning that $\langle \alpha \rangle \sqsubseteq \mathsf{U}$ should not cause α to be unified with U, because there are many types whose qualifier is U and which type is chosen matters. Thus, if we can determine the qualifier of a flexible type variable, then it is safe to substitute for it any type with that qualifier; but for a non-flexible type variable, the qualifier alone is not enough information to choose a type.

Qualifier solving consists of four subphases, each of which act only on the subqualifier portion of the constraint:

1. *Standardization* rewrites the subqualifier portion of the constraint to a standard form, which the other subphases expect. The other phases do not preserve standardization, so this phase is run again whenever necessary to return the constraint to standard form.

2. *Reduction*, like the reduction subphase in the previous section, attempts to use the variances of type variables and their relationships to coalesce them.

3. Usually the first two subphases are sufficient to find bounds for generalization candidates, but if not, the *SAT solving* subphase provides a backstop. It treats the qualifier portion of the constraint as a SAT instance and runs a SAT solver, which ensures that the constraint is satisfiable and yields a substitution.

4. The *bounding* subphase selects constant qualifier bounds for all remaining generalization candidates.

$\boxed{\mathscr{C};\Gamma;\tau \vdash C_1 \overset{\theta}{\mapsto} C_2}$ *(constraint rewriting)*

CQ-BOT
$$\mathscr{C};\Gamma;\tau \vdash \mathsf{U} \sqsubseteq \xi \dot{\mapsto} \top$$

CQ-TOP
$$\mathscr{C};\Gamma;\tau \vdash \xi \sqsubseteq \mathsf{A} \dot{\mapsto} \top$$

CQ-DECOMPOSE
$$\frac{\overline{\beta_i} = \alpha_1,\ldots,\alpha_j \setminus \alpha'_1,\ldots,\alpha'_k}{\mathscr{C};\Gamma;\tau \vdash \langle\alpha_1\rangle \sqcup \ldots \sqcup \langle\alpha_j\rangle \sqsubseteq \langle\alpha'_1\rangle \sqcup \ldots \sqcup \langle\alpha'_k\rangle \dot{\mapsto} \overline{\langle\beta_i\rangle \sqsubseteq \langle\alpha'_1\rangle \sqcup \ldots \sqcup \langle\alpha'_k\rangle}}$$

CQ-FILTERA
$$\frac{\overline{\alpha'_i} = \{\alpha \in \overline{\alpha_i} \mid \alpha \text{ is flexible}\}}{\mathscr{C};\Gamma;\tau \vdash \mathsf{A} \sqsubseteq \overline{\langle\alpha_i\rangle} \dot{\mapsto} \mathsf{A} \sqsubseteq \overline{\langle\alpha'_i\rangle}}$$

CQ-FILTERV
$$\frac{\beta \text{ is not flexible} \qquad \overline{\alpha'_i} = \{\alpha \in \overline{\alpha_i} \mid \alpha \text{ is flexible}\}}{\mathscr{C};\Gamma;\tau \vdash \langle\beta\rangle \sqsubseteq \overline{\langle\alpha_i\rangle} \dot{\mapsto} \langle\beta\rangle \sqsubseteq \overline{\langle\alpha'_i\rangle}}$$

Figure 6.18: Qualifier constraint standardization (lossy and non-lossy)

6.4.1 Standardization

The goal of the standardization phase is to get all atomic subqualifier constraints into one of two standard forms:

- $\mathsf{A} \sqsubseteq \langle\beta_1\rangle \sqcup \ldots \sqcup \langle\beta_k\rangle$ where the $\overline{\beta_i}$ are flexible;
- $\langle\alpha\rangle \sqsubseteq \langle\beta_1\rangle \sqcup \ldots \sqcup \langle\beta_k\rangle$ where the $\overline{\beta_i}$ are flexible if α is not flexible.

The flexibility requirements make sense as follows. Constraints of the first form may be read as disjunctions: at least one of $\beta_1 \ldots \beta_k$ must be affine. Thus, such a constraint may be satisfied by substituting any affine type for any flexible variable in the upper bound, but because non-flexible variables cannot be substituted in the qualifier solving phase, they do not help with solving such a constraint. The rationale behind the second form is similar, except that when α is flexible, the constraint can also be solved by substituting $\langle\beta_1\rangle \sqcup \ldots \sqcup \langle\beta_k\rangle$ for α; thus, non-flexible type variables in the upper bound remain relevant.

The first three standardization rules in figure 6.18 are non-lossy. Rules CQ-BOT and CQ-TOP immediately discharge subqualifier constraints with U as a lower bound or A as an upper bound. Rule CQ-DECOMPOSE breaks down an

$\boxed{\mathscr{C};\Gamma;\tau \vdash C_1 \overset{\theta}{\mapsto} C_2}$ *(constraint rewriting)*

All rules in this figure implicitly have the premise "β is flexible."

$$\text{Sugar: } \beta\,\text{cand} \triangleq \beta \in \text{FTV}(\tau) \setminus (\text{FTV}(\Gamma) \cup \text{FTV}(\mathscr{C}|_{\leq}))$$

$$\text{CQ-ForceU}\quad \mathscr{C};\Gamma;\tau \vdash \langle\beta\rangle \sqsubseteq \mathsf{U} \xrightarrow{\{\mathsf{U}/\beta\}} \top$$

$$\text{CQ-SubstNeg}\quad \frac{\text{var}_\beta(\mathscr{C}) \sqcup \text{var}_\beta(\tau) \sqsubseteq - \qquad \beta\,\text{cand}}{\mathscr{C};\Gamma;\tau \vdash \langle\beta\rangle \sqsubseteq \xi \xrightarrow{\{\xi/\beta\}} \top}$$

$$\text{CQ-SubstNegTop}\quad \frac{\text{var}_\beta(\mathscr{C}) \sqcup \text{var}_\beta(\tau) \sqsubseteq - \qquad \beta\,\text{cand}}{\mathscr{C};\Gamma;\tau \vdash \top \xrightarrow{\{\mathsf{A}/\beta\}} \top}$$

$$\text{CQ-SubstPos}\quad \frac{\text{var}_\beta(\mathscr{C}) \sqcup \text{var}_\beta(\tau) \sqsubseteq + \qquad \beta\,\text{cand}}{\mathscr{C};\Gamma;\tau \vdash \xi_1 \sqsubseteq \langle\beta\rangle \wedge \ldots \wedge \xi_k \sqsubseteq \langle\beta\rangle \xrightarrow{\{\xi_1 \sqcup \ldots \sqcup \xi_k/\beta\}} \top}$$

$$\text{CQ-SubstInv}\quad \frac{\text{var}_\beta(\mathscr{C}) \sqsubseteq + \qquad \text{var}_\beta(\tau) = \pm \qquad \beta' \notin \text{FTV}(\mathscr{C},\Gamma,\tau,\beta,\overline{\xi_i}) \qquad \beta\,\text{cand}}{\mathscr{C};\Gamma;\tau \vdash \xi_1 \sqsubseteq \langle\beta\rangle \wedge \ldots \wedge \xi_k \sqsubseteq \langle\beta\rangle \xrightarrow{\{\langle\beta'\rangle \sqcup \xi_1 \sqcup \ldots \sqcup \xi_k/\beta\}} \top}$$

Figure 6.19: Qualifier constraint reduction (lossy and non-lossy)

inequality on joins of type variables into an inequality for each type variable in the lower bound that does not appear in the upper bound.

The last two standardization rules, CQ-FilterA and CQ-FilterV, are lossy. These rules remove non-flexible type variables from upper bounds, in order to reach the flexibility requirements of the two standard forms. Removing type variables from a join on the right side of an inequality is sound, because it effectively tightens the inequality.

6.4.2 Reduction

The reduction phase substitutes for flexible type variables in order to shrink the subqualifier portion of the constraint (figure 6.19). Rule CQ-ForceU substitutes an unlimited type for flexible type variables bounded above by U. The remaining rules all substitute for a flexible, generalization candidate

type variable β, based on its variance in the rest of the constraint and the generalizee. These follow the usual rules for polarized constraint reduction: contravariant variables are unified with their upper bounds and covariant variables are unified with their lower bounds. Rule CQ-SUBSTNEGTOP is like CQ-SUBSTNEG, except that it assumes an upper bound of A for any type variable that has no other upper bound. Rule CQ-SUBSTINV substitutes for an invariant type variable, using its lower bound joined with a fresh slack variable β'.

6.4.3 SAT Solving

In practice, the first two subphases of qualifier solving often are sufficient to find a constant qualifier bound for each remaining generalization candidate. As a backstop to the constraint rewriting system, however, the generalization and constraint-solving procedure can invoke a SAT solver on the subqualifier portion of the constraint.

Constraints with subqualifier inequalities in standard form are easily converted to conjunctive normal form:

$$\begin{aligned}
[\![\top]\!] &= \top \\
[\![\tau_1 \leq \tau_2]\!] &= \top \\
[\![\mathsf{A} \sqsubseteq \langle\beta_1\rangle \sqcup \ldots \sqcup \langle\beta_k\rangle]\!] &= \beta_1 \vee \cdots \vee \beta_k \\
[\![\langle\alpha\rangle \sqsubseteq \langle\beta_1\rangle \sqcup \ldots \sqcup \langle\beta_k\rangle]\!] &= \neg\alpha \vee \beta_1 \vee \cdots \vee \beta_k \\
[\![C \wedge D]\!] &= [\![C]\!] \wedge [\![D]\!] \\
[\![\exists\alpha.C]\!] &= [\![C]\!] \quad \alpha \text{ fresh}
\end{aligned}$$

The result of the SAT solver is then used to construct a substitution. The entire result of the solver is not necessarily used, but enough is used to ensure that no generalization candidate type variables appear on the right side of any subqualifier inequalities.

$\boxed{\mathscr{C};\Gamma;\tau \vdash C_1 \overset{\theta}{\mapsto} C_2}$ *(constraint rewriting)*

$$\text{CQ-BoundU}\quad \frac{\beta \in \mathrm{FTV}(\tau) \setminus (\mathrm{FTV}(\Gamma) \cup \mathrm{FTV}(\mathscr{C}))}{\mathscr{C};\Gamma;\tau \vdash \overline{\langle\beta\rangle \sqsubseteq \xi_i} \dot{\mapsto} \langle\beta\rangle \sqsubseteq \mathsf{U}} \qquad \text{CQ-BoundA}\quad \frac{\beta \in \mathrm{FTV}(\tau) \setminus (\mathrm{FTV}(\Gamma) \cup \mathrm{FTV}(\mathscr{C}))}{\mathscr{C};\Gamma;\tau \vdash \top \dot{\mapsto} \langle\beta\rangle \sqsubseteq \mathsf{A}}$$

Figure 6.20: Finding constant qualifier bounds (lossy)

6.4.4 Bounding

Constraint solving ends with the bounding phase, which rewrites the constraint into the final form expected by the generalization procedure, using two rules that appear in figure 6.20. For each generalization candidate type variable β, it computes an upper bound of either U or A. In particular, if β appears with any upper bounds in the qualifier constraint, rule CQ-BoundU bounds it by U and ensures that β appears nowhere else in the constraint. Otherwise, if β does not appear in the constraint at all, then rule CQ-BoundA bounds it by A. The end result is a constraint that can be split by rule Gen into bounds for generalized variables in a type scheme and the remaining constraint, in which the generalized variables do not appear.

CHAPTER 7

Mixing Affine and Conventional Types

THE DEVELOPMENT OF software systems benefits from access to comprehensive libraries, which mainstream programming languages usually provide but experimental language implementations often do not. While it is possible to translate libraries from another language—or to design them from scratch—another possibility is to allow code written in an affine language like Alms to interact with code written in a similar but conventional (*i.e.,* not substructural) language.

I envision complementary scenarios:

- A programmer wishes to import legacy library code for use by affine-typed client code. Unfortunately, legacy code unaware of the substructural conditions may duplicate values received from the substructural language.

- A programmer wishes to export substructural library code for access from a conventional language. A client may duplicate values received from the library and resubmit them, causing aliasing that the library could not produce on its own and bypassing the substructural type system's guarantees.

In a higher-order setting, both scenarios arise naturally when higher-order values cross the interlanguage boundary. Beyond the two-language case, my solution is useful within a single affine language, such as Alms, because it

provides a principled approach to combining dynamic and static enforcement of affinity.

In this chapter, I describe a novel approach to regulating the interaction between an affine language and a conventionally-typed language, with the following features:

- The non-affine language may gain access to affine values and may apply affine-language functions.
- The non-affine type system is utterly standard, making no concessions to the affine type system.
- And yet, the composite system preserves the affine language's invariants.

The principal features of such a system are modeled by a multi-language calculus that enjoys type soundness. In particular, the conventional language, although it has access to the affine language's functions and values, cannot be used to subvert the affine type system.

My solution, in short, is to impose dynamic checks by wrapping each exchanged value in a behavioral contract (Findler and Felleisen 2002) that uses one bit of state to track when an affine value has been used. While the idea is simple, the details can be subtle.

The bulk of this chapter comprises a model of a two-language system with affine contracts (§7.2) and its type soundness proof (§7.3). After that, I discuss an implementation strategy for a hypothetical two-language system (§7.4.1) and show how affine contracts are also useful in Alms even though it does not interface with a conventional language (§7.4.2).

7.1 Related Work

To monitor the flow of values between affine and conventional modules, I use Findler and Felleisen's (2002) higher-order contracts. My approach to integrating affine and conventional types borrows from more recent literature on multi-language interoperability (Matthews and Findler 2007; Tobin-Hochstadt and Felleisen 2008). In particular, the technique and models in this

chapter closely follow that of a predecessor of Typed Racket (Tobin-Hochstadt and Felleisen 2006), which uses contracts to mediate between modules in an untyped, Scheme-like language and a typed language. Instead, I use software contracts to mediate between modules in an affine, Alms-like language and a conventionally-typed language.

One important feature of the contracts in this chapter is that they are *stateful*. When an affine contract is created, it allocates a reference cell, which creates a channel of communication between subsequent invocations of the same contract. Contracts with state have found another application in enforcing parametricity, either by dynamic sealing (Matthews and Ahmed 2008) or by keeping track of which values pass through a function parameter whose type involves a universally quantified type variable (Sam Tobin-Hochstadt 2009, personal communication).

7.2 A Model of Affine Contracts

This chapter concerns safe interoperation between a pair of languages, one with affine types and the other with a conventional type system. This can be achieved using software contracts to mediate the interaction.

In Findler and Felleisen's (2002) formulation, a contract is an agreement between two software components, or *parties*, about some property of a value. The *positive party* produces a value, which must satisfy the specified property. The *negative party* consumes the value and is held responsible for treating it appropriately. Contracts are concerned with catching violations of the property and blaming the guilty party, which may help locate the source of a bug. For first-order values the contract may be immediately checkable, but for functional values nontrivial properties are undecidable, so the check must wait until the negative party applies the function, at which point the negative party is responsible for providing a suitable argument and the positive party for producing a suitable result. Thus, for higher-order functions, checks are delayed until first-order values are reached.

In the model described in this section, the parties to contracts are modules, which must be in entirely one language or the other. For simplicity, modules

do not nest, and a module associates one name with one value. I begin by describing the model of the conventional language, $F_{\mathscr{C}}$, which is the call-by-value, polymorphic λ calculus (Girard 1972) extended with single-value modules; I then describe the affine language, $F^{\mathscr{A}}$, which is similar to $F_{\mathscr{C}}$, but with affine types in the style of ${}^{a}\lambda_{ms}$, the model in chapter 5. The two sublanguages are combined in the mixed model $F^{\mathscr{A}}_{\mathscr{C}}$.

7.2.1 The Conventional Language $F_{\mathscr{C}}$

The syntax of $F_{\mathscr{C}}$ appears in figure 7.1. As in System F, $F_{\mathscr{C}}$'s types include type variables, function types, and universally quantified types, and its expressions include variables, λ abstractions, and type abstractions. Additionally, $F_{\mathscr{C}}$ includes modules. A module $\mathbf{f} : \boldsymbol{\tau} = \mathbf{v}$ associates a module name with a type and value, and a module invocation expression (**f**) merely reduces to the value. Modules allow recursion and, more importantly, provide the means for interlanguage interaction, since each language will be be able to invoke modules written in the other. An $F_{\mathscr{C}}$ program (**P**) comprises a collection of modules (M) and an expression. ($F_{\mathscr{C}}$ terms are set in boldface to distinguish them from $F^{\mathscr{A}}$.)

Operational semantics. The operational semantics of $F_{\mathscr{C}}$ appears in figure 7.2. The reduction relation is parametrized by a collection of modules, M. The only non-standard rule is C-MOD, which reduces a module name to its body.

Static semantics. The type system of $F_{\mathscr{C}}$ appears in figure 7.3. Expressions are typed under a module context M, which is merely a collection of modules, and a typing context $\boldsymbol{\Gamma}$ that associates variables names with their types. All expression type rules are standard except for rule CT-MOD, which assigns a module name the type declared in the module itself, as found in the module context M.

$F_{\mathscr{C}}$ also requires judgments for checking individual modules and whole programs. A module $\mathbf{f} : \boldsymbol{\tau} = \mathbf{v}$ is okay in a module context M if **v** has type $\boldsymbol{\tau}$ in that same context. A program let M in **e** has type $\boldsymbol{\tau}$ if all modules in M are

$\boldsymbol{\alpha}, \boldsymbol{\beta}$	$\in$	$TVar_{\mathscr{C}}$	type variables
$\mathbf{x}, \mathbf{y}$	$\in$	$Var_{\mathscr{C}}$	variables
$\mathbf{f}, \mathbf{g}$	$\in$	$MVar_{\mathscr{C}}$	module names
$\boldsymbol{\tau}$	::=		types
	|	$\boldsymbol{\alpha}$	type variable
	|	$\boldsymbol{\tau}_1 \to \boldsymbol{\tau}_2$	function type
	|	$\forall\, \boldsymbol{\alpha}.\, \boldsymbol{\tau}$	universal type
$\boldsymbol{\Gamma}$	::=		typing contexts
	|	$\bullet$	empty context
	|	$\boldsymbol{\Gamma}, \mathbf{x} : \boldsymbol{\tau}$	with a variable
$\mathbf{v}$	::=		values
	|	$\mathbf{x}$	variable occurrence
	|	$\lambda \mathbf{x}{:}\boldsymbol{\tau}.\, \mathbf{e}$	abstraction
	|	$\Lambda \boldsymbol{\alpha}.\, \mathbf{v}$	type abstraction
$\mathbf{e}$	::=		expressions
	|	$\mathbf{v}$	values
	|	$\mathbf{f}$	module invocation
	|	$\mathbf{e}_1\, \mathbf{e}_2$	application
	|	$\mathbf{e}\, \boldsymbol{\tau}$	type application
m	::=	$\mathbf{f} : \boldsymbol{\tau} = \mathbf{v}$	modules
M	::=	$\mathrm{m}_1, \ldots, \mathrm{m}_k$	module contexts
$\mathbf{P}$	::=	let M in $\mathbf{e}$	programs

Figure 7.1: Syntax of $\mathrm{F}_{\mathscr{C}}$

$\boxed{\mathbf{e} \xrightarrow[\mathrm{M}]{} \mathbf{e}'}$ *($F_{\mathscr{C}}$ reduction)*

$$\begin{array}{lll}
\text{(C-}\beta_v\text{)} & (\lambda \mathbf{x}{:}\boldsymbol{\tau}.\,\mathbf{e})\,\mathbf{v} \xrightarrow[\mathrm{M}]{} \{\mathbf{v}/\mathbf{x}\}\mathbf{e} & \\
\text{(C-B}_\tau\text{)} & (\Lambda \boldsymbol{\alpha}.\,\mathbf{v})\,\boldsymbol{\tau} \xrightarrow[\mathrm{M}]{} \{\boldsymbol{\tau}/\boldsymbol{\alpha}\}\mathbf{v} & \\
\text{(C-MOD)} & \mathbf{f} \xrightarrow[\mathrm{M}]{} \mathbf{v} & \text{when } (\mathbf{f} : \boldsymbol{\tau} = \mathbf{v}) \in \mathrm{M}
\end{array}$$

$\boxed{\mathbf{e} \xmapsto[\mathrm{M}]{} \mathbf{e}'}$ *($F_{\mathscr{C}}$ reduction)*

$$\text{(C-CXT)} \qquad \frac{\mathbf{e} \xrightarrow[\mathrm{M}]{} \mathbf{e}'}{\mathbf{E}[\mathbf{e}] \xmapsto[\mathrm{M}]{} \mathbf{E}[\mathbf{e}']}$$

where $\mathbf{E} ::= [] \mid \mathbf{E}\,\mathbf{e} \mid \mathbf{v}\,\mathbf{E} \mid \mathbf{E}\,\boldsymbol{\tau}$

Figure 7.2: Operational semantics of $\mathrm{F}_{\mathscr{C}}$

okay in the context of M (hence, recursion), and if **e** has type $\boldsymbol{\tau}$ in the same module context.

7.2.2 The Affine Language $\mathrm{F}^{\mathscr{A}}$

The syntax of $\mathrm{F}^{\mathscr{A}}$ appears in figure 7.4. $\mathrm{F}^{\mathscr{A}}$ extends $\mathrm{F}_{\mathscr{C}}$ with products, reference cells, and affine types.

Like Alms, $\mathrm{F}^{\mathscr{A}}$ includes two qualifiers, U and A, but unlike Alms, it does not include qualifier expressions built out of variables. Rather than use a kind system to track the qualifiers of type variables, $\mathrm{F}^{\mathscr{A}}$ has two disjoint sets of type variables: the unlimited type variables $TVarU_{\mathscr{A}}$ and the affine type variables $TVarA_{\mathscr{A}}$. (Disjoint sets are a simplification to dispense with kind environments, but nothing here is incompatible with using kinding instead.) Type variables are decorated with their qualifier (α^{U} and α^{A}), and we use the form α^{q} to refer generically to a type variable of either qualifier.

The types of $\mathrm{F}^{\mathscr{A}}$ (τ) distinguish a class of "opaque types" (ρ), which will be embedded opaquely, with no operations, in $\mathrm{F}_{\mathscr{C}}$. This distinction has no relevance for $\mathrm{F}^{\mathscr{A}}$ alone, but will play a rule in defining the mixed language

$\boxed{\mathrm{M};\Gamma \vdash_{\mathscr{C}} \mathbf{e} : \tau}$ *($F_{\mathscr{C}}$ expression typing)*

$$\textsc{CT-Var}\ \frac{\mathbf{x} : \tau \in \Gamma}{\mathrm{M};\Gamma \vdash_{\mathscr{C}} \mathbf{x} : \tau} \qquad \textsc{CT-Mod}\ \frac{(\mathbf{f} : \tau = \mathbf{v}) \in \mathrm{M}}{\mathrm{M};\Gamma \vdash_{\mathscr{C}} \mathbf{f} : \tau}$$

$$\textsc{CT-TAbs}\ \frac{\mathrm{M};\Gamma \vdash_{\mathscr{C}} \mathbf{v} : \tau}{\mathrm{M};\Gamma \vdash_{\mathscr{C}} \Lambda\alpha.\,\mathbf{v} : \forall\alpha.\,\tau} \qquad \textsc{CT-Abs}\ \frac{\mathrm{M};\Gamma, \mathbf{x} : \tau \vdash_{\mathscr{C}} \mathbf{e} : \tau'}{\mathrm{M};\Gamma \vdash_{\mathscr{C}} \lambda\mathbf{x}{:}\tau.\,\mathbf{e} : \tau \to \tau'}$$

$$\textsc{CT-TApp}\ \frac{\mathrm{M};\Gamma \vdash_{\mathscr{C}} \mathbf{e} : \forall\alpha.\,\tau'}{\mathrm{M};\Gamma \vdash_{\mathscr{C}} \mathbf{e}\,\tau : \{\tau/\alpha\}\tau'} \qquad \textsc{CT-App}\ \frac{\mathrm{M};\Gamma \vdash_{\mathscr{C}} \mathbf{e}_1 : \tau' \to \tau \qquad \mathrm{M};\Gamma \vdash_{\mathscr{C}} \mathbf{e}_2 : \tau'}{\mathrm{M};\Gamma \vdash_{\mathscr{C}} \mathbf{e}_1\,\mathbf{e}_2 : \tau}$$

$\boxed{\mathrm{M} \vdash \mathrm{m}\ \mathrm{okay}}$, $\boxed{\vdash_{\mathscr{C}} \mathbf{P} : \tau}$ *($F_{\mathscr{C}}$ module and program typing)*

$$\textsc{ModuleC}\ \frac{\mathrm{M};\bullet \vdash_{\mathscr{C}} \mathbf{v} : \tau \qquad \mathrm{FTV}(\tau) = \emptyset}{\mathrm{M} \vdash \mathbf{f} : \tau = \mathbf{v}\ \mathrm{okay}} \qquad \textsc{C-Prog}\ \frac{(\forall \mathrm{m} \in \mathrm{M})\,\mathrm{M} \vdash \mathrm{m}\ \mathrm{okay} \qquad \mathrm{M};\bullet \vdash_{\mathscr{C}} \mathbf{e} : \tau}{\vdash_{\mathscr{C}} \mathsf{let}\ \mathrm{M}\ \mathsf{in}\ \mathbf{e} : \tau}$$

Figure 7.3: Static semantics of $\mathrm{F}_{\mathscr{C}}$

in §7.2.3. Opaque types include product types ($\tau_1 \otimes \tau_2$) and reference types ($\mathsf{ref}\ \tau$). The values include pair construction ($\langle v_1, v_2 \rangle$). Expressions include a destructuring let for pair elimination, and two operations on references, allocation and swap, which combines reading and writing. $\mathrm{F}^{\mathscr{A}}$'s treatment of modules follows $\mathrm{F}_{\mathscr{C}}$'s.

Operational semantics. The operational semantics of $\mathrm{F}^{\mathscr{A}}$ appears in figure 7.5. The run-time syntax includes locations (ℓ), and stores (s), which map locations to values. The reduction relation is defined over a pair of configurations ($s, \mathbf{e}$), each of which pairs a store with an expression; as with $\mathrm{F}_{\mathscr{C}}$, reduction is parametrized by a module context M. Four reduction rules are worth noting: $\mathrm{F}^{\mathscr{A}}$'s treatment of modules, by rule A-Mod, follows that of $\mathrm{F}_{\mathscr{C}}$. Rule A-LetPair destructures a pair, substituting its components in the

$\mathfrak{q}$	$\in$	$\{\mathsf{U}, \mathsf{A}\}$	qualifiers
$\alpha^{\mathsf{U}}, \beta^{\mathsf{U}}$	$\in$	$TVarU_{\mathscr{A}}$	unlimited type variables
$\alpha^{\mathsf{A}}, \beta^{\mathsf{A}}$	$\in$	$TVarA_{\mathscr{A}}$	affine type variables
x, y	$\in$	$Var_{\mathscr{A}}$	variables
f, g	$\in$	$MVar_{\mathscr{A}}$	module names
τ	::=		types
	$\mid$	ρ	opaque type
	$\mid$	$\tau_1 \xrightarrow{\mathfrak{q}} \tau_2$	affine function type
	$\mid$	$\forall\, \alpha^{\mathfrak{q}}.\, \tau$	universal type
ρ	::=		opaque types
	$\mid$	$\alpha^{\mathfrak{q}}$	type variable
	$\mid$	$\mathsf{ref}\ \tau$	reference type
	$\mid$	$\tau_1 \otimes \tau_2$	product type
Γ	::=		typing contexts
	$\mid$	$\bullet$	empty context
	$\mid$	$\Gamma, x : \tau$	with a variable
v	::=		values
	$\mid$	x	variable occurrence
	$\mid$	$\lambda x{:}\tau.\, e$	abstraction
	$\mid$	$\Lambda \alpha^{\mathfrak{q}}.\, v$	type abstraction
	$\mid$	$\langle v_1, v_2 \rangle$	pair introduction
e	::=		expressions
	$\mid$	v	values
	$\mid$	f	module invocation
	$\mid$	$e_1\, e_2$	application
	$\mid$	$e\, \tau$	type application
	$\mid$	$\mathsf{let}\ \langle x, y \rangle = e_1\ \mathsf{in}\ e_2$	pair elimination
	$\mid$	$\mathsf{new}\ e$	reference allocation
	$\mid$	$\mathsf{swap}\ e$	reference read/write
m	::=	$f : \tau = v$	modules
$\mathbf{M}$	::=	$\mathrm{m}_1, \ldots, \mathrm{m}_k$	module contexts
P	::=	$\mathsf{let}\ \mathbf{M}\ \mathsf{in}\ e$	programs

Figure 7.4: Syntax of $\mathrm{F}^{\mathscr{A}}$

$\ell \in Loc$ locations

$v ::= \cdots \mid \ell$ run-time values

$s ::= \{\} \mid \{\ell \mapsto v\} \mid s_1 \uplus s_2$ stores

$\boxed{(s,e) \xrightarrow[\mathrm{M}]{} (s',e')}$ *($F^{\mathscr{A}}$ reduction)*

(A-β_v) $(s,(\lambda x{:}\tau.\, e)\, v) \xrightarrow[\mathrm{M}]{} (s,\{v/x\}e)$

(A-B_τ) $(s,(\Lambda\alpha^{\mathsf{q}}.\, v)\,\tau) \xrightarrow[\mathrm{M}]{} (s,\{\tau/\alpha^{\mathsf{q}}\}v)$

(A-Mod) $(s,f) \xrightarrow[\mathrm{M}]{} (s,v)$ when $(f : \tau = v) \in \mathrm{M}$

(A-LetPair) $(s,\mathsf{let}\,\langle x,y\rangle = \langle v_1,v_2\rangle\,\mathsf{in}\,e) \xrightarrow[\mathrm{M}]{} (s,\{v_1/x\}\{v_2/y\}e)$

(A-New) $(s,\mathsf{new}\,v) \xrightarrow[\mathrm{M}]{} (s \uplus \{\ell \mapsto v\},\ell)$

(A-Swap) $(s \uplus \{\ell \mapsto v_1\},\mathsf{swap}\,\langle \ell,v_2\rangle) \xrightarrow[\mathrm{M}]{} (s \uplus \{\ell \mapsto v_2\},\langle \ell,v_1\rangle)$

$\boxed{(s,e) \xrightarrow[\mathrm{M}]{} (s',e')}$ *($F^{\mathscr{A}}$ reduction)*

(A-Cxt)
$$\frac{(s,e) \xrightarrow[\mathrm{M}]{} (s',e')}{(s,E[e]) \xrightarrow[\mathrm{M}]{} (s',E[e'])}$$

where $E ::= [] \mid E\,e \mid v\,E \mid E\,\tau \mid \mathsf{let}\,\langle x,y\rangle = E\,\mathsf{in}\,e \mid \mathsf{new}\,E \mid \mathsf{swap}\,E$

Figure 7.5: Operational semantics of $\mathrm{F}^{\mathscr{A}}$

body of the let. By rule A-New, new v allocates a fresh location ℓ in the store and associates it with value v, reducing to location ℓ. Finally, by rule A-Swap, swap $\langle \ell, v_2\rangle$ requires the store to map location ℓ to some value v_1, in which case it replaces v_1 with v_2 in the store and returns a pair of the same location ℓ and value v_1 read from the store.

Static semantics. The presence of affine types requires several more judgments in $\mathrm{F}^{\mathscr{A}}$'s type system than $\mathrm{F}_{\mathscr{C}}$ requires. Figure 7.6 defines three new judgments and one metafunction:

- The qualifier subsumption judgment ($\mathsf{q}_1 \sqsubseteq \mathsf{q}_2$) defines the qualifier lattice, with U on the bottom and A on top.

$\boxed{\mathfrak{q}_1 \sqsubseteq \mathfrak{q}_2}$ *($F^{\mathscr{A}}$ qualifier subsumption)*

$$\textsc{AQSub-Bot}\ \frac{}{\mathsf{U} \sqsubseteq \mathfrak{q}} \qquad \textsc{AQSub-Top}\ \frac{}{\mathfrak{q} \sqsubseteq \mathsf{A}}$$

$\boxed{\langle \tau \rangle = \mathfrak{q}}$, $\boxed{\langle \Gamma \rangle = \mathfrak{q}}$ *($F^{\mathscr{A}}$ qualifier assignment)*

$$\langle \alpha^{\mathfrak{q}} \rangle = \mathfrak{q} \qquad \langle \mathsf{ref}\ \tau \rangle = \mathsf{A} \qquad \langle \tau_1 \otimes \tau_2 \rangle = \langle \tau_1 \rangle \sqcup \langle \tau_2 \rangle \qquad \langle \tau_1 \xrightarrow{\mathfrak{q}} \tau_2 \rangle = \mathfrak{q}$$

$$\langle \forall\, \alpha^{\mathfrak{q}}.\, \tau \rangle = \langle \tau \rangle \qquad \langle \Gamma \rangle = \bigsqcup_{x:\tau \in \Gamma} \langle \tau \rangle$$

$\boxed{\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2}$ *($F^{\mathscr{A}}$ context splitting)*

$$\textsc{AS-ConsL}\ \frac{\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \langle \tau \rangle = \mathsf{A}}{\Gamma, x:\tau \rightsquigarrow \Gamma_1, x:\tau \boxplus \Gamma_2} \qquad \textsc{AS-ConsR}\ \frac{\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \langle \tau \rangle = \mathsf{A}}{\Gamma, x:\tau \rightsquigarrow \Gamma_1 \boxplus \Gamma_2, x:\tau}$$

$$\textsc{AS-Cons}\ \frac{\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \langle \tau \rangle = \mathsf{U}}{\Gamma, x:\tau \rightsquigarrow \Gamma_1, x:\tau \boxplus \Gamma_2, x:\tau} \qquad \textsc{AS-Nil}\ \frac{}{\bullet \rightsquigarrow \bullet \boxplus \bullet}$$

$\boxed{\tau_1 <: \tau_2}$ *($F^{\mathscr{A}}$ subtyping)*

$$\textsc{ASub-Prod}\ \frac{\tau_1 <: \tau_1' \qquad \tau_2 <: \tau_2'}{\tau_1 \otimes \tau_2 <: \tau_1' \otimes \tau_2'} \qquad \textsc{ASub-Arr}\ \frac{\tau_1' <: \tau_1 \qquad \mathfrak{q} \sqsubseteq \mathfrak{q}' \qquad \tau_2 <: \tau_2'}{\tau_1 \xrightarrow{\mathfrak{q}} \tau_2 <: \tau_1' \xrightarrow{\mathfrak{q}'} \tau_2'}$$

$$\textsc{ASub-All}\ \frac{\mathfrak{q}' \sqsubseteq \mathfrak{q} \qquad \{\beta^{\mathfrak{q}'}/\alpha^{\mathfrak{q}}\}\tau <: \tau'}{\forall\, \alpha^{\mathfrak{q}}.\, \tau <: \forall\, \beta^{\mathfrak{q}'}.\, \tau'} \qquad \textsc{ASub-Refl}\ \frac{}{\tau <: \tau}$$

Figure 7.6: Static semantics of $\mathrm{F}^{\mathscr{A}}$(i)

- Metafunction $\langle\cdot\rangle$ maps types and typing contexts (Γ) to qualifiers. The qualifier of a type variable $\alpha^{\mathfrak{q}}$ or function type $\tau_1 \xrightarrow{\mathfrak{q}} \tau_2$ is the qualifier $\mathfrak{q}$ that decorates that type; the qualifier of a reference is always A; the qualifier of a product type is the least upper bound of the qualifiers of the components; and the qualifier of a universal type $\forall\,\alpha^{\mathfrak{q}}.\,\tau$ is the qualifier of its body τ.

  The qualifier of a typing context is the least upper bound of the qualifiers of all the types in its range.

- The context-splitting judgment $\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$ splits a context Γ into two that share Γ's unlimited variables and partition Γ's affine variables. This is used for checking multiplicative expressions such as application, which must partition affine assumptions between the subterms.

- The last judgment of figure 7.6 defines subtyping, which is similar to subtyping in Alms, but is defined for $\mathrm{F}^{\mathscr{A}}$'s few built-in types rather than relying on a kind system. Product types are covariant in both components. Function types, as in Alms, are contravariant in the domain, covariant in the qualifier, and covariant in the range. Universally quantified types are contravariant in the qualifiers of their bound variables and covariant in their bodies; the rule relies on substitution to rename bound variables because α equivalence for $\mathrm{F}^{\mathscr{A}}$ does not include bound variables with different qualifiers.

The judgments for typing expressions, modules, and programs appear in figure 7.7. The module and program judgments are as in $\mathrm{F}_{\mathscr{C}}$, with one difference: $\mathrm{F}^{\mathscr{A}}$ requires that the types of modules be unlimited, because as in Alms the type system does not track how many times top level bindings are used. The expression typing judgment is as expected for an affine language with Alms-style dereliction subtyping. In particular, rule AT-SUBSUME does subsumption according to the subtyping relation, which allows treating a function with qualifier U as if it has qualifier A. Rule AT-TAPP requires not only that in a type application $e\,\tau$, e have a universal type $\forall\,\alpha^{\mathfrak{q}}.\,\tau'$, but that the qualifier of τ not exceed qualifier $\mathfrak{q}$. Rule AT-SWAP does not require

$\boxed{\mathrm{M};\Gamma \vdash_{\mathscr{A}} e : \tau}$ *($F^{\mathscr{A}}$ expression typing)*

$$\textsc{AT-Subsume}\ \frac{\mathrm{M};\Gamma \vdash_{\mathscr{A}} e : \tau' \qquad \tau' <: \tau}{\mathrm{M};\Gamma \vdash_{\mathscr{A}} e : \tau} \qquad \textsc{AT-Var}\ \frac{x : \tau \in \Gamma}{\mathrm{M};\Gamma \vdash_{\mathscr{A}} x : \tau} \qquad \textsc{AT-Mod}\ \frac{(f : \tau = v) \in \mathrm{M}}{\mathrm{M};\Gamma \vdash_{\mathscr{A}} f : \tau}$$

$$\textsc{AT-TAbs}\ \frac{\mathrm{M};\Gamma \vdash_{\mathscr{A}} v : \tau}{\mathrm{M};\Gamma \vdash_{\mathscr{A}} \Lambda\alpha^{\mathfrak{q}}.v : \forall\,\alpha^{\mathfrak{q}}.\tau} \qquad \textsc{AT-Abs}\ \frac{\mathrm{M};\Gamma, x:\tau \vdash_{\mathscr{A}} e : \tau' \qquad \langle \Gamma|_{\mathrm{FV}(\lambda x:\tau.e)} \rangle = \mathfrak{q}}{\mathrm{M};\Gamma \vdash_{\mathscr{A}} \lambda x{:}\tau.e : \tau \xrightarrow{\mathfrak{q}} \tau'}$$

$$\textsc{AT-Pair}\ \frac{\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \mathrm{M};\Gamma_1 \vdash_{\mathscr{A}} v_1 : \tau_1 \qquad \mathrm{M};\Gamma_2 \vdash_{\mathscr{A}} v_2 : \tau_2}{\mathrm{M};\Gamma \vdash_{\mathscr{A}} \langle v_1, v_2 \rangle : \tau_1 \otimes \tau_2}$$

$$\textsc{AT-TApp}\ \frac{\mathrm{M};\Gamma \vdash_{\mathscr{A}} e : \forall\,\alpha^{\mathfrak{q}}.\tau' \qquad \langle \tau \rangle \sqsubseteq \mathfrak{q}}{\mathrm{M};\Gamma \vdash_{\mathscr{A}} e\,\tau : \{\tau/\alpha^{\mathfrak{q}}\}\tau'} \qquad \textsc{AT-App}\ \frac{\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \mathrm{M};\Gamma_1 \vdash_{\mathscr{A}} e_1 : \tau' \xrightarrow{\mathfrak{q}} \tau \qquad \mathrm{M};\Gamma_2 \vdash_{\mathscr{A}} e_2 : \tau'}{\mathrm{M};\Gamma \vdash_{\mathscr{A}} e_1\,e_2 : \tau}$$

$$\textsc{AT-LetPair}\ \frac{\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \mathrm{M};\Gamma_1 \vdash_{\mathscr{A}} e_1 : \tau_1 \otimes \tau_2 \qquad \mathrm{M};\Gamma_2, x:\tau_1, y:\tau_2 \vdash_{\mathscr{A}} e_2 : \tau}{\mathrm{M};\Gamma \vdash_{\mathscr{A}} \mathsf{let}\,\langle x, y\rangle = e_1\,\mathsf{in}\,e_2 : \tau}$$

$$\textsc{AT-New}\ \frac{\mathrm{M};\Gamma \vdash_{\mathscr{A}} e : \tau}{\mathrm{M};\Gamma \vdash_{\mathscr{A}} \mathsf{new}\,e : \mathsf{ref}\,\tau} \qquad \textsc{AT-Swap}\ \frac{\mathrm{M};\Gamma \vdash_{\mathscr{A}} e : \mathsf{ref}\,\tau_1 \otimes \tau_2}{\mathrm{M};\Gamma \vdash_{\mathscr{A}} \mathsf{swap}\,e : \mathsf{ref}\,\tau_2 \otimes \tau_1}$$

$\boxed{\mathrm{M} \vdash \mathrm{m}\ \mathrm{okay}}$, $\boxed{\vdash_{\mathscr{A}} P : \tau}$ *($F^{\mathscr{A}}$ module and program typing)*

$$\textsc{ModuleA}\ \frac{\mathrm{M};\bullet \vdash_{\mathscr{A}} v : \tau \qquad \mathrm{FTV}(\tau) = \emptyset \qquad \langle \tau \rangle = \mathsf{U}}{\mathrm{M} \vdash f : \tau = v\ \mathrm{okay}}$$

$$\textsc{A-Prog}\ \frac{(\forall \mathrm{m} \in \mathrm{M})\,\mathrm{M} \vdash \mathrm{m}\ \mathrm{okay} \qquad \mathrm{M};\bullet \vdash_{\mathscr{A}} e : \tau}{\vdash_{\mathscr{A}} \mathsf{let}\,\mathrm{M}\,\mathsf{in}\,e : \tau}$$

Figure 7.7: Static semantics of $\mathrm{F}^{\mathscr{A}}$(ii)

$\boldsymbol{\tau}$	::=	$\cdots$	additional $\mathrm{F}_{\mathscr{C}}$ types
	|	$\{\rho\}$	embedded $\mathrm{F}^{\mathscr{A}}$ opaque type
τ	::=	$\cdots$	additional $\mathrm{F}^{\mathscr{A}}$ types
	|	$\{\boldsymbol{\alpha}\}$	embedded $\mathrm{F}_{\mathscr{C}}$ type variable
$\mathbf{e}$	::=	$\cdots$	additional $\mathrm{F}_{\mathscr{C}}$ expressions
	|	$f^{\mathbf{g}}$	$\mathrm{F}^{\mathscr{A}}$ module invocation
e	::=	$\cdots$	additional $\mathrm{F}^{\mathscr{A}}$ expressions
	|	$\mathbf{f}^{g}$	$\mathrm{F}_{\mathscr{C}}$ module invocation
m	::=		modules
	|	$\mathbf{f} : \boldsymbol{\tau} = \mathbf{v}$	an $\mathrm{F}_{\mathscr{C}}$ module
	|	$f : \tau = v$	an $\mathrm{F}^{\mathscr{A}}$ module
	|	$\mathbf{f} :> \tau = \mathbf{g}$	an interface
M	::=		module contexts
	|	$\mathrm{m}_1, \ldots, \mathrm{m}_k$	module context
P	::=	let M in $\mathbf{e}$	programs

Figure 7.8: Additional syntax for $\mathrm{F}^{\mathscr{A}}_{\mathscr{C}}$

that the type of a reference match the type of the value to be stored in the reference, which means that swap performs a *strong update*: It allows the type of a reference to change. The remaining rules are straightforward. Rules AT-PAIR, AT-APP, and AT-LETPAIR each have two subexpressions combined multiplicatively, which means that each needs to split the context.

7.2.3 The Mixed Language $\mathrm{F}^{\mathscr{A}}_{\mathscr{C}}$

The primary aim of this chapter is to construct (type-safe) programs by mixing modules written in an affine language and modules written in a non-affine language, and to have them interoperate as seamlessly as possible. We can then model an affine program calling into a library written in a legacy language, or a conventional program calling into code written in an affine language. In

either case, we must ensure that the non-affine portions of the program do not break the affine portions' invariants.

In this section, I show how to combine $\mathrm{F}_{\mathscr{C}}$ modules and $\mathrm{F}^{\mathscr{A}}$ modules into a single program. The additional syntax necessary appears in figure 7.8.

When one language invokes a module from the other language, it needs some way to refer to the type of that module. In the type system for $\mathrm{F}^{\mathscr{A}}_{\mathscr{C}}$, I will give a mapping between the two languages' types. Some types translate into native types of the other language, but some types do not. From the perspective of $\mathrm{F}_{\mathscr{C}}$, type variables, product types, and reference types imported from $\mathrm{F}^{\mathscr{A}}$ must appear opaque, supporting no operations other than sending them back to $\mathrm{F}^{\mathscr{A}}$. These are the opaque types (ρ) of $\mathrm{F}^{\mathscr{A}}$, and appear as $\{\rho\}$ in $\mathrm{F}_{\mathscr{C}}$. From the perspective of $\mathrm{F}^{\mathscr{A}}$, only $\mathrm{F}_{\mathscr{C}}$ type variables are opaque, because all other types translate; the $\mathrm{F}_{\mathscr{C}}$ type variable $\boldsymbol{\alpha}$ appears as $\{\boldsymbol{\alpha}\}$ in $\mathrm{F}^{\mathscr{A}}$.

To invoke modules from another language, each language needs among its expressions a way to refer to modules of the other language. We will protect interlanguage communication with contracts, and for each interlanguage module invocation, the module being invoked is the positive party. For the negative party, we use the module in which the invocation appears, and we assume that each interlanguage module invocation is already annotated with the negative party. Thus, to invoke an $\mathrm{F}^{\mathscr{A}}$ module f from $\mathrm{F}_{\mathscr{C}}$ module $\mathbf{g}$, we use the $\mathrm{F}_{\mathscr{C}}$ expression $f^{\mathbf{g}}$. Likewise, to invoke an $\mathrm{F}_{\mathscr{C}}$ module $\mathbf{g}$ from $\mathrm{F}^{\mathscr{A}}$ module f, we use the $\mathrm{F}^{\mathscr{A}}$ expression $\mathbf{g}^{f}$.

Modules now include $\mathrm{F}_{\mathscr{C}}$ modules ($\mathbf{f} : \boldsymbol{\tau} = \mathbf{v}$), $\mathrm{F}^{\mathscr{A}}$ modules ($f : \tau = v$), and interface modules, which allow assigning an $\mathrm{F}^{\mathscr{A}}$ type to an $\mathrm{F}_{\mathscr{C}}$ module. By default, when an $\mathrm{F}_{\mathscr{C}}$ module is referenced from $\mathrm{F}^{\mathscr{A}}$, it will be assumed to have a type with only unlimited arrows—that is, it is assumed not to respect affine invariants. An interface module $\mathbf{f} :> \tau = \mathbf{f}'$ declares $\mathbf{f}$ to evaluate to the same value as $\mathbf{f}'$ but to behave like type τ. As we will see, this requires run-time checks.

An $\mathrm{F}^{\mathscr{A}}_{\mathscr{C}}$ program comprises a collection of modules, which may be in either language or may be interfaces, and a main expression, which is (arbitrarily) in $\mathrm{F}_{\mathscr{C}}$.

$\boxed{\langle\tau\rangle = \mathfrak{q}}$ *(addition to $F^{\mathscr{A}}$ qualifier assignment)*

$$\langle\{\boldsymbol{\alpha}\}\rangle = \mathsf{U}$$

$\boxed{\boldsymbol{\tau}^{\mathscr{A}} = \tau}$, $\boxed{\tau^{\mathscr{C}} = \boldsymbol{\tau}}$ *($F^{\mathscr{A}}_{\mathscr{C}}$ type translation)*

$$\begin{aligned}
(\boldsymbol{\tau}_1 \to \boldsymbol{\tau}_2)^{\mathscr{A}} &= \boldsymbol{\tau}_1^{\mathscr{A}} \xrightarrow{\mathsf{U}}\!\circ\, \boldsymbol{\tau}_2^{\mathscr{A}} & (\tau_1 \xrightarrow{\mathfrak{q}}\!\circ\, \tau_2)^{\mathscr{C}} &= \tau_1^{\mathscr{C}} \to \tau_2^{\mathscr{C}} \\
(\forall\, \boldsymbol{\alpha}.\, \boldsymbol{\tau})^{\mathscr{A}} &= \forall\, \beta^{\mathsf{U}}.(\{\{\beta^{\mathsf{U}}\}/\boldsymbol{\alpha}\}\boldsymbol{\tau})^{\mathscr{A}} & (\forall\, \alpha^{\mathfrak{q}}.\, \tau)^{\mathscr{C}} &= \forall\, \boldsymbol{\beta}.(\{\{\boldsymbol{\beta}\}/\alpha^{\mathfrak{q}}\}\tau)^{\mathscr{C}} \\
\boldsymbol{\alpha}^{\mathscr{A}} &= \{\boldsymbol{\alpha}\} & \rho^{\mathscr{C}} &= \{\rho\} \\
\{\rho\}^{\mathscr{A}} &= \rho & \{\boldsymbol{\alpha}\}^{\mathscr{C}} &= \boldsymbol{\alpha}
\end{aligned}$$

$\boxed{\mathrm{M};\boldsymbol{\Gamma} \vdash_{\mathscr{C}} \mathbf{e} : \boldsymbol{\tau}}$, $\boxed{\mathrm{M};\Gamma \vdash_{\mathscr{A}} e : \tau}$ *(additions to expression typing)*

$$\textsc{CT-ModA}\ \frac{(f : \tau = v) \in \mathrm{M}}{\mathrm{M};\boldsymbol{\Gamma} \vdash_{\mathscr{C}} f^{\mathbf{g}} : \tau^{\mathscr{C}}} \qquad \textsc{AT-ModC}\ \frac{(\mathbf{f} : \boldsymbol{\tau} = \mathbf{v}) \in \mathrm{M}}{\mathrm{M};\Gamma \vdash_{\mathscr{A}} \mathbf{f}^{g} : \boldsymbol{\tau}^{\mathscr{A}}} \qquad \textsc{AT-ModI}\ \frac{(\mathbf{f} :> \tau = \mathbf{f}') \in \mathrm{M}}{\mathrm{M};\Gamma \vdash_{\mathscr{A}} \mathbf{f}^{g} : \tau}$$

$\boxed{\mathrm{M} \vdash \mathrm{m}\ \mathrm{okay}}$, $\boxed{\vdash \mathrm{P} : \boldsymbol{\tau}}$ *($F^{\mathscr{A}}_{\mathscr{C}}$ module and program typing)*

$$\textsc{Interface}\ \frac{(\mathbf{g} : \tau^{\mathscr{C}} = \mathbf{v}) \in \mathrm{M} \qquad \mathrm{FTV}(\tau) = \emptyset \qquad \langle\tau\rangle = \mathsf{U}}{\mathrm{M} \vdash \mathbf{f} :> \tau = \mathbf{g}\ \mathrm{okay}}$$

$$\textsc{Prog}\ \frac{(\forall \mathrm{m} \in \mathrm{M})\, \mathrm{M} \vdash \mathrm{m}\ \mathrm{okay} \qquad \mathrm{M};\bullet \vdash_{\mathscr{C}} \mathbf{e} : \boldsymbol{\tau}}{\vdash \mathsf{let}\ \mathrm{M}\ \mathsf{in}\ \mathbf{e} : \boldsymbol{\tau}}$$

Figure 7.9: Static semantics of $\mathrm{F}^{\mathscr{A}}_{\mathscr{C}}$

Static semantics. To type check an $F^{\mathscr{A}}_{\mathscr{C}}$ program, we type check the $F^{\mathscr{A}}$ modules using the type system for $F^{\mathscr{A}}$, and the $F_{\mathscr{C}}$ modules and main expression using the type system for $F^{\mathscr{A}}$. To make this work, the individual language type systems need additional rules to handle interlanguage module references. The type system for $F^{\mathscr{A}}_{\mathscr{C}}$ appears in figure 7.9.

First, we add a case to the qualifier assignment metafunction $\langle\cdot\rangle$, which maps $F^{\mathscr{A}}$ types to qualifiers. The qualifier of an embedded $F_{\mathscr{C}}$ type variable ($\{\boldsymbol{\alpha}\}$) is U.

Next, we define the metafunctions $(\cdot)^{\mathscr{A}}$ and $(\cdot)^{\mathscr{C}}$, which map $F_{\mathscr{C}}$ types to $F^{\mathscr{A}}$ types and $F^{\mathscr{A}}$ types to $F_{\mathscr{C}}$ types, respectively:

- Function types convert to function types. $F_{\mathscr{C}}$ function types go to unlimited functions in $F^{\mathscr{A}}$, and both unlimited and affine $F^{\mathscr{A}}$ functions collapse to ordinary ($\rightarrow$) functions in $F_{\mathscr{C}}$.

- Quantified types map to quantified types, but they require renaming because we distinguish type variables between the two languages. In particular, $F^{\mathscr{A}}$ type variables carry qualifiers, which indicate whether they may be instantiated to any type or only to unlimited types. $F_{\mathscr{C}}$ type variables are renamed to unlimited $F^{\mathscr{A}}$ type variables, which means that using the default mapping, polymorphic $F_{\mathscr{C}}$ modules imported into $F^{\mathscr{A}}$ may only be instantiated with unlimited types. Using an interface module can overcome this restriction.

- Opaque types are embedded by the mapping, and types already embedded from the other language are unembedded by the mapping.

Note that the mapping from $F_{\mathscr{C}}$ to $F^{\mathscr{A}}$ is injective, but the reverse mapping is not because it collapses qualifiers.

The mixed language adds three new expression typing rules, one to $F_{\mathscr{C}}$ and two to $F^{\mathscr{A}}$, for typing interlanguage module invocations. When $F^{\mathscr{A}}$ module $f : \tau = v$ appears in an $F_{\mathscr{C}}$ expression, it is assigned $F_{\mathscr{C}}$ type $\tau^{\mathscr{C}}$. Likewise, when $F_{\mathscr{C}}$ module $\mathbf{f} : \boldsymbol{\tau} = \mathbf{v}$ appears in an $F^{\mathscr{A}}$ expression, it is assigned $F^{\mathscr{A}}$ type $\boldsymbol{\tau}^{\mathscr{A}}$. When an interface $\mathbf{f} :> \tau = \mathbf{g}$ is invoked from an $F^{\mathscr{A}}$ expression, it is given the declared $F^{\mathscr{A}}$ type τ rather than the translation of the $F_{\mathscr{C}}$ type of $\mathbf{g}$.

Finally, we give rules for typing interface modules and programs. An interface module $\mathbf{f} :> \tau = \mathbf{g}$ is admissible when τ is unlimited (as for $\mathrm{F}^{\mathscr{A}}$ modules) and $\mathbf{g}$ is an $\mathrm{F}_{\mathscr{C}}$ module of type $\tau^{\mathscr{C}}$. The mapping from $\mathrm{F}^{\mathscr{A}}$ types to $\mathrm{F}_{\mathscr{C}}$ is a projection that erases qualifiers, and the mapping from $\mathrm{F}_{\mathscr{C}}$ back to $\mathrm{F}^{\mathscr{A}}$ makes all qualifiers U. An interface module allows viewing the type of an $\mathrm{F}_{\mathscr{C}}$ module with qualifiers other than U, because given an $\mathrm{F}_{\mathscr{C}}$ module $\mathbf{g} : \boldsymbol{\tau}' = \mathbf{v}$, an interface module allows ascribing any type in the pre-image of $(\cdot)^{\mathscr{C}}$ that maps to $\boldsymbol{\tau}'$.

Operational semantics. We extend the syntax of the mixed language with several new forms (figure 7.10). Whereas the source syntax segregates the two sublanguages into separate modules, module invocation reduces to the body of the module, which leads expressions of both languages to nest at run time. Rather than allow $\mathrm{F}^{\mathscr{A}}$ terms to appear directly in $\mathrm{F}_{\mathscr{C}}$, and vice versa, we need a way to cordon off terms from one calculus embedded in the other and to ensure that the interaction is well-behaved. These new expression forms are called "boundaries." Whereas Matthews and Findler (2007) allow nested expressions in source programs and eschew modules, I follow Tobin-Hochstadt and Felleisen (2006) in segregating the two languages by module, and thus boundaries arise only at run time. While this is not necessary for soundness in their system, it is necessary in mine, because the type system for $\mathrm{F}_{\mathscr{C}}$ does not do context splitting as required by $\mathrm{F}^{\mathscr{A}}$. Because the goal is to integrate with an existing language, one of the design criteria is that the type system of the non-affine language must not be modified.

The new run-time syntax includes both boundary expressions $(\tau \underset{f\,\mathbf{g}}{\Leftarrow})\mathbf{e}$ for embedding $\mathrm{F}_{\mathscr{C}}$ expressions in $\mathrm{F}^{\mathscr{A}}$ and boundary expressions $(\underset{\mathbf{g}\,f}{\Leftarrow} \tau)e$ for embedding $\mathrm{F}^{\mathscr{A}}$ expressions in $\mathrm{F}_{\mathscr{C}}$. Each of these forms includes a type τ, written on the $\mathrm{F}^{\mathscr{A}}$ side, which represents a contract between the two modules that gave rise to the nested expression. Some contracts, for example $\forall\, \alpha^{\mathsf{U}}.\, \alpha^{\mathsf{U}} \overset{\mathsf{U}}{\multimap} \alpha^{\mathsf{U}}$, are fully enforced by both type systems. Others, such as $\forall\, \alpha^{\mathsf{U}}.\, \alpha^{\mathsf{U}} \overset{\mathsf{A}}{\multimap} \alpha^{\mathsf{U}}$, require dynamic checks. The type system guarantees that any value flowing into such a boundary will be a function, but this type also imposes an obligation on the negative party to apply the function at most once, which

$\mathbf{e}$	::=	$\cdots$	additional $\mathrm{F}_{\mathscr{C}}$ expressions
	|	$(\underset{\mathbf{f}g}{\Leftarrow}\tau)e$	boundary embedding $\mathrm{F}^{\mathscr{A}}$ expression
e	::=	$\cdots$	additional $\mathrm{F}^{\mathscr{A}}$ expressions
	|	$(\tau \underset{g\mathbf{f}}{\Leftarrow})\mathbf{e}$	boundary embedding $\mathrm{F}_{\mathscr{C}}$ expression
$\mathbf{v}$	::=	$\cdots$	additional $\mathrm{F}_{\mathscr{C}}$ values
	|	$(\underset{\mathbf{f}g}{\Leftarrow}\tau)^{\ell}\, v$	embedded, sealed $\mathrm{F}^{\mathscr{A}}$ value
	|	BLSSD	distinguished, unused seal value
	|	DFNCT	distinguished, used seal value
v	::=	$\cdots$	additional $\mathrm{F}^{\mathscr{A}}$ values
	|	$(\tau \underset{g\mathbf{f}}{\Leftarrow})^{\bullet}\, \mathbf{v}$	wrapped $\mathrm{F}_{\mathscr{C}}$ value
$\mathbf{E}$	::=	$\cdots$	additional $\mathrm{F}_{\mathscr{C}}$ evaluation contexts
	|	$(\underset{\mathbf{f}g}{\Leftarrow}\tau)E$	embedded $\mathrm{F}^{\mathscr{A}}$ evaluation
E	::=	$\cdots$	additional $\mathrm{F}^{\mathscr{A}}$ evaluation contexts
	|	$(\tau \underset{g\mathbf{f}}{\Leftarrow})\mathbf{E}$	embedded $\mathrm{F}_{\mathscr{C}}$ evaluation
C	::=		$\mathrm{F}^{\mathscr{A}}_{\mathscr{C}}$ configurations
	|	$(s, \mathbf{e})$	store and expression
	|	blame $\mathbf{f}$	blame
A	::=		$\mathrm{F}^{\mathscr{A}}_{\mathscr{C}}$ answers
	|	$(s, \mathbf{v})$	store and value
	|	blame $\mathbf{f}$	blame
s	::=		stores
	|	$\{\}$	the empty store
	|	$\{\ell \mapsto \mathbf{v}\}$	store with $\mathrm{F}_{\mathscr{C}}$ value
	|	$\{\ell \mapsto v\}$	store with $\mathrm{F}^{\mathscr{A}}$ value
	|	$s_1 \uplus s_2$	store concatenation

Figure 7.10: Operational semantics of $\mathrm{F}^{\mathscr{A}}_{\mathscr{C}}$ (i): run-time syntax

the $\mathrm{F}_{\mathscr{C}}$ type system cannot enforce. The arrow in a boundary expression has two subscripts representing the parties to the contract. The right subscript of a boundary is a module name in the inner language, representing the positive party to the contract: It promises that if the enclosed subexpression reduces to a value, then the value will obey contract τ. The left subscript is the negative party, which promises to treat the resulting value properly. In particular, if the contract is affine, then the negative party promises to use the resulting value at most once.

The remaining run-time syntax is best explained along with the relevant portions of the $\mathrm{F}^{\mathscr{A}}_{\mathscr{C}}$ reduction relation, which is defined in figure 7.11. As in $\mathrm{F}^{\mathscr{A}}$, reduction relates configurations that consist of a store and an expression, again parametrized by a module context M. Stores (s) may now include both $\mathrm{F}_{\mathscr{C}}$ and $\mathrm{F}^{\mathscr{A}}$ values.

Boundaries first arise when a module in one calculus refers to a module in the other calculus. When the name of an $\mathrm{F}^{\mathscr{A}}$ module appears in an $\mathrm{F}_{\mathscr{C}}$ term, rule C-MOD-A (figure 7.11) wraps the module name with an $\mathrm{F}^{\mathscr{A}}$-in-$\mathrm{F}_{\mathscr{C}}$ boundary, using the $\mathrm{F}^{\mathscr{A}}$ module's type as the contract. In the other direction, when an $\mathrm{F}_{\mathscr{C}}$ module appears in $\mathrm{F}^{\mathscr{A}}$, rule A-MOD-C wraps the module name in a boundary using the $\mathrm{F}^{\mathscr{A}}$ translation of the module's type $\boldsymbol{\tau}$ as the contract. For an interface module, the contract is the type declared by the interface, and the name of the interface is the positive party (rule A-MOD-I).

We add evaluation contexts for reduction under boundaries, which means it is now possible to construct an $\mathrm{F}_{\mathscr{C}}$ evaluation context with an $\mathrm{F}^{\mathscr{A}}$ hole, and vice versa. (When there is the possibility of ambiguity, I indicate the language of the hole: $\mathbf{E}[]_{\mathscr{C}}$ versus $\mathbf{E}[]_{\mathscr{A}}$ and $E[]_{\mathscr{C}}$ versus $E[]_{\mathscr{A}}$.) Rule CXT-A supports evaluating $\mathrm{F}^{\mathscr{A}}$ expressions embedded in the top-level $\mathrm{F}_{\mathscr{C}}$ expression of a configuration, whereas rule CXT-C reduces an $\mathrm{F}_{\mathscr{C}}$ expression in a configuration. When the expression under a boundary reduces to a value, it is time to apply the boundary's contract to the value. There are two possibilities:

- Functional values and opaque affine values must have their checks deferred: functions until the time of their application, and opaque values until they pass back into their native language. For deferred checks, we

$\boxed{C \xrightarrow[\mathrm{M}]{} C'}$ *($F^{\mathscr{A}}_{\mathscr{C}}$ reduction)*

(C-MOD-A) $(s, f^{\mathbf{g}}) \xrightarrow[\mathrm{M}]{} (s, (\underset{\mathbf{g}f}{\Leftarrow} \tau) f)$ when $(f : \tau = v) \in \mathrm{M}$

(A-MOD-C) $(s, \mathbf{g}^{f}) \xrightarrow[\mathrm{M}]{} (s, (\boldsymbol{\tau}^{\mathscr{A}} \underset{f\mathbf{g}}{\Leftarrow}) \mathbf{g})$ when $(\mathbf{g} : \boldsymbol{\tau} = \mathbf{v}) \in \mathrm{M}$

(A-MOD-I) $(s, \mathbf{g}^{f}) \xrightarrow[\mathrm{M}]{} (s, (\tau \underset{f\mathbf{g}}{\Leftarrow}) \mathbf{g}')$ when $(\mathbf{g} :> \tau = \mathbf{g}') \in \mathrm{M}$

(C-SEAL) $(s, (\underset{\mathbf{g}f}{\Leftarrow} \tau) v) \xrightarrow[\mathrm{M}]{} (s \uplus \{\ell \mapsto \mathrm{BLSSD}\}, (\underset{\mathbf{g}f}{\Leftarrow} \tau)^{\ell} v)$ when $v \neq (\{\boldsymbol{\alpha}\} \underset{f'\mathbf{g}'}{\Leftarrow})^{\bullet} \mathbf{v}'$

(A-WRAP) $(s, (\tau \underset{f\mathbf{g}}{\Leftarrow}) \mathbf{v}) \xrightarrow[\mathrm{M}]{} (s, (\tau \underset{f\mathbf{g}}{\Leftarrow})^{\bullet} \mathbf{v})$ when $\mathbf{v} \neq (\underset{\mathbf{g}'f'}{\Leftarrow} \rho)^{\ell} v'$

(C-UNWRAP) $(s, (\underset{\mathbf{g}f}{\Leftarrow} \tau)((\{\boldsymbol{\alpha}\} \underset{f'\mathbf{g}'}{\Leftarrow})^{\bullet} \mathbf{v})) \xrightarrow[\mathrm{M}]{} (s, \mathbf{v})$

(A-UNSEAL) $(s, (\tau \underset{f\mathbf{g}}{\Leftarrow})((\underset{\mathbf{g}'f'}{\Leftarrow} \rho)^{\ell} v)) \xrightarrow[\mathrm{M}]{} \mathit{check}(s, \ell, \langle \rho \rangle, v, \mathbf{g}')$

(C-B_{τ}-A) $(s, \big((\underset{\mathbf{g}f}{\Leftarrow} \forall\, \alpha^{\mathsf{q}}.\, \tau')^{\ell} v\big)\, \boldsymbol{\tau}) \xrightarrow[\mathrm{M}]{} \mathit{check}(s, \ell, \langle \tau' \rangle, (\underset{\mathbf{g}f}{\Leftarrow} \{\boldsymbol{\tau}^{\mathscr{A}}/\alpha^{\mathsf{q}}\}\tau)(v\, \boldsymbol{\tau}^{\mathscr{A}}), \mathbf{g})$

(C-β_v-A) $(s, \big((\underset{\mathbf{g}f}{\Leftarrow} \tau_1 \xrightarrow{\mathsf{q}} \tau_2)^{\ell} v\big)\, \mathbf{v}') \xrightarrow[\mathrm{M}]{} \mathit{check}(s, \ell, \mathsf{q}, (\underset{\mathbf{g}f}{\Leftarrow} \tau_2)\big(v\,((\tau_1 \underset{f\mathbf{g}}{\Leftarrow}) \mathbf{v}')\big), \mathbf{g})$

(A-B_{τ}-C) $(s, \big((\forall\, \alpha^{\mathsf{q}}.\, \tau \underset{f\mathbf{g}}{\Leftarrow})^{\bullet} \mathbf{v}\big)\, \tau') \xrightarrow[\mathrm{M}]{} (s, (\{\tau'/\alpha^{\mathsf{q}}\}\tau \underset{f\mathbf{g}}{\Leftarrow})(\mathbf{v}\, \tau^{\mathscr{C}}))$

(A-β_v-C) $(s, \big((\tau_1 \xrightarrow{\mathsf{q}} \tau_2 \underset{f\mathbf{g}}{\Leftarrow})^{\bullet} \mathbf{v}\big)\, v') \xrightarrow[\mathrm{M}]{} (s, (\tau_2 \underset{f\mathbf{g}}{\Leftarrow})\big(\mathbf{v}((\underset{\mathbf{g}f}{\Leftarrow} \tau_1) v')\big))$

$$\mathit{check}(s, \ell, \mathsf{q}, e, \mathbf{g}) = \begin{cases} (s, e) & \text{if } \mathsf{q} = \mathsf{U} \\ (s' \uplus \{\ell \mapsto \mathrm{DFNCT}\}, e) & \text{if } s = s' \uplus \{\ell \mapsto \mathrm{BLSSD}\} \\ \mathsf{blame}\, \mathbf{g} & \text{otherwise} \end{cases}$$

$\boxed{C \xmapsto[\mathrm{M}]{} C'}$ *($F^{\mathscr{A}}_{\mathscr{C}}$ reduction)*

(CXT-C) $$\frac{(s, \mathbf{e}) \xrightarrow[\mathrm{M}]{} (s', \mathbf{e}')}{(s, \mathbf{E}[\mathbf{e}]) \xmapsto[\mathrm{M}]{} (s', \mathbf{E}[\mathbf{e}'])}$$

(CXT-A) $$\frac{(s, e) \xrightarrow[\mathrm{M}]{} (s', e')}{(s, \mathbf{E}[e]) \xmapsto[\mathrm{M}]{} (s', \mathbf{E}[e'])}$$

Figure 7.11: Operational semantics of $\mathrm{F}^{\mathscr{A}}_{\mathscr{C}}$ (ii): reduction

leave the value in a "sealed" boundary $(\underset{\mathbf{g}f}{\Leftarrow} \tau)^{\ell}\, v$ or a "wrapped" boundary $(\tau \underset{f\mathbf{g}}{\Leftarrow})^{\bullet}\, \mathbf{v}$, both of which are themselves value forms. (An $\mathrm{F}^{\mathscr{A}}$ value is sealed, which tracks its usage, whereas an $\mathrm{F}_{\mathscr{C}}$ value is wrapped, which merely indicates that it is not a redex and that its contract checking is delayed.)

- When a previously sealed or wrapped opaque value reaches a boundary back to its original language, both that boundary and the sealed or wrapped boundary are discarded. For a sealed boundary, there is also a check that the seal has not previously been opened.

Rule C-SEAL implements contract application for $\mathrm{F}^{\mathscr{A}}$ values embedded in $\mathrm{F}_{\mathscr{C}}$ expressions. The rule requires that the $\mathrm{F}^{\mathscr{A}}$ value v is not, in fact, an opaque $\mathrm{F}_{\mathscr{C}}$ value (which is handled by rule C-UNWRAP). In that case, rule C-SEAL seals and *blesses* an $\mathrm{F}^{\mathscr{A}}$ value, by allocating a location ℓ, to which it stores a distinguished value BLSSD; it adds this location to the boundary, which marks the sealed value as not yet used. (The run-time syntax in figure 7.10 adds two distinguished values, BLSSD and DFNCT, to the values of $\mathrm{F}_{\mathscr{C}}$. These values are used to track resource usage, and while they need to be distinguishable from one another, they need not be distinct from other $\mathrm{F}_{\mathscr{C}}$ values.)

Rule A-WRAP implements contract application for $\mathrm{F}_{\mathscr{C}}$ values embedded in $\mathrm{F}^{\mathscr{A}}$ expressions. Like rule C-SEAL, this rule requires that the value to be wrapped, $\mathbf{v}$, not already be a wrapped opaque $\mathrm{F}^{\mathscr{A}}$ value. In that case, rule A-WRAP merely marks the value as being wrapped. No seal location needs to be allocated because $\mathrm{F}_{\mathscr{C}}$ values are always unlimited.

Unwrapping happens under three circumstances: when a wrapped or sealed opaque value flows into a boundary back to its native language, when a wrapped or sealed abstraction is applied to a value, or when a wrapped or sealed type abstraction is applied to a type. The first case is handled by rules C-UNWRAP and A-UNSEAL. These rules reduce a boundary expression in cases complementary to rules C-SEAL and A-WRAP, when the value flowing into the boundary is itself a wrapped $\mathrm{F}_{\mathscr{C}}$ value or sealed $\mathrm{F}^{\mathscr{A}}$ value. In the wrapped case (rule C-UNWRAP), the boundaries are simply discarded. In the sealed case, the boundaries are removed and the seal must be checked. This

step is specified by metafunction *check*, which has three cases. Unlimited values are unsealed with no check. If an affine value remains blessed, *check* updates the store to mark it "defunct" and returns the unsealed value. If, on the other hand, there is an attempt to unseal a defunct affine value, *check* blames the negative party. This is the key dynamic check that enforces the affine invariant for non-functional values.

Rules C-B$_\tau$-A, C-β_v-A, A-B$_\tau$-C, and A-β_v-C all handle sealed abstractions, which are unsealed when they are applied. For application expressions, this follows the technique of Findler and Felleisen (2002), whereby the argument is coerced to the domain of the contract with the blame labels reversed, and the result is coerced to the codomain of the contract. For type applications, the argument, which is a type, is converted using the interlanguage type maps $(\cdot)^{\mathscr{A}}$ and $(\cdot)^{\mathscr{C}}$, rather than coerced with a contract, and the result of the type application is coerced as usual. For sealed $\mathrm{F}^{\mathscr{A}}$ abstractions, the seal location ℓ must be checked, again using metafunction *check*, to ensure that an affine function or type abstraction is not unsealed and applied more than once. This is the dynamic check that enforces the affine invariant for functions and type abstractions.

7.3 Type Soundness for $\mathrm{F}^{\mathscr{A}}_{\mathscr{C}}$

The presence of strong updates in $\mathrm{F}^{\mathscr{A}}$ (rule AT-SWAP in figure 7.7) means that aliasing a location can result in a program getting "stuck": if an aliased location is updated at a different type, reading from the alias produces a value of unexpected type. $\mathrm{F}^{\mathscr{A}}$'s type system prevents this, but adding $\mathrm{F}_{\mathscr{C}}$ means that an $\mathrm{F}^{\mathscr{A}}$ value may be aliased outside $\mathrm{F}^{\mathscr{A}}$. The soundness criterion is that no program that gets stuck is assigned a type. This means, in particular, all aliasing of affine values is either prevented by $\mathrm{F}^{\mathscr{A}}$'s type system or detected by a contract at run time.

To prove a Wright-Felleisen–style type soundness theorem (1994) requires identifying precisely what property is preserved by subject reduction. I use an internal type system to track which portions of the store are reachable from $\mathrm{F}^{\mathscr{A}}$ values that have flowed into $\mathrm{F}_{\mathscr{C}}$. Under this type system, configurations

enjoy standard progress and preservation, which allows me to state and prove a syntactic type soundness theorem using the internal type system's configuration typing judgment.

Conventions. I define the free variables of an expression **e**, written FV(**e**) inductively in the conventional way (and likewise for $F^{\mathscr{A}}$); however, I consider the module names in a program to be syntactically distinct from the λ- and let-bound variables, and I take the free variables to exclude module names.

The free locations of an expression **e**, written FL(**e**), is the set of locations ℓ that occur in **e** (and likewise for $F^{\mathscr{A}}$). Note that there are no binders for locations at the expression level.

Note also that exchange and weakening of store and typing contexts is implicit in the type system, by inspection of the type rules: All variable, type variable, and location lookup rules look anywhere in the environment, and ignore the rest. Conversely, it should be apparent that any assumption in an environment that is not free in the subject is not needed to type the subject, which justifies in discarding such assumptions.

7.3.1 The Internal Type System

Figure 7.12 shows the new syntax for the internal type system. A store type (Σ) maps locations to types in either sublanguage, or to "protected" types of the form $[\tau]^{\ell'}$. A location ℓ mapped to a protected type $[\tau]^{\ell'}$ means that location ℓ may appear only under $F^{\mathscr{A}}$-in-$F_{\mathscr{C}}$ boundaries sealed by ℓ'. The store-type–splitting relation ($\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2$) defined in figure 7.12 allows protected locations to be duplicated arbitrarily; but as we will see, they can only be used to type locations in terms that are protected by a contract. Store splitting also duplicates locations containing $F_{\mathscr{C}}$ values, but it requires locations containing $F^{\mathscr{A}}$ values, both unlimited and affine, to go only one way or the other. This ensures that such locations appear only once in a well-typed term, which ensures the safety of strong updates.

Figure 7.12 also defines store typing. The type of a store contains the types of all its locations. Additionally, each location ℓ containing an $F^{\mathscr{A}}$ value of type

$$
\begin{array}{lll}
\Sigma & ::= & \text{store types} \\
& \mid \ \bullet & \quad \text{the empty store type} \\
& \mid \ \Sigma, \ell : \mathbf{\tau} & \quad \text{with an } \mathrm{F}_{\mathscr{C}} \text{ location} \\
& \mid \ \Sigma, \ell : \tau & \quad \text{with an } \mathrm{F}^{\mathscr{A}} \text{ location} \\
& \mid \ \Sigma, \ell : [\tau]^{\ell'} & \quad \text{with a protected } \mathrm{F}^{\mathscr{A}} \text{ location}
\end{array}
$$

$\boxed{\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2}$ *(store type splitting)*

$$\textsc{RSplit-Nil}\ \frac{}{\bullet \rightsquigarrow \bullet \boxplus \bullet} \qquad \textsc{RSplit-ConsAL}\ \frac{\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2}{\Sigma, \ell : \tau \rightsquigarrow \Sigma_1, \ell : \tau \boxplus \Sigma_2} \qquad \textsc{RSplit-ConsAR}\ \frac{\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2}{\Sigma, \ell : \tau \rightsquigarrow \Sigma_1 \boxplus \Sigma_2, \ell : \tau}$$

$$\textsc{RSplit-ConsAProt}\ \frac{\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2}{\Sigma, \ell : [\tau]^{\ell'} \rightsquigarrow \Sigma_1, \ell : [\tau]^{\ell'} \boxplus \Sigma_2, \ell : [\tau]^{\ell'}} \qquad \textsc{RSplit-ConsC}\ \frac{\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2}{\Sigma, \ell : \mathbf{\tau} \rightsquigarrow \Sigma_1, \ell : \mathbf{\tau} \boxplus \Sigma_2, \ell : \mathbf{\tau}}$$

$\boxed{\mathrm{M}; \Sigma \rhd s : \Sigma'}$ *(store typing)*

$$\textsc{RS-Nil}\ \frac{}{\mathrm{M}; \Sigma' \rhd \{\} : \bullet} \qquad \textsc{RS-LocC}\ \frac{\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2 \qquad \mathrm{M}; \Sigma_1 \rhd s : \Sigma' \qquad \mathrm{M}; \Sigma_2; \bullet \rhd_{\mathscr{C}} \mathbf{v} : \mathbf{\tau}}{\mathrm{M}; \Sigma \rhd s \uplus \{\ell \mapsto \mathbf{v}\} : (\Sigma', \ell : \mathbf{\tau})}$$

$$\textsc{RS-LocA}\ \frac{\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2 \qquad \mathrm{M}; \Sigma_1 \rhd s : \Sigma' \qquad \mathrm{M}; \Sigma_2; \bullet \rhd_{\mathscr{A}} v : \tau}{\mathrm{M}; \Sigma \rhd s \uplus \{\ell \mapsto v\} : (\Sigma', \ell : \tau)} \qquad \textsc{RS-LocAProt}\ \frac{\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2 \qquad \mathrm{M}; \Sigma_1 \rhd s : \Sigma' \qquad \mathrm{M}; \Sigma_2; \bullet \rhd_{\mathscr{A}} v : \tau}{\mathrm{M}; \Sigma \rhd s \uplus \{\ell \mapsto v\} : (\Sigma', \ell : [\tau]^{\ell'})}$$

$\boxed{[\Sigma]^{\ell} = \Sigma'}$ *(store type protection)*

$$[\bullet]^{\ell} = \bullet \qquad [\Sigma, \ell' : \tau]^{\ell} = [\Sigma]^{\ell}, \ell' : [\tau]^{\ell} \qquad [\Sigma, \ell' : [\tau]^{\ell''}]^{\ell} = [\Sigma]^{\ell}, \ell' : [\tau]^{\ell''}$$

$$[\Sigma, \ell' : \mathbf{\tau}]^{\ell} = [\Sigma]^{\ell}, \ell' : \mathbf{\tau}$$

$\boxed{\langle \Sigma \rangle = \mathsf{q}}$ *(qualifier assignment for store types)*

$$\langle \bullet \rangle = \mathsf{U} \qquad \langle \Sigma, \ell : \tau \rangle = \mathsf{A} \qquad \langle \Sigma, \ell : [\tau]^{\ell'} \rangle = \langle \Sigma \rangle \qquad \langle \Sigma, \ell : \mathbf{\tau} \rangle = \langle \Sigma \rangle$$

Figure 7.12: Internal type system for $\mathrm{F}^{\mathscr{A}}_{\mathscr{C}}$ (i): store types

$\boxed{\boldsymbol{\tau} \in \mathbf{W}}$, $\boxed{\tau \in W}$ *(wrappable types)*

$$\mathbf{W} = \{\, \boldsymbol{\tau} \mid \neg(\exists \rho)\, \boldsymbol{\tau} = \{\rho\} \,\} \qquad W = \{\, \tau \mid \neg(\exists \boldsymbol{\alpha})\, \tau = \{\boldsymbol{\alpha}\} \,\}$$

$\boxed{\mathrm{M}; \Sigma; \boldsymbol{\Gamma} \rhd_{\mathscr{C}} \mathbf{e} : \boldsymbol{\tau}}$ *(internal $F_{\mathscr{C}}$ expression typing)*

$$\textsc{RCT-Boundary}\ \frac{\mathrm{M}; \Sigma; \bullet \rhd_{\mathscr{A}} e : \tau \qquad \mathrm{FTV}(\tau) = \emptyset}{\mathrm{M}; \Sigma; \boldsymbol{\Gamma} \rhd_{\mathscr{C}} (\underset{\mathbf{g}f}{\Leftarrow} \tau) e : \tau^{\mathscr{C}}}$$

$$\textsc{RCT-Sealed}\ \frac{\mathrm{M}; \Sigma; \bullet \rhd_{\mathscr{A}} v : \tau \qquad \mathrm{FTV}(\tau) = \emptyset \qquad \langle \tau \rangle = \mathsf{U} \qquad \tau \in W}{\mathrm{M}; \Sigma; \boldsymbol{\Gamma} \rhd_{\mathscr{C}} (\underset{\mathbf{g}f}{\Leftarrow} \tau)^{\ell} v : \tau^{\mathscr{C}}}$$

$$\textsc{RCT-Blessed}\ \frac{\mathrm{M}; \Sigma_1, \Sigma_2; \bullet \rhd_{\mathscr{A}} v : \tau \qquad \mathrm{FTV}(\tau) = \emptyset \qquad \langle \tau \rangle = \mathsf{A} \qquad \tau \in W}{\mathrm{M}; [\Sigma_1]^{\ell}, \ell : \mathbb{B}, [\Sigma_2]^{\ell}; \boldsymbol{\Gamma} \rhd_{\mathscr{C}} (\underset{\mathbf{g}f}{\Leftarrow} \tau)^{\ell} v : \tau^{\mathscr{C}}}$$

$$\textsc{RCT-Defunct}\ \frac{\mathrm{FTV}(\tau) = \emptyset \qquad \langle \tau \rangle = \mathsf{A} \qquad \tau \in W}{\mathrm{M}; [\Sigma_1]^{\ell}, \ell : \mathbb{D}, [\Sigma_2]^{\ell}; \boldsymbol{\Gamma} \rhd_{\mathscr{C}} (\underset{\mathbf{g}f}{\Leftarrow} \tau)^{\ell} v : \tau^{\mathscr{C}}}$$

$$\textsc{RCT-BlessedVal}\ \frac{}{\mathrm{M}; \Sigma; \boldsymbol{\Gamma} \rhd_{\mathscr{C}} \textsc{blssd} : \mathbb{B}} \qquad \textsc{RCT-DefunctVal}\ \frac{}{\mathrm{M}; \Sigma; \boldsymbol{\Gamma} \rhd_{\mathscr{C}} \textsc{dfnct} : \mathbb{D}}$$

$\boxed{\mathrm{M}; \Sigma; \Gamma \rhd_{\mathscr{A}} e : \tau}$ *(internal $F^{\mathscr{A}}$ expression typing)*

$$\textsc{RAT-Loc}\ \frac{\ell : \tau \in \Sigma}{\mathrm{M}; \Sigma; \Gamma \rhd_{\mathscr{A}} \ell : \mathsf{ref}\ \tau}$$

$$\textsc{RAT-Boundary}\ \frac{\mathrm{M}; \Sigma; \bullet \rhd_{\mathscr{C}} \mathbf{e} : \tau^{\mathscr{C}} \qquad \mathrm{FTV}(\tau) = \emptyset}{\mathrm{M}; \Sigma; \Gamma \rhd_{\mathscr{A}} (\tau \underset{f\mathbf{g}}{\Leftarrow}) \mathbf{e} : \tau}$$

$$\textsc{RAT-Wrapped}\ \frac{\mathrm{M}; \Sigma; \bullet \rhd_{\mathscr{C}} \mathbf{v} : \tau^{\mathscr{C}} \qquad \mathrm{FTV}(\tau) = \emptyset \qquad \tau^{\mathscr{C}} \in \mathbf{W}}{\mathrm{M}; \Sigma; \Gamma \rhd_{\mathscr{A}} (\tau \underset{f\mathbf{g}}{\Leftarrow})^{\bullet} \mathbf{v} : \tau}$$

Figure 7.13: Internal type system for $F^{\mathscr{A}}_{\mathscr{C}}$ (ii): new expressions

τ may appear in the store type, non-deterministically, in bare form $(\ell : \tau)$ or protected $(\ell : [\tau]^{\ell'})$ by any location ℓ'.

Several typing rules rely on protecting a whole store, written $[\Sigma]^{\ell}$, which protects any bare $F^{\mathscr{A}}$ locations in Σ by location ℓ. We also define the qualifier of a store type: If it maps any location to a bare $F^{\mathscr{A}}$ type, then the qualifier is A; otherwise it is U.

Typing new expression forms. In figure 7.13, we define sets of *wrappable types* $\mathbf{W}$ and W, which for each language include all types except for foreign types imported from the other language ($\{\rho\}$ and $\{\boldsymbol{\alpha}\}$). These sets of types are used in several new expression typing judgments.

The new expression type judgments are $\mathrm{M};\Sigma;\mathbf{\Gamma} \rhd_{\mathscr{C}} \mathbf{e} : \boldsymbol{\tau}$ and $\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} e : \tau$ (figure 7.13). These add a store type to the context, which is used to type locations that appear in run-time expressions, by rule RAT-LOC. Boundary expressions are typed by rules RCT-BOUNDARY and RAT-BOUNDARY, each of which requires the $\mathrm{F}^{\mathscr{A}}$ type τ in the premise (resp., conclusion) to convert to the $\mathrm{F}_{\mathscr{C}}$ type $\tau^{\mathscr{C}}$ in the conclusion (resp., premise). Also, notably, both drop the typing contexts $\mathbf{\Gamma}$ (resp., Γ) in the premise; this is because the scope of bound variables (not modules) is always within a single language. Rule RAT-WRAPPED is used to type wrapped $\mathrm{F}_{\mathscr{C}}$-in-$\mathrm{F}^{\mathscr{A}}$ boundaries in the same manner as RAT-BOUNDARY. Wrapped $\mathrm{F}_{\mathscr{C}}$ values must have a wrappable $\mathrm{F}_{\mathscr{C}}$ type in $\mathbf{W}$.

Three rules are used to type sealed $\mathrm{F}^{\mathscr{A}}$ values. By all three rules, sealed $\mathrm{F}^{\mathscr{A}}$ values must have a wrappable $\mathrm{F}^{\mathscr{A}}$ type in W. For unlimited values, rule RCT-SEALED is substantially the same as rule RCT-BOUNDARY. For sealed boundaries $(\underset{\mathbf{g}f}{\Leftarrow} \tau)^{\ell}\, v$ that contain values of affine type, either rule RCT-BLESSED or rule RCT-DEFUNCT applies, depending on the type of the $\mathrm{F}_{\mathscr{C}}$ value stored at the seal location ℓ. In particular, we assume distinct types $\mathbb{B}$ and $\mathbb{D}$ for the special seal values BLSSD and DFNCT (by rules RCT-BLESSEDVAL and RCT-DEFUNCTVAL). If location ℓ maps to $\mathbb{B}$—that is, it contains BLSSD—then we expose any locations protected by that same location ℓ when typing the value v that appears under the seal. This means that when a sealed, affine value is duplicated by $\mathrm{F}_{\mathscr{C}}$, all locations appearing in that value may still type, provided they *all* remain sealed. When one instance of the sealed value is unwrapped, location ℓ is updated to have type $\mathbb{D}$, which means that we no longer attempt to type other instances of the sealed value at all, and just give them the type indicated by the boundary. This is safe because the contract checking in the operational semantics ensures that such values can never be unsealed.

$\boxed{\mathrm{M} \rhd C : \tau}$ *($F^{\mathscr{A}}_{\mathscr{C}}$ configuration typing)*

$$\textsc{RConf}\ \frac{(\forall \mathrm{m} \in \mathrm{M})\ \mathrm{M} \vdash \mathrm{m}\ \mathrm{okay} \qquad \mathrm{M}; \Sigma_1 \rhd s : \Sigma \qquad \Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2 \qquad \mathrm{M}; \Sigma_2; \bullet \rhd_{\mathscr{C}} \mathbf{e} : \tau}{\mathrm{M} \rhd (s, \mathbf{e}) : \tau} \qquad \textsc{RBlame}\ \frac{}{\mathrm{M} \rhd \mathsf{blame}\, \mathbf{f} : \tau}$$

Figure 7.14: Internal type system for $\mathrm{F}^{\mathscr{A}}_{\mathscr{C}}$ (iii): configurations

$\boxed{\mathrm{M}; \Sigma; \Gamma \rhd_{\mathscr{C}} \mathbf{e} : \tau}$ *(internal $F_{\mathscr{C}}$ expression typing)*

$$\textsc{RCT-Var}\ \frac{\mathbf{x} : \tau \in \Gamma}{\mathrm{M}; \Sigma; \Gamma \rhd_{\mathscr{C}} \mathbf{x} : \tau} \qquad \textsc{RCT-Mod}\ \frac{(\mathbf{f} : \tau = \mathbf{v}) \in \mathrm{M}}{\mathrm{M}; \Sigma; \Gamma \rhd_{\mathscr{C}} \mathbf{f} : \tau} \qquad \textsc{RCT-ModA}\ \frac{(f : \tau = v) \in \mathrm{M}}{\mathrm{M}; \Sigma; \Gamma \rhd_{\mathscr{C}} f^{\mathbf{g}} : \tau^{\mathscr{C}}}$$

$$\textsc{RCT-TAbs}\ \frac{\mathrm{M}; \Sigma; \Gamma \rhd_{\mathscr{C}} \mathbf{v} : \tau}{\mathrm{M}; \Sigma; \Gamma \rhd_{\mathscr{C}} \Lambda\boldsymbol{\alpha}.\mathbf{v} : \forall\boldsymbol{\alpha}.\tau} \qquad \textsc{RCT-Abs}\ \frac{\mathrm{M}; \Sigma; \Gamma, \mathbf{x} : \tau \rhd_{\mathscr{C}} \mathbf{e} : \tau' \qquad \langle \Sigma|_{\mathrm{FL}(\lambda \mathbf{x}:\tau.\mathbf{e})} \rangle = \mathsf{U}}{\mathrm{M}; \Sigma; \Gamma \rhd_{\mathscr{C}} \lambda \mathbf{x}{:}\tau.\mathbf{e} : \tau \to \tau'}$$

$$\textsc{RCT-TApp}\ \frac{\mathrm{M}; \Sigma; \Gamma \rhd_{\mathscr{C}} \mathbf{e} : \forall\boldsymbol{\alpha}.\tau'}{\mathrm{M}; \Sigma; \Gamma \rhd_{\mathscr{C}} \mathbf{e}\,\tau : \{\tau/\boldsymbol{\alpha}\}\tau'} \qquad \textsc{RCT-App}\ \frac{\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2 \qquad \mathrm{M}; \Sigma_1; \Gamma \rhd_{\mathscr{C}} \mathbf{e}_1 : \tau' \to \tau \qquad \mathrm{M}; \Sigma_2; \Gamma \rhd_{\mathscr{C}} \mathbf{e}_2 : \tau'}{\mathrm{M}; \Sigma; \Gamma \rhd_{\mathscr{C}} \mathbf{e}_1\,\mathbf{e}_2 : \tau}$$

Figure 7.15: Internal type system for $\mathrm{F}^{\mathscr{A}}_{\mathscr{C}}$ (iv): old $\mathrm{F}_{\mathscr{C}}$ expressions

Typing configurations. Figure 7.14 gives the type rule for configurations. It requires that the store s have some type $\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2$, where Σ_1 is used to type the store (since values in the store can contain locations), and Σ_2 is used to type the configuration's expression $\mathbf{e}$. Additionally, blame configurations may be given any type.

Updated expression typing. Figures 7.15 and 7.16 update the type rules for the remaining expression forms for the internal type system. These rules extend each of the old rules with a store context, which is split for

$\boxed{\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} e : \tau}$ *(internal $F^{\mathscr{A}}$ expression typing)*

$$\textsc{Rat-Subsume}\ \frac{\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} e : \tau' \qquad \tau' <: \tau}{\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} e : \tau} \qquad \textsc{Rat-Var}\ \frac{x : \tau \in \Gamma}{\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} x : \tau}$$

$$\textsc{Rat-Mod}\ \frac{(f : \tau = v) \in \mathrm{M}}{\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} f : \tau} \qquad \textsc{Rat-ModC}\ \frac{(\mathbf{f} : \boldsymbol{\tau} = \mathbf{v}) \in \mathrm{M}}{\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} \mathbf{f}^{g} : \boldsymbol{\tau}^{\mathscr{A}}} \qquad \textsc{Rat-ModI}\ \frac{(\mathbf{f} :> \tau = \mathbf{f}') \in \mathrm{M}}{\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} \mathbf{f}^{g} : \tau}$$

$$\textsc{Rat-TAbs}\ \frac{\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} v : \tau}{\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} \Lambda\alpha^{\mathfrak{q}}.v : \forall\, \alpha^{\mathfrak{q}}.\tau} \qquad \textsc{Rat-Abs}\ \frac{\mathrm{M};\Sigma;\Gamma, x : \tau \rhd_{\mathscr{A}} e : \tau' \qquad \langle \Gamma|_{\mathrm{FV}(\lambda x{:}\tau.e)} \rangle \sqcup \langle \Sigma|_{\mathrm{FL}(\lambda x{:}\tau.e)} \rangle = \mathfrak{q}}{\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} \lambda x{:}\tau.e : \tau \xrightarrow{\mathfrak{q}} \tau'}$$

$$\textsc{Rat-Pair}\ \frac{\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2 \qquad \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \mathrm{M};\Sigma_1;\Gamma_1 \rhd_{\mathscr{A}} v_1 : \tau_1 \qquad \mathrm{M};\Sigma_2;\Gamma_2 \rhd_{\mathscr{A}} v_2 : \tau_2}{\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} \langle v_1, v_2 \rangle : \tau_1 \otimes \tau_2} \qquad \textsc{Rat-TApp}\ \frac{\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} e : \forall\, \alpha^{\mathfrak{q}}.\tau' \qquad \langle \tau \rangle \sqsubseteq \mathfrak{q}}{\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} e\,\tau : \{\tau/\alpha^{\mathfrak{q}}\}\tau'}$$

$$\textsc{Rat-App}\ \frac{\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2 \qquad \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \mathrm{M};\Sigma_1;\Gamma_1 \rhd_{\mathscr{A}} e_1 : \tau' \xrightarrow{\mathfrak{q}} \tau \qquad \mathrm{M};\Sigma_2;\Gamma_2 \rhd_{\mathscr{A}} e_2 : \tau'}{\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} e_1\, e_2 : \tau} \qquad \textsc{Rat-LetPair}\ \frac{\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2 \qquad \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \mathrm{M};\Sigma_1;\Gamma_1 \rhd_{\mathscr{A}} e_1 : \tau_1 \otimes \tau_2 \qquad \mathrm{M};\Sigma_2;\Gamma_2, x : \tau_1, y : \tau_2 \rhd_{\mathscr{A}} e_2 : \tau}{\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} \mathsf{let}\,\langle x, y \rangle = e_1\,\mathsf{in}\,e_2 : \tau}$$

$$\textsc{Rat-New}\ \frac{\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} e : \tau}{\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} \mathsf{new}\, e : \mathsf{ref}\, \tau} \qquad \textsc{Rat-Swap}\ \frac{\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} e : \mathsf{ref}\, \tau_1 \otimes \tau_2}{\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} \mathsf{swap}\, e : \mathsf{ref}\, \tau_2 \otimes \tau_1}$$

Figure 7.16: Internal type system for $\mathrm{F}^{\mathscr{A}}_{\mathscr{C}}$ (v): old $\mathrm{F}^{\mathscr{A}}$ expressions

multiplicative forms such as application in $F_{\mathscr{C}}$ as well as $F^{\mathscr{A}}$. The only other change is for typing λ abstractions. For $F^{\mathscr{A}}$ (rule RAT-ABS), we use not only the value context but the store context to determine the qualifier q in the resulting function type. For $F_{\mathscr{C}}$, rule RCT-ABS requires that the term contain no unprotected locations containing $F^{\mathscr{A}}$ values.

External typing implies internal typing. Most of the soundness proof concerns the internal type system. However, because we care about checking programs (not run-time configurations) with the external type system, it is important to relate the two:

LEMMA 7.1 (Equivalence of expression typing)**.**

If an expression types in the external type system ($\vdash$), then it types in the internal type system ($\rhd$) with empty store type:

1. *If* $\mathrm{M};\mathbf{\Gamma} \vdash_{\mathscr{C}} \mathbf{e} : \boldsymbol{\tau}$ *then* $\mathrm{M};\bullet;\mathbf{\Gamma} \rhd_{\mathscr{C}} \mathbf{e} : \boldsymbol{\tau}$.

2. *If* $\mathrm{M};\Gamma \vdash_{\mathscr{A}} e : \tau$ *then* $\mathrm{M};\bullet;\Gamma \rhd_{\mathscr{A}} e : \tau$.

Proof. By a simple induction on the structure of the external typing derivation. □

COROLLARY 7.2 (Programs to configurations)**.**

If $\vdash \mathsf{let}\ \mathrm{M}\ \mathsf{in}\ \mathbf{e} : \boldsymbol{\tau}$ *then* $\mathrm{M} \rhd (\{\}, \mathbf{e}) : \boldsymbol{\tau}$

Proof. By inversion of rule PROG, all modules in M are okay. Furthermore, $\mathrm{M};\bullet \vdash_{\mathscr{C}} \mathbf{e} : \boldsymbol{\tau}$, and by lemma 7.1, $\mathrm{M};\bullet;\bullet \rhd_{\mathscr{C}} \mathbf{e} : \boldsymbol{\tau}$. Since $s = \{\}$, $\Sigma = \bullet$, and thus, $\mathrm{M};\bullet \rhd s : \bullet$. Thus, by rule RCONF, $\mathrm{M} \rhd (\{\}, \mathbf{e}) : \boldsymbol{\tau}$. □

7.3.2 Syntactic Type Soundness

In this section, I prove a type soundness theorem. Most of the development is standard. I give a high-level overview of the proof here and include additional details in appendix B.

Preservation. The first portion of the proof leads to a preservation lemma, which relies on several lemmas about substitution and contexts. I begin with a definition of well-behaved type substitutions, which must not substitute a type with a higher qualifier for a type variable of a lower qualifier, and then show that well-behaved type substitutions respect type derivations.

DEFINITION 7.3 (Type substitutions and respecting qualifiers)**.**

*A **type substitution** θ is either the substitution of an $F^{\mathscr{A}}$ type for an $F^{\mathscr{A}}$ type variable ($\{\tau/\alpha^{\mathfrak{q}}\}$), the substitution of an $F_{\mathscr{C}}$ type for an $F_{\mathscr{C}}$ type variable ($\{\boldsymbol{\tau}/\boldsymbol{\alpha}\}$), or some composition thereof.*

*A type substitution **respects qualifiers** if one of the following holds:*

- *It is an $F^{\mathscr{A}}$ substitution $\{\tau/\alpha^{\mathfrak{q}}\}$ where $\langle\tau\rangle \sqsubseteq \mathfrak{q}$;*
- *It is an $F_{\mathscr{C}}$ substitution $\{\boldsymbol{\tau}/\boldsymbol{\alpha}\}$; or*
- *It is a composition of qualifier-respecting substitutions.*

LEMMA 7.4 (Type substitution on expressions preserves types)**.**

For any qualifier-respecting type substitution θ and any Σ such that $\mathrm{FTV}(\Sigma) = \emptyset$:

1. *If* $\mathrm{M};\Sigma;\boldsymbol{\Gamma} \rhd_{\mathscr{C}} \mathbf{e} : \boldsymbol{\tau}$ *then* $\mathrm{M};\Sigma;\theta\boldsymbol{\Gamma} \rhd_{\mathscr{C}} \theta\mathbf{e} : \theta\boldsymbol{\tau}$.
2. *If* $\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} e : \tau$ *then* $\mathrm{M};\Sigma;\theta\Gamma \rhd_{\mathscr{A}} \theta e : \theta\tau$.

Proof. By structural induction on $\mathbf{e}$ and e. Please see p. 341 for details. ▷

Another important fact to establish is that the types of values accurately reflect any resourced embedded in those values. To state the lemma, I first define a notion of *promotion-worthiness*:

DEFINITION 7.5 (Worthy of promotion)**.**

*An $F_{\mathscr{C}}$ term $\mathbf{e}$ is **worthy with respect to** Σ, written $\Sigma \rhd_{\mathscr{C}} \mathbf{e}$ worthy, if $\mathrm{FL}(\mathbf{e}) \subseteq \mathrm{dom}\,\Sigma$ and $\langle\Sigma|_{\mathrm{FL}(\mathbf{e})}\rangle = \mathsf{U}$. Likewise an $F^{\mathscr{A}}$ term e is **worthy with respect to** Σ if $\mathrm{FL}(e) \subseteq \mathrm{dom}\,\Sigma$ and $\langle\Sigma|_{\mathrm{FL}(e)}\rangle = \mathsf{U}$, and is **worthy with respect to** Γ if*

$\mathrm{FV}(e) \subseteq \mathrm{dom}\,\Gamma$ *and* $\langle \Gamma|_{\mathrm{FV}(e)} \rangle = \mathsf{U}$. *If* e *is worthy with respect to both* Σ *and* Γ, *I write* $\Sigma;\Gamma \rhd_{\mathscr{A}} e$ worthy.

If $\mathbf{e}$ *or* e *is not worthy, I write* $\Sigma \not\rhd_{\mathscr{C}} \mathbf{e}$ worthy *or* $\Sigma;\Gamma \not\rhd_{\mathscr{A}} e$ worthy, *respectively.*

Worthiness captures the notion of expressions that can be "promoted" to allow for unlimited use.[1] In particular, λ abstractions in $F^{\mathscr{A}}$ are given an unlimited (U) type if they are worthy, and an affine (A) type if they are not. Abstractions in $F_{\mathscr{C}}$ are required to be worthy, since they should not close over affine resources.

Note that I impose no such requirement on Λ (type) abstractions, as they have the same qualifier as their body, which regulates their usage accordingly.

LEMMA 7.6 (No hidden locations)**.**

The type of a value reveals whether locations might appear in that value:

1. *If* $\mathrm{M};\Sigma;\bullet \rhd_{\mathscr{C}} \mathbf{v} : \tau$ *then* $\Sigma \rhd_{\mathscr{C}} \mathbf{v}$ worthy.

2. *If* $\mathrm{M};\Sigma;\bullet \rhd_{\mathscr{A}} v : \tau$ *then* $\langle \Sigma|_{\mathrm{FL}(v)} \rangle \sqsubseteq \langle \tau \rangle$. *That is, if* τ *is unlimited then* $\Sigma;\bullet \rhd_{\mathscr{A}} v$ worthy.

Proof. By structural induction on $\mathbf{v}$ and v. Please see p. 347 for details. ▷

The substitution lemma is mostly standard, except that it must split the store type context between the value being substituted and the expression being substituted in. Then I can state and prove the preservation lemma, which shows that reduction preserves configuration typing.

LEMMA 7.7 (Substitution)**.**

1. *If* $\mathrm{M};\Sigma_1;\mathbf{\Gamma},\mathbf{x} : \boldsymbol{\tau}_{\mathbf{x}} \rhd_{\mathscr{C}} \mathbf{e} : \tau$ *and* $\mathrm{M};\Sigma_2;\bullet \rhd_{\mathscr{C}} \mathbf{v} : \boldsymbol{\tau}_{\mathbf{x}}$ *where* $\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2$, *then* $\mathrm{M};\Sigma;\mathbf{\Gamma} \rhd_{\mathscr{C}} \{\mathbf{v}/\mathbf{x}\}\mathbf{e} : \tau$.

2. *If* $\mathrm{M};\Sigma_1;\Gamma, x : \tau_x \rhd_{\mathscr{A}} e : \tau$ *and* $\mathrm{M};\Sigma_2;\bullet \rhd_{\mathscr{A}} v : \tau_x$ *where* $\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2$, *then* $\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} \{v/x\}e : \tau$.

[1] This corresponds to intuitionistic linear logic's (Bierman 1993) promotion rule, which allows adding the exponential modality to a proposition when it depends on no linear resources.

Proof. By induction on the structure of the type derivations for $\mathbf{e}$ and e. Please see p. 350 for details. ▷

LEMMA 7.8 (Preservation)**.**

If $\mathrm{M} \rhd C_1 : \boldsymbol{\tau}$ *and* $C_1 \xrightarrow[\mathrm{M}]{} C_2$ *then* $\mathrm{M} \rhd C_2 : \boldsymbol{\tau}$.

Proof. By cases on the reduction relation. Please see p. 361 for details. ▷

Progress. For the progress lemma, I first identify a class of *faulty expressions* that do not reduce. The definition is large because of the interaction of the two languages and the presence of the store:

DEFINITION 7.9 (Faulty expressions and configurations)**.**

Define these classes of values:

$$\mathbf{Q}^{\Lambda} ::= \lambda \mathbf{x}{:}\boldsymbol{\tau}.\mathbf{e} \mid (\underset{\mathbf{g}f}{\Leftarrow} \tau)^{\ell} v \;\; \textit{where } \tau \textit{ is not } \forall\, \alpha^{\mathfrak{q}}.\tau' \qquad \textit{(faulty for type application)}$$

$$\mathbf{Q}^{\lambda} ::= \Lambda \boldsymbol{\alpha}.\mathbf{v} \mid (\underset{\mathbf{g}f}{\Leftarrow} \tau)^{\ell} v \;\; \textit{where } \tau \textit{ is not } \tau_1 \xrightarrow{\mathfrak{q}} \tau_2 \qquad \textit{(faulty for application)}$$

$$Q^{\Lambda} ::= \lambda x{:}\tau.e \mid \langle v_1, v_2 \rangle \mid \ell \qquad \textit{(faulty for type application)}$$
$$\mid (\tau \underset{f\mathbf{g}}{\Leftarrow})^{\bullet} \mathbf{v} \;\; \textit{where } \tau \textit{ is not } \forall\, \alpha^{\mathfrak{q}}.\tau'$$

$$Q^{\lambda} ::= \Lambda \alpha^{\mathfrak{q}}.v \mid \langle v_1, v_2 \rangle \mid \ell \qquad \textit{(faulty for application)}$$
$$\mid (\tau \underset{f\mathbf{g}}{\Leftarrow})^{\bullet} \mathbf{v} \;\; \textit{where } \tau \textit{ is not } \tau_1 \xrightarrow{\mathfrak{q}} \tau_2$$

$$Q^{\otimes} ::= \Lambda \alpha^{\mathfrak{q}}.v \mid \lambda x{:}\tau.e \mid \ell \qquad \textit{(faulty for unpairing)}$$
$$\mid (\tau \underset{f\mathbf{g}}{\Leftarrow})^{\bullet} \mathbf{v} \;\; \textit{where } \tau \textit{ is not } \tau_1 \otimes \tau_2$$

$$Q_s^{\leftrightarrow} ::= \Lambda \alpha^{\mathfrak{q}}.v \mid \lambda x{:}\tau.e \mid \ell \qquad \textit{(faulty for swapping in } s\textit{)}$$
$$\mid (\tau \underset{f\mathbf{g}}{\Leftarrow})^{\bullet} \mathbf{v} \;\; \textit{where } \tau \textit{ is not } \mathsf{ref}\ \tau_1 \otimes \tau_2 \mid \langle v_1, v_2 \rangle \;\; \textit{where } v_1 \textit{ is not } \ell \in \mathrm{dom}\, s$$

Then we can define the ***faulty expressions with respect to*** *s:*

$$\mathbf{Q}_s ::= \mathbf{Q}^{\Lambda}\boldsymbol{\tau} \mid \mathbf{Q}^{\lambda}\mathbf{v} \mid \mathbf{E}[\mathbf{Q}_s] \mid \mathbf{E}[Q_s] \qquad \textit{(faulty } F_{\mathscr{C}} \textit{ expressions)}$$

$$Q_s ::= Q^{\Lambda}\tau \mid Q^{\lambda} v \mid \mathsf{let}\,\langle x, y\rangle = Q^{\otimes}\ \mathsf{in}\ e \mid \mathsf{swap}\, Q_s^{\leftrightarrow} \qquad \textit{(faulty } F^{\mathscr{A}} \textit{ expressions)}$$
$$\mid E[Q_s] \mid E[\mathbf{Q}_s]$$

A configuration $(s, \mathbf{e})$ *or* (s, e) *is* ***faulty*** *if* $\mathbf{e} \in \mathbf{Q}_s$ *or* $e \in Q_s$*, respectively.*

I next prove two lemmas about faulty expressions: first, that faulty expressions correctly characterize the non-answer normal forms of reduction, and second that faulty expressions do not have types.

LEMMA 7.10 (Uniform evaluation)**.**

For any C *closed with respect to* M*, either* C *is faulty or an answer, or there exists some* C' *closed with respect to* M *such that* $C \longmapsto_{\mathrm{M}} C'$*.*

Proof. Let $(s, \mathbf{e}) = C$; then by structural induction on $\mathbf{e}$. Please see p. 376 for details. ▷

LEMMA 7.11 (Faulty expressions are ill-typed)**.**

1. *For expression* $\mathbf{Q}_s$ *faulty with respect to* s*, there exist no* M*,* Σ_1*,* Σ_2*, and* $\boldsymbol{\tau}$ *such that*
   - $\mathrm{M}; \Sigma_1 \rhd s : \Sigma_1 \boxplus \Sigma_2$ *and*
   - $\mathrm{M}; \Sigma_2; \bullet \rhd_{\mathscr{C}} \mathbf{Q}_s : \boldsymbol{\tau}$.
2. *For expression* Q_s *faulty with respect to* s*, there exist no* M*,* Σ_1*,* Σ_2*, and* τ *such that*
   - $\mathrm{M}; \Sigma_1 \rhd s : \Sigma_1 \boxplus \Sigma_2$ *and*
   - $\mathrm{M}; \Sigma_2; \bullet \rhd_{\mathscr{A}} Q_s : \tau$.

Proof by contradiction. We assume that a faulty expression has a type, and show that it leads to a contradiction, by mutual induction on $\mathbf{Q}_s$ and Q_s. Please see p. 381 for details. ▷

The progress lemma is then a trivial corollary of the previous two lemmas:

COROLLARY 7.12 (Progress)**.**

If $\mathrm{M} \rhd C : \boldsymbol{\tau}$*, then either* C *is an answer or there exists some* C' *such that* $C \longmapsto_{\mathrm{M}} C'$*.*

Proof. Since C types, it is closed; thus, by lemma 7.10, it is an answer, it takes a step, or it is faulty. Because it types, we can eliminate the faulty case by lemma 7.11. □

Type soundness. This brings us, finally, to the main type soundness theorem for $\mathrm{F}^{\mathscr{A}}_{\mathscr{C}}$:

THEOREM 7.13 (Strong soundness)**.**

If $\mathrm{M}; \bullet \vdash_{\mathscr{C}} \mathbf{e} : \tau$ *and* $(\{\}, \mathbf{e}) \xrightarrow[\mathrm{M}]{*} C$*, such that configuration* C *cannot take another step, then* C *is an answer with* $\mathrm{M} \rhd C : \tau$.

Proof. By corollary 7.2 (Programs to configurations), corollary 7.12 (Progress), and lemma 7.8 (Preservation), and induction on the length of the reduction sequence. □

7.4 Implementing Affine Contracts

In this section, I consider two approaches to implementing affine contracts. First, I show how affine contracts might be implemented in system combining an affine language and a conventional language, where the affine language is compiled to the conventional language, in order to allow safe interaction between the two languages. Second, I show how affine contracts are implemented in Alms, which does not currently interact with a conventional language, and discuss why affine contracts are useful even in a single-language system.

7.4.1 Implementing a Two-Language System

Given an affine language such as Alms and a similar, non-affine language, it would be useful to allow modules written in the two languages to interact within a single program. This is the situation that motivated this work and is modeled in the previous section. Here, I consider how to allow safe interaction between the two languages when the affine language is implemented by translation to the non-affine language. For the purposes of informal code

examples, I assume that the non-affine language shares its concrete syntax with Alms.

Representing contracts. Interaction between the two languages will be mediated by behavioral contracts. We represent a contract for a type *'a* as a function taking two parties (which might be module names and source locations) and a value of type *'a* to a value of type *'a*:

```
type 'a contract = party × party → 'a → 'a
```

As an example, a simple (flat) contract might assert some predicate on a first order value:

```
let evenContract (neg: party, pos: party) (x: int) =
  if isEven x then x else blame pos
```

The contract is parametrized by the negative party *neg*, which consumes the value, and the positive party *pos*, which produces the value. If the contracted value satisfies the predicate then it returns the value; otherwise, it signals a contract violation, blaming the positive party.

Following Findler and Felleisen (2002), we may also construct contracts for functional values, given contracts for the domain and codomain:

```
let funContract (dom: 'a contract, cod: 'b contract) : ('a → 'b) contract =
  λ (neg, pos) (f: 'a → 'b) →
    λ (x: 'a) →
      cod (neg, pos) (f (dom (pos, neg) x))
```

When this contract is applied to a function, it performs no checks immediately. Instead, it wraps the function so that, when the resulting function is applied, the domain contract is applied to the actual parameter and the codomain contract to the actual result.

We follow this approach closely for affine function contracts, but with one small change—contracts for affine functions are stateful:

```
let affineContract (dom: 'a contract, cod: 'b contract) : ('a → 'b) contract =
  λ (neg, pos) (f: 'a → 'b) →
    let stillGood = ref true in
      λ (x: 'a) →
        if deref stillGood
          then stillGood ← false; cod (neg, pos) (f (dom (pos, neg) x))
          else blame neg
```

It is important to note when the state is allocated: when the contract is applied to a functional value. This means that each contracted function can be applied exactly once, and the state is checked when the function is applied. This approach works for functions because we can wrap a function to modify its behavior. But what about for other affine values such as typestate capabilities? We must consider how non-functional values move between the two sublanguages.

According to the type translation metafunction $(\cdot)^{\mathscr{A}}$ defined in the model (figure 7.9 on p. 167), an *opaque* type ρ translates to a wrapped type $\{\rho\}$ in the other language, but an actual implementation needs to define how such a type is represented in the other language. In the model, values of opaque type are inert: They have no available operations other than passing them back to their native sublanguage.[2]

What this means for an implementation is that a value of opaque type can be represented by an abstract type that pairs the foreign value with a location. For example, a signature and structure implementing wrappers for opaque types appears in figure 7.17. Function *wrap*, given the negative party and a value, creates a wrapped, opaque value. Function *unwrap* unwraps a value or blames the negative party.

Translation. In the two-language scheme, contracts are added during translation from the affine source language to the non-affine object language. After

[2]Opaque types may seem limiting, but Matthews and Findler (2007) show that it is possible, in what they call the "lump embedding," for each sublanguage to marshal its opaque values for the other sublanguage as desired. In practice, this amounts to exporting a fold to the other sublanguage.

```
module type OPAQUE = sig
  type 'a opaque
  val wrap : party × party → 'a → 'a opaque
  val unwrap : 'a opaque → 'a
end

module Opaque : OPAQUE = struct
  type 'a opaque = 'a × party × bool ref
  let wrap (neg, pos) v = (v, neg, ref true)
  let unwrap (v, neg, stillGood) =
    if deref stillGood
      then stillGood ← false; v
      else blame neg
end
```

Figure 7.17: Wrappers for opaque types

$$
\begin{aligned}
\mathscr{CA}[\![\tau_1 \xrightarrow{\mathsf{U}} \tau_2]\!] &= \mathit{funContract}\ (\mathscr{AC}[\![\tau_1]\!], \mathscr{CA}[\![\tau_2]\!]) \\
\mathscr{CA}[\![\tau_1 \xrightarrow{\mathsf{A}} \tau_2]\!] &= \mathit{affineContract}\ (\mathscr{AC}[\![\tau_1]\!], \mathscr{CA}[\![\tau_2]\!]) \\
\mathscr{CA}[\![\rho]\!] &= \mathit{Opaque.wrap} && \text{if } \langle\rho\rangle = \mathsf{A} \\
\mathscr{CA}[\![\mathrm{int}]\!] &= \boldsymbol{\lambda}\ \_\ z \to z \\
\mathscr{AC}[\![\tau_1 \xrightarrow{\mathsf{q}} \tau_2]\!] &= \mathit{funContract}\ (\mathscr{CA}[\![\tau_1]\!], \mathscr{AC}[\![\tau_2]\!]) \\
\mathscr{AC}[\![\rho]\!] &= \boldsymbol{\lambda}\ \_ \to \mathit{Opaque.unwrap} && \text{if } \langle\rho\rangle = \mathsf{A} \\
\mathscr{AC}[\![\mathrm{int}]\!] &= \boldsymbol{\lambda}\ \_\ z \to z
\end{aligned}
$$

Figure 7.18: Type directed generation of coercions

type checking, such an implementation translates affine language modules to non-affine language modules and wraps all interlanguage variable references with contracts that enforce the affine language's view of the variable.

Figure 7.18 shows several cases from a pair of metafunctions $\mathscr{CA}[\![\cdot]\!]$ and $\mathscr{AC}[\![\cdot]\!]$, which perform this wrapping. Each metafunction generates a coercion based on the affine language type of the coerced value.

Metafunction $\mathscr{CA}[\![\cdot]\!]$ constructs the coercion for references to affine language values from the conventional language. Whereas unlimited functions

are wrapped using a regular function contract, one-shot functions are wrapped using an affine function contract. In each case, the metafunctions are used to recursively generate coercions for function domains and codomains. Note that the direction of the coercion generation is reversed for domain contravariance. Values of affine, opaque type are wrapped using *Opaque.wrap*. As the final case for $\mathscr{CA}[\![\cdot]\!]$, integers pass between the two languages unwrapped, since their representations are the same and integers are never affine. The translation needs to generate a no-op coercion that takes an extra argument in order to discard the parties expected by a contract value. (Note that this translation does not handle functions with non-constant qualifier expressions built out of type variables, but I discuss one way to handle that situation in the next section.)

Metafunction $\mathscr{AC}[\![\cdot]\!]$ constructs the coercion for references to conventional language values from the affine language. All functions, regardless of qualifier, are coerced using a regular (not affine) function contract, because the affine language's type system already guarantees not to allow their reuse. As before, the domain and codomain are wrapped recursively. Values of affine, opaque type are unwrapped using *Opaque.unwrap*, which is placed in a function in order to ignore the parties. (Wrapped opaque values remember the negative party from when they were wrapped.) Finally, as before, some values, such as integers, pass between the languages unchecked.

7.4.2 Dynamic Promotion in Alms

While this chapter is concerned with interaction between an Alms-like affine language and a conventional, non-affine language, Alms's implementation currently does not support such interaction. However, affine contracts are still useful in Alms because they provide a means of dynamically checking substructural constraints. In particular, affine contracts allow using a one-shot function in a context whose type demands an unlimited function, while checking dynamically that the context actually uses the function only once. I call this feature *dynamic promotion*.

For example, Alms relies on a primitive thread fork operation provided by the run-time system to start new threads:

$primThreadFork$: (unit → unit) → thread

While there is good reason to believe that *primThreadFork* applies its argument at most once, this is not guaranteed by the run-time system, which is implemented in a conventional language. A contract can be used to promote *primThreadFork*'s type to (unit $\xrightarrow{\mathsf{A}}$ unit) → thread while checking, dynamically, that it actually satisfies the necessary invariant. (Note that this coercion amounts to a downcast according to Alms's subtype relation)

To perform such coercions, Alms includes a special expression form:

⟨*expr*⟩ ::= ⟨*expr*⟩ ":>" ⟨*type*⟩

This generates a coercion, if possible, to give the expression the specified type. For example, we can define *threadFork* using a coercion as follows:

let *threadFork* = *primThreadFork* :> (unit $\xrightarrow{\mathsf{A}}$ unit) → thread

This wraps *primThreadFork* as necessary to give it the requested type.

How it works. The implementation of dynamic promotion is split between the standard library, which defines the basic contract combinators for affine contracts, and the type checker, which performs type-directed generation of coercions, constructing them out of the values provided by the standard library.

The standard library defines contracts as in the two-language version above, where a contract for type *'a* is a function that takes a pair of parties and a value of type *'a* to a value of type *'a*. The definition of *funContract* is the same as above, as well. However, the definition of *affineContract* will not work. Because the value wrapped by *affineContract* above closes over the affine value being wrapped, Alms would give the wrapped value an affine type as well. The key trick is to store the affine value itself in an unlimited reference to

an option type, so that the result of wrapping mentions no affine variables directly:

```
let affineContract (dom: 'a contract, cod: 'b contract)
      : party × party → ('a -A-> 'b) → ('a -U-> 'b)
  λ (neg, pos) (f: 'a -A-> 'b) →
    let stillGood = ref (Some f) in
      λ (x: 'a) →
        match stillGood ↔ None with
        | Some g → cod (neg, pos) (g (dom (pos, neg) x))
        | None → blame neg
```

The operation $r \leftrightarrow v$ corresponds to the reference swap operation in the model of §7.2: It stores value v in reference r and returns the old value read from r, atomically.

When checking a coercion expression e :> t, the type checker generates a coercion that refers to *funContract* and *affineContract* in the standard library. Unlike the interlanguage coercions of figure 7.18, however, Alms's type checker has to contend with incomplete types that contains unification variables, including qualifier expressions that may contain both quantified type variables and unification variables. Alms interleaves coercion generation and the rest of type inference, because each provides valuable information to the other. Coercion generation needs to know the types between which a coercion should be generated, and generating a coercion between incomplete types can generate more inference constraints.

A simplified version of the algorithm used by Alms for coercion generation appears in figure 7.19. (For the remainder of this section, I use `typewriter font` for Alms, in order to make the distinction between object and meta-language clear.) Metafunction *build* takes two types, the type inferred for an expression and the type to coerce it to, and generates an expression to perform the coercion. To interleave coercion generation and type inference, *build* is defined in a constraint solving monad, which is why it returns type ConstraintM Expr. The constraint solving monad provides an operation *addConstraint* : Constraint → ConstraintM (), which adds a constraint to the current constraint set, simplifying as appropriate.

This simplified version of the algorithm has three cases:

```
build : Type → Type → ConstraintM Expr

build ⟦τ1 -ξ1> τ1'⟧ ⟦τ2 -U> τ2'⟧ when ξ1 ≠ U =
  do dom ← build τ2 τ1
     cod ← build τ1' τ2'
     return ⟦ affineContract (dom, cod) ⟧

build ⟦τ1 -ξ1> τ1'⟧ ⟦τ2 -ξ2> τ2'⟧ =
  do addConstraint (ξ1 ⊑ ξ2)
     dom ← build τ2 τ1
     cod ← build τ1' τ2'
     return ⟦ funContract (dom, cod) ⟧

build τ1 τ2 =
  do addConstraint (τ1 ≤ τ2)
     return ⟦ anyContract ⟧
```

Figure 7.19: Coercion generation in Alms (simplified)

- To coerce from a function type τ_1 `-`ξ_1`>` τ_1' to a function type τ_2 `-U>` τ_2', where ξ_1 is not `U`, a run-time check allows us to dispense with the static constraint on ξ_1. Thus, we recursively generate coercions for the domain and codomain of the function to be coerced, and then use `affineContract` from the standard library to generate the immediate coercion.

- To coerce from a function type τ_1 `-`ξ_1`>` τ_1' to a function type τ_2 `-`ξ_2`>` τ_2', when not covered by the previous case, we perform a static check instead by asserting the constraint that $\xi_1 \sqsubseteq \xi_2$. We recursively generate domain and codomain coercions, and then build the contract using `funContract`, which does not enforce affinity its contracted value.

- In other cases, *build* enforces the coercion statically by asserting a subtyping constraint. Because no dynamic check is then necessary, it returns `anyContract`, which is the contract that always succeeds.

The choice of which qualifiers to enforce dynamically and which statically is a pragmatic one; the rule is designed to be simple and predictable. Note that

a programmer can always request a dynamic check by writing `U` in the right place, and can always request a static check by preceding the coercion with a non-coercing type annotation. This makes it possible to get the any desired combination of static and dynamic checks.

Figure 7.20 extends the algorithm of figure 7.19 to handle recursive and universal types. Function *build'* now takes an additional argument, which is a partial map from pairs of Alms types to Alms identifiers. We use this to handle recursive types by constructing recursive coercions. The first case of *build'* checks whether we have seen the given types before, and if so, uses the identifier from the partial map as the coercion between them. The next two cases unfold equirecursive types and extend map μ with a fresh Alms variable name, in order to tie a knot in case it finds a cycle in the types. (The actual implementation avoids generating `let recs` unnecessarily.) Function types are handled as before. Quantified types ($Q\ \alpha\,.\,\tau$) are handled by recurring on the bodies, which works because Alms types are represented in a standardized form that causes quantifiers to align.

The example, revisited. Given the implementation above, Alms can generate the coercion necessary to safely pass one-shot functions to the primitive `threadFork` operation. Specifically, given the program text

```
let threadFork = primThreadFork :> (unit -A> unit) -> thread,
```

Alms generates

```
let threadFork =
  funContract (affineContract anyContract anyContract)
              anyContract
              ("context at <stdin>:8:18-31",
               "value at <stdin>:8:18-31")
              primThreadFork.
```

```
build' : (Type × Type ⇀ Var) → Type → Type → ConstraintM Expr

build' μ τ1 τ2 when (τ1, τ2) ∈ dom μ =
  return ⟦ μ(τ1, τ2) ⟧

build' μ ⟦mu α . τ1'⟧ τ2 =
    x    ← gensym
    let τ1 = {⟦mu α . τ1'⟧/α}τ1'
        μ' = μ[(⟦mu α . τ1'⟧, τ2) ↦ x]
    c    ← build' μ' τ1 τ2
    return ⟦ let rec x = c in x ⟧

build' μ τ1 ⟦mu α . τ2'⟧ =
    x    ← gensym
    let τ2 = {⟦mu α . τ2'⟧/α}τ2'
        μ' = μ[(τ1, ⟦mu α . τ2'⟧) ↦ x]
    c    ← build' μ' τ1 τ2
    return ⟦ let rec x = c in x ⟧

build' μ ⟦τ1 -ξ1> τ1'⟧ ⟦τ2 -U> τ2'⟧ when ξ1 ≠ U =
  do dom ← build' μ τ2 τ1
     cod ← build' μ τ1' τ2'
     return ⟦ affineContract cod dom ⟧

build' μ ⟦τ1 -ξ1> τ1'⟧ ⟦τ2 -ξ2> τ2'⟧ =
  do addConstraint (ξ1 ⊑ ξ2)
     dom ← build' μ τ2 τ1
     cod ← build' μ τ1' τ2'
     return ⟦ funContract cod dom ⟧

build' μ ⟦Q α . τ1⟧ ⟦Q α . τ2⟧ =
  build' μ τ1 τ2

build' μ τ1 τ2 =
  do addConstraint (τ1 ≤ τ2)
     return ⟦ anyContract ⟧
```

Figure 7.20: Coercion generation in Alms

CHAPTER 8

Substructural Types and Control

ONE SHORTCOMING OF typestate in Alms is that it cannot guarantee that a protocol will run to completion. Because tracked values or capabilities in Alms are affine, they prevent duplication, which prevents faulty usage of some resource, but they cannot prevent the owner of some resource from merely discarding it.

As an example, consider affine file handles with the following interface:

```
type file : A
val fopen  : string → file
val fread  : file → string × file
val fwrite : file → string -A-> file
val fclose : file → unit
```

Because file handles are affine, they must be single-threaded through a program, which ensures that no program will attempt to read or write a file after it has been closed. However, the interface enforces no guarantee that a program will explicitly close a file, which means that the run-time system is still responsible to close file handles that are implicitly dropped.

One solution to this problem might be to add linear types to Alms. Then we could make file handles linear, which should guarantee that all file handles get closed explicitly:

```
type file : L
```

Unfortunately, the situation is not so simple. Consider the following function, which opens a configuration file and uses it to find out the name of a log file, which it then opens and returns along with the configuration file:

```
let initializeFiles () =
  let configFile             = fopen configFileName in
  let (config, configFile) = parseConfigFile configFile in
  let logFile                = fopen config.logFileName in
    (logFile, configFile)
```

We made file handles linear to ensure their explicit release, which relieves the run-time system of the need to close files automatically, but that means that *initializeFiles* as written has a resource leak. If the log file cannot be opened, then *fopen* throws a file-not-found exception, which happens after opening the configuration file. Because the function raises an exception but does not return the successfully opened configuration file, there is no way for recovery code that catches the exception to close the configuration file.

In short, exceptions and linear types refuse to get along, because linear types make promises that exceptions do not let them keep.

With affine rather than linear file handles, as in Alms, *initializeFiles* is not a problem, because such a type system cannot require that resources be released explicitly. In a language with affine types, implicitly dropping a value is just fine—presumably there is a garbage collector to finalize resources—and only duplication is forbidden. However, while exceptions do not cause a problem for an affine language, other forms of nonlocal control do: Consider adding the delimited continuation operators *shift* and *reset* (Danvy and Filinski 1989) to Alms.

As an example, consider an event-based parsing library with a callback-based interface that takes a file handle to parse and a function to call for each parsing event:

```
val forEvents : file → (event → unit) → unit
```

Suppose we would like, instead, a stream-like interface to parsing, where we can request each event as desired. We can obtain this interface in terms of

forEvents using **shift** and **reset**:

```
let parseStream (f: file)
  reset (forEvents f (λ ev → shift k in ‘Next (ev, k)); ‘Done)
```

Function *parseStream* takes a file handle to parse, and returns a pair of the first event and a function that will return the next event and a new function, and so on, until *forEvents* finishes, at which point the last function returns ‘*Done*. This works as follows: *forEvents* begins parsing, and at some point applies the callback to a parsing event. The callback uses **shift** to capture its continuation up to the delimiter **reset**, which includes the rest of the parsing process; rather than invoke the continuation, the callback returns the event and the continuation to the context of the **reset**, which can use the continuation later to resume parsing.

As in the previous example, this interaction of substructural types and control fails to preserve the resource invariants. Presumably, *forEvents* keeps the file open while parsing from it and closes it when done. (It may maintain other substructural state as well, but for this example the file handle suffices.) So long as we call each function returned by *parseStream* once to get the next event and function, that should work correctly. But if we hold onto one of the functions and call it after *forEvents* has finished and closed the file, that will cause an attempt to access the closed file handle.

What went wrong, and how to fix it. Typically, an affine type system works by imposing two syntactic requirements: a variable of affine type, such as *f*, cannot appear twice in its scope (up to branching), and a function that closes over an affine variable must itself have an affine type. The *parseStream* example apparently violates neither dictum. In the presence of delimited continuations, we need a third rule: that *a captured continuation that contains an affine value must not duplicated.* A simple approximation of this rule is to give *all* captured continuations an affine (or in a linear system, linear) type. Such a rule permits some uses of delimited continuations, such as coroutines, but not others.

Similarly, *initializeFiles* does not appear to violate linearity, because all file handles are used, syntactically, exactly once. In the presence of exceptions,

linearity requires an additional rule: that *a continuation that contains a linear value may not be discarded.* A simple approximation to this rule is to give all continuations a linear type, which amounts to prohibiting exceptions.

Treating all continuations as affine or linear is overly restrictive because the resource leak and closed file handle access in the above examples can be fixed by small changes to the code. For *initializeFiles*, it is necessary to close the configuration file when an exception is raised while opening the log file:

```
let initializeFiles () =
  let configFile            = fopen configFileName in
  let (config, configFile) = parseConfigFile configFile in
    match tryfun (λ () → fopen config.logFileName) with
    | Left e          → fclose configFile; raise e
    | Right logFile → (logFile, configFile)
```

The definition of *parseStream* given above is acceptable, but it is necessary to ensure that it gets the right type:

$$\mathsf{file} \to \mu\text{`}a.\ [\ \text{`}\mathit{Next}\ \mathsf{of}\ \mathsf{event} \times (\mathsf{unit} \xrightarrow{\mathsf{A}} \text{`}a) \mid \text{`}\mathit{Done}\ ]$$

The presence of qualifier A means that each function to get the next event is invoked at most once, so there is no way to cause *forEvents* to access a closed file.

My solution is a type-and-effect system (Lucassen and Gifford 1988) that permits the repaired version of *initializeFiles* while rejecting the original, erroneous version, and that gives the correct, affine type to *parseStream*, without requiring that all continuations captured by **shift** be affine. The key idea is to assign to each expression a control effect that reflects whether it may duplicate or drop its continuation, and to prohibit using an expression in a context that cannot be treated as the control effect allows.

In this chapter, I

- exhibit a *generic* type system for substructural types and control defined in terms of an unspecified, abstract control effect (§8.3);

- give soundness criteria for the abstract control effect and prove type soundness for the generic system, relying on the soundness of the abstract control effect (§8.4); and
- demonstrate three concrete instantiations of control effects—exceptions and two versions of delimited continuations—and prove that they meet the soundness criteria (§8.5).

The generic type-and-effect system in §8.3 is defined as an extension to λ^{URAL} (Ahmed et al. 2005), a substructural λ calculus. I use λ^{URAL} rather ${}^{a}\lambda_{ms}$ (chapter 5) because I wish to demonstrate that my solution works not only for affine types, but for linear and relevant types as well; however, applying the technique described in this chapter to other languages with substructural types, such as ${}^{a}\lambda_{ms}$, should be straightforward. I review λ^{URAL} in §8.2 after discussing related work in §8.1.

8.1 Related Work

This work is not the first to relate substructural types to control operators and control effects. Thielecke (2003) shows how to use a type-and-effect system to reason about how expressions treat their continuations. In particular, he gives a continuation-passing style transform where continuations that will be used linearly are given a linear type. Thielecke notes that many useful applications of continuations treat them linearly. However, his goals are different than mine. He uses substructural types in his object language to reason about how continuations will be used in a non-substructural source language, whereas I want to reason about continuations in order to safely use substructural types. Thielecke has linear types only in the object language of his translation, whereas I am interested in linear (and other substructural) types in the source language.

Other recent work relates substructural logics and control. Kiselyov and Shan (2007) use a substructural logic to allow the "dynamic" control operator *shift0* to modify answer types in a typed setting. Unlike this work, their *terms* are structures in substructural logic, not their types. Mazurak and Zdancewic's

(2010) Lolliproc relates double negation elimination in classical linear logic to delimited control.

This chapter draws significantly on other work on control operators, effect systems, and substructural types as well.

Control operators. The literature contains a large vocabulary of control operators, extending back to ISWIM's J operator (Landin 1965), Reynolds's (1972) *escape*, and Scheme's *catch* (Steele Jr. and Sussman 1975) and *call-with-current-continuation* (Clinger 1985). However, for integration in a language with substructural types, control operators with delimited extent, originating with Felleisen's (1988) $\mathcal{F}$, are most appropriate, because without some way to mask out control effects, any use of control pollutes the entire program and severely limits the utility of substructural types.

As examples of control features to add to a substructural calculus, I consider the delimited continuation operators *shift* and *reset* (Danvy and Filinski 1989) and structured exception handling (Goodenough 1975). Both *shift*/*reset* and structured exceptions have been combined with type-and-effect systems to make them more amenable to static reasoning.

Type-and-effect systems for control. Java (Gosling et al. 1996) provides checked exceptions, an effect system for tracking the exceptions that a method may raise. My version of exception effects is similar to Java's, except that mine offers effect polymorphism, which makes higher-order programming with checked exceptions more convenient. My type system for exceptions appears in §8.5.3.

Because Danvy and Filinski's (1989) *shift* captures a delimited continuation up to the nearest *reset* delimiter, typing *shift* and *reset* requires some nonlocal means of communicating types between delimiters and control operators. They realize this communication with a type-and-effect system, which allows *shift* to capture and compose continuations of varying types. Asai and Kameyama (2007) extend Danvy and Filinski's (monomorphic) type system with polymorphism, which includes polymorphism of answer types. I give two substructural type systems with *shift* and *reset*. Section 8.5.1 presents a simpler version that severely limits the answer types of continuations that may be captured.

v	::=		values
	|	x	variable
	|	$\lambda x.e$	abstraction
	|	$\Lambda.e$	type abstraction
	|	$\mathsf{inl}\, v$	sum construction, left
	|	$\mathsf{inr}\, v$	sum construction, right
	|	$[v_1, v_2]$	sum elimination
	|	$\langle v_1, v_2 \rangle$	pair construction
	|	$\mathsf{uncurry}\, v$	pair elimination
	|	$\langle\rangle$	the nil value
	|	$\mathsf{ignore}\, v$	nil elimination
	|	ℓ	location (run time only)
e	::=		expressions
	|	v	values
	|	$e_1\, e_2$	application
	|	$e\, \_$	type application
	|	$\mathsf{new}^{\mathsf{q}}\, e$	reference allocation
	|	$\mathsf{delete}\, e$	reference deallocation
	|	$\mathsf{read}\, e$	reference read
	|	$\mathsf{swap}\, e_1\, e_2$	reference read and write

Figure 8.1: λ^{URAL} syntax (i): expression level

Then, in §8.5.2, I combine the simpler system with a polymorphic version of Danvy and Filinski's, similar to Asai and Kameyama's, to allow answer-type modification and polymorphism in a substructural setting.

8.2 Syntax and Semantics of λ^{URAL}

Because I wish to treat not only affine types, but linear and relevant types as well, I use a model in this chapter that is more general in that respect than models in previous chapters: I add control effects to Ahmed et al.'s (2005) λ^{URAL}, a substructural λ calculus. My presentation of λ^{URAL} is heavily based on theirs, with a few small changes.

$\mathfrak{q}$	::=		constant qualifiers
	|	U	unlimited
	|	R	relevant
	|	A	affine
	|	L	linear
ξ	::=		qualifiers
	|	α	qualifier variable
	|	$\mathfrak{q}$	qualifier constant
$\overline{\tau}$	::=		pretypes
	|	α	pretype variable
	|	1	multiplicative unit
	|	$\tau_1 \otimes \tau_2$	multiplicative conjunction
	|	$\tau_1 \oplus \tau_2$	additive disjunction
	|	$\tau_1 \multimap \tau_2$	function
	|	$\mathsf{ref}\ \tau$	reference
	|	$\forall \alpha{:}\kappa.\, \tau$	universal quantification
τ	::=		types
	|	α	type variable
	|	${}^{\xi}\overline{\tau}$	qualified pretype
ι	::=		type-level terms
	|	ξ	qualifier
	|	$\overline{\tau}$	pretype
	|	τ	type
κ	::=		kinds
	|	QUAL	qualifiers
	|	$\overline{\star}$	pretypes
	|	$\star$	types

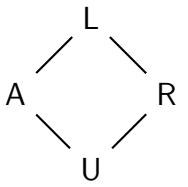
Figure 8.2: λ^{URAL} syntax (ii): type and kind level

The expression level. The syntax of values and expressions in λ^{URAL} appears in figure 8.1. Values include abstractions, type abstractions, and introduction and elimination forms for sums, products, and the nil value. At run time, values also include location names. (This differs from Ahmed et al.'s presentation of λ^{URAL} by including sums—additive disjunctions, to be precise.) Expressions include values, application, type application, and operations on mutable references. Following Ahmed et al., I elide the formal parameter in type abstractions and the actual parameter in type applications.

The type level. Expressions in λ^{URAL} are classified by types (τ), the syntax of which appears in figure 8.2. The four constant qualifiers (q) include U and A, as in Alms, and two additional qualifiers:

L as in *linear*, for values that may be neither duplicated nor implicitly dropped; and

R as in *relevant*, for values that may be duplicated (contraction) but not dropped.



The four constant qualifiers form a lattice that extends Alms's qualifier lattice. As in Alms, it is safe to treat a value as if it has a higher qualifier than its own.

Qualifiers (ξ) include both qualifier constants and type variables, allowing for qualifier polymorphism. Pretypes ($\overline{\tau}$) specify the representation of a value, and its introduction and elimination rules. Pretypes include type variables, the unit type, multiplicative conjunction, additive disjunction, function types, references, and universal quantification. Types (τ) classify expressions. A type is either a pretype decorated with its qualifier (${}^{\xi}\overline{\tau}$) or a type variable. Non-terminal ι stands for all three kinds of type-level terms as a group.

The kind level. Types in λ^{URAL} are classified by three kinds (κ): QUAL for qualifiers, $\overline{\star}$ for pretypes, and $\star$ for types. Type variables may have any of these three kinds, which is why universal quantification ($\forall\alpha{:}\kappa.\,\tau$) specifies the kind of α.

$$s ::= \{\} \mid \{\ell \overset{\mathfrak{q}}{\mapsto} v\} \mid s_1 \uplus s_2 \qquad \text{stores}$$

$$E ::= [] \mid E\,e \mid v\,E \mid E\,\_ \mid \mathsf{new}^{\mathfrak{q}}\,E \mid \mathsf{delete}\,E \mid \mathsf{read}\,E \mid \mathsf{swap}\,E\,e \mid \mathsf{swap}\,v\,E \qquad \text{evaluation contexts}$$

$\boxed{(s,e) \longmapsto (s',e')}$ *(reduction)*

$$(\beta_v) \quad (s,(\lambda x.\,e)\,v) \longmapsto (s,\{v/x\}e)$$

$$(\mathrm{B}_\tau) \quad (s,(\Lambda.\,e)\,\_) \longmapsto (s,e)$$

$$(\textsc{Ignore}) \quad (s,\mathsf{ignore}\,\langle\rangle\,v) \longmapsto (s,v)$$

$$(\textsc{LetPair}) \quad (s,\mathsf{uncurry}\,v\,\langle v_1,v_2\rangle) \longmapsto (s,v\,v_1\,v_2)$$

$$(\textsc{CaseL}) \quad (s,[v_1,v_2](\mathsf{inl}\,v)) \longmapsto (s,v_1\,v)$$

$$(\textsc{CaseR}) \quad (s,[v_1,v_2](\mathsf{inr}\,v)) \longmapsto (s,v_2\,v)$$

$$(\textsc{New}) \quad (s,\mathsf{new}^{\mathfrak{q}}\,v) \longmapsto (s \uplus \{\ell \overset{\mathfrak{q}}{\mapsto} v\},\ell)$$

$$(\textsc{Delete}) \quad (s \uplus \{\ell \overset{\mathfrak{q}}{\mapsto} v\},\mathsf{delete}\,\ell) \longmapsto (s,v)$$

$$(\textsc{Read}) \quad (s \uplus \{\ell \overset{\mathfrak{q}}{\mapsto} v\},\mathsf{read}\,\ell) \longmapsto (s \uplus \{\ell \overset{\mathfrak{q}}{\mapsto} v\},v)$$

$$(\textsc{Swap}) \quad (s \uplus \{\ell \overset{\mathfrak{q}}{\mapsto} v_1\},\mathsf{swap}\,\ell\,v_2) \longmapsto (s \uplus \{\ell \overset{\mathfrak{q}}{\mapsto} v_2\},\langle\ell,v_1\rangle)$$

$$(\textsc{Cxt}) \quad \frac{(s,e) \longmapsto (s',e')}{(s,E[e]) \longmapsto (s',E[e'])}$$

Figure 8.3: λ^{URAL} operational semantics

8.2.1 Operational Semantics

The operational semantics of λ^{URAL} is mostly standard and appears in figure 8.3. Note that each store location is annotated with a qualifier constant ($\mathfrak{q}$), which is selected when allocating a new reference and indicates which structural rules apply to the corresponding reference. Reduction is call-by-value and evaluates operators before operands, which is important when considering the sequencing of effects in §8.3.

$\boxed{\Delta \vdash \iota : \kappa}$ *(kinding type-level terms)*

$$\textsc{K-Var}\ \frac{\alpha{:}\kappa \in \Delta}{\Delta \vdash \alpha : \kappa} \qquad \textsc{K-Qual}\ \frac{}{\Delta \vdash \mathfrak{q} : \mathsf{QUAL}} \qquad \textsc{K-Arr}\ \frac{\Delta \vdash \tau_1 : \star \qquad \Delta \vdash \tau_2 : \star}{\Delta \vdash \tau_1 \multimap \tau_2 : \overline{\star}} \qquad \textsc{K-All}\ \frac{\Delta, \alpha{:}\kappa \vdash \tau : \star}{\Delta \vdash \forall \alpha{:}\kappa.\tau : \overline{\star}}$$

$$\textsc{K-Unit}\ \frac{}{\Delta \vdash 1 : \overline{\star}} \qquad \textsc{K-Sum}\ \frac{\Delta \vdash \tau_1 : \star \qquad \Delta \vdash \tau_2 : \star}{\Delta \vdash \tau_1 \oplus \tau_2 : \overline{\star}} \qquad \textsc{K-Prod}\ \frac{\Delta \vdash \tau_1 : \star \qquad \Delta \vdash \tau_2 : \star}{\Delta \vdash \tau_1 \otimes \tau_2 : \overline{\star}}$$

$$\textsc{K-Ref}\ \frac{\Delta \vdash \tau : \star}{\Delta \vdash \mathsf{ref}\ \tau : \overline{\star}} \qquad \textsc{K-Type}\ \frac{\Delta \vdash \overline{\tau} : \overline{\star} \qquad \Delta \vdash \xi : \mathsf{QUAL}}{\Delta \vdash {}^{\xi}\overline{\tau} : \star}$$

Figure 8.4: λ^{URAL} statics (i): kinding

$\boxed{\Delta \vdash \xi_1 \preceq \xi_2}$ *(qualifier subsumption)*

$$\textsc{QSub-Bot}\ \frac{\Delta \vdash \xi : \mathsf{QUAL}}{\Delta \vdash \mathsf{U} \preceq \xi} \qquad \textsc{QSub-Top}\ \frac{\Delta \vdash \xi : \mathsf{QUAL}}{\Delta \vdash \xi \preceq \mathsf{L}} \qquad \textsc{QSub-Refl}\ \frac{\Delta \vdash \xi : \mathsf{QUAL}}{\Delta \vdash \xi \preceq \xi}$$

$\boxed{\Delta \vdash \tau \preceq \xi}$ *(qualifier bound for types)*

$$\textsc{B-Var}\ \frac{\Delta \vdash \alpha : \star}{\Delta \vdash \alpha \preceq \mathsf{L}} \qquad \textsc{B-Type}\ \frac{\Delta \vdash \overline{\tau} : \overline{\star} \qquad \Delta \vdash \xi' \preceq \xi}{\Delta \vdash {}^{\xi'}\overline{\tau} \preceq \xi}$$

$\boxed{\Delta \vdash \Gamma \preceq \xi}$ *(qualifier bound for type contexts)*

$$\textsc{B-Nil}\ \frac{\Delta \vdash \xi : \mathsf{QUAL}}{\Delta \vdash \bullet \preceq \xi} \qquad \textsc{B-Cons}\ \frac{\Delta \vdash \Gamma \preceq \xi \qquad \Delta \vdash \tau \preceq \xi}{\Delta \vdash \Gamma, x{:}\tau \preceq \xi}$$

Figure 8.5: λ^{URAL} statics (ii): qualifiers

$\boxed{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2}$ *(type context splitting)*

$$\frac{}{\Delta \vdash \bullet \rightsquigarrow \bullet \boxplus \bullet}\ \textsc{S-Nil} \qquad \frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Delta \vdash \tau : \star}{\Delta \vdash \Gamma, x{:}\tau \rightsquigarrow (\Gamma_1, x{:}\tau) \boxplus \Gamma_2}\ \textsc{S-ConsL}$$

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Delta \vdash \tau : \star}{\Delta \vdash \Gamma, x{:}\tau \rightsquigarrow \Gamma_1 \boxplus (\Gamma_2, x{:}\tau)}\ \textsc{S-ConsR} \qquad \frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Delta \vdash \tau \preceq \mathsf{R}}{\Delta \vdash \Gamma, x{:}\tau \rightsquigarrow (\Gamma_1, x{:}\tau) \boxplus (\Gamma_2, x{:}\tau)}\ \textsc{S-Contract}$$

Figure 8.6: λ^{URAL} statics (iii): context splitting

8.2.2 Static Semantics

Type judgments for λ^{URAL} use two kinds of contexts:

Δ	::=		kind contexts
	|	$\bullet$	empty
	|	$\Delta, \alpha{:}\kappa$	kind of type variable
Γ	::=		type contexts
	|	$\bullet$	empty
	|	$\Gamma, x{:}\tau$	type of variable

Figure 8.4 contains the kinding judgment ($\Delta \vdash \iota : \kappa$), which assigns kinds to type-level terms. This judgment enforces the type/pretype structure, whereby type constructors such as $\oplus$ form a pretype from types (rule K-SUM), and decorating a pretype with a qualifier forms a type (rule K-TYPE).

In figure 8.5, three judgments relate qualifiers to each other, to types, and to type contexts. Qualifier subsumption ($\Delta \vdash \xi_1 \preceq \xi_2$) defines the qualifier order, with top L and bottom U. The next judgment bounds a type by a qualifier; judgment $\Delta \vdash \tau \preceq \xi$ means that values of type τ may safely be used according to the structural rules implied by ξ. Finally, bounding a type context by a qualifier ($\Delta \vdash \Gamma \preceq \xi$) means that every type in context Γ is bounded by qualifier ξ.

$\boxed{\Delta;\Gamma \vdash e : \tau}$ *(typing expressions)*

T-Weak
$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Delta \vdash \Gamma_2 \preceq \mathsf{A} \qquad \Delta;\Gamma_1 \vdash e : \tau}{\Delta;\Gamma \vdash e : \tau}$$

T-Var
$$\frac{\Delta \vdash \tau : \star}{\Delta;\bullet, x{:}\tau \vdash x : \tau}$$

T-Abs
$$\frac{\Delta \vdash \Gamma \preceq \xi \qquad \Delta;\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Delta;\Gamma \vdash \lambda x.e : {}^{\xi}(\tau_1 \multimap \tau_2)}$$

T-TAbs
$$\frac{\Delta \vdash \Gamma \preceq \xi \qquad \Delta, \alpha{:}\kappa;\Gamma \vdash e : \tau}{\Delta;\Gamma \vdash \Lambda.e : {}^{\xi}\forall \alpha{:}\kappa.\tau}$$

T-Inl
$$\frac{\Delta \vdash \tau_1 \preceq \xi \qquad \Delta \vdash \tau_2 : \star \qquad \Delta;\Gamma \vdash v_1 : \tau_1}{\Delta;\Gamma \vdash \mathsf{inl}\, v_1 : {}^{\xi}(\tau_1 \oplus \tau_2)}$$

T-Inr
$$\frac{\Delta \vdash \tau_2 \preceq \xi \qquad \Delta \vdash \tau_1 : \star \qquad \Delta;\Gamma \vdash v_2 : \tau_2}{\Delta;\Gamma \vdash \mathsf{inr}\, v_2 : {}^{\xi}(\tau_1 \oplus \tau_2)}$$

T-Prod
$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Delta;\Gamma_1 \vdash v_1 : \tau_1 \qquad \Delta \vdash \tau_1 \preceq \xi \qquad \Delta;\Gamma_2 \vdash v_2 : \tau_2 \qquad \Delta \vdash \tau_2 \preceq \xi}{\Delta;\Gamma \vdash \langle v_1, v_2 \rangle : {}^{\xi}(\tau_1 \otimes \tau_2)}$$

T-Unit
$$\frac{\Delta \vdash \xi : \mathsf{QUAL}}{\Delta;\bullet \vdash \langle\rangle : {}^{\xi}1}$$

T-App
$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Delta;\Gamma_1 \vdash e_1 : {}^{\xi}(\tau_1 \multimap \tau_2) \qquad \Delta;\Gamma_2 \vdash e_2 : \tau_1}{\Delta;\Gamma \vdash e_1\, e_2 : \tau_2}$$

T-TApp
$$\frac{\Delta;\Gamma \vdash e : {}^{\xi}\forall \alpha{:}\kappa.\tau \qquad \Delta \vdash \iota : \kappa}{\Delta;\Gamma \vdash e\ \_ : \{\iota/\alpha\}\tau}$$

T-SumE
$$\frac{\Delta \vdash \xi' : \mathsf{QUAL} \qquad \Delta;\Gamma \vdash v_1 : {}^{\xi_1}(\tau_1 \multimap \tau) \qquad \Delta \vdash \xi_1 \preceq \xi \qquad \Delta;\Gamma \vdash v_2 : {}^{\xi_2}(\tau_2 \multimap \tau) \qquad \Delta \vdash \xi_2 \preceq \xi}{\Delta;\Gamma \vdash [v_1, v_2] : {}^{\xi}({}^{\xi'}(\tau_1 \oplus \tau_2) \multimap \tau)}$$

T-ProdE
$$\frac{\Delta \vdash \xi : \mathsf{QUAL} \qquad \Delta;\Gamma \vdash v : {}^{\xi'}(\tau_1 \multimap {}^{\xi'}(\tau_2 \multimap \tau))}{\Delta;\Gamma \vdash \mathsf{uncurry}\, v : {}^{\xi'}({}^{\xi}(\tau_1 \otimes \tau_2) \multimap \tau)}$$

T-UnitE
$$\frac{\Delta \vdash \xi : \mathsf{QUAL} \qquad \Delta \vdash \tau : \star \qquad \Delta;\Gamma \vdash v : {}^{\xi'}1}{\Delta;\Gamma \vdash \mathsf{ignore}\, v : {}^{\xi}(\tau \multimap \tau)}$$

Figure 8.7: λ^{URAL} statics (iv): typing

(continued in figure 8.8)

(continued from figure 8.7)

$\boxed{\Delta;\Gamma \vdash e : \tau}$ *(typing expressions)*

$$\textsc{T-NewUA}\ \frac{\mathsf{q} \preceq \mathsf{A} \qquad \Delta;\Gamma \vdash e : \tau \qquad \Delta \vdash \tau \preceq \mathsf{A}}{\Delta;\Gamma \vdash \mathsf{new}^{\mathsf{q}}\, e : {}^{\mathsf{q}}\,\mathsf{ref}\ \tau} \qquad \textsc{T-NewRL}\ \frac{\mathsf{R} \preceq \mathsf{q} \qquad \Delta;\Gamma \vdash e : \tau}{\Delta;\Gamma \vdash \mathsf{new}^{\mathsf{q}}\, e : {}^{\mathsf{q}}\,\mathsf{ref}\ \tau}$$

$$\textsc{T-SwapWeak}\ \frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Delta;\Gamma_1 \vdash e_1 : {}^{\xi}\,\mathsf{ref}\ \tau \qquad \Delta;\Gamma_2 \vdash e_2 : \tau}{\Delta;\Gamma \vdash \mathsf{swap}\, e_1\, e_2 : {}^{\mathsf{L}}({}^{\xi}\,\mathsf{ref}\ \tau \otimes \tau)}$$

$$\textsc{T-SwapStrong}\ \frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Delta;\Gamma_1 \vdash e_1 : {}^{\xi}\,\mathsf{ref}\ \tau_1 \qquad \Delta \vdash \mathsf{A} \preceq \xi \qquad \Delta;\Gamma_2 \vdash e_2 : \tau_2 \qquad \Delta \vdash \tau_2 \preceq \xi}{\Delta;\Gamma \vdash \mathsf{swap}\, e_1\, e_2 : {}^{\mathsf{L}}({}^{\xi}\,\mathsf{ref}\ \tau_2 \otimes \tau_1)}$$

$$\textsc{T-Read}\ \frac{\Delta;\Gamma \vdash e : {}^{\xi}\,\mathsf{ref}\ \tau \qquad \Delta \vdash \tau \preceq \mathsf{R}}{\Delta;\Gamma \vdash \mathsf{read}\, e : \tau} \qquad \textsc{T-Delete}\ \frac{\Delta;\Gamma \vdash e : {}^{\xi}\,\mathsf{ref}\ \tau \qquad \Delta \vdash \mathsf{A} \preceq \xi}{\Delta;\Gamma \vdash \mathsf{delete}\, e : \tau}$$

Figure 8.8: λ^{URAL} statics (v): typing

Figure 8.6 gives rules for splitting a type context into two ($\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$), which is necessary for distributing typing assumptions to multiple subterms of a term. Any variable may be distributed to one side or the other. Rule S-CONTRACT implements the contraction structural rule, whereby variables whose type is unlimited or relevant may be duplicated to both contexts.

Finally, figures 8.7 and 8.8 give the judgment for assigning types to expressions ($\Delta;\Gamma \vdash e : \tau$). Several points are worthy of note:

- The weakening rule, T-WEAK, allows discarding portions of the context that are upper-bounded by A, which means that all the values dropped are either affine or unlimited.

- The rules for application expressions and reference swapping, T-APP, T-SWAPSTRONG, and T-SWAPWEAK, split the context to distribute assumptions to subterms.

- Rule T-ABS selects a qualifier ξ for a function type based on bounding the context, Γ. This means that the qualifier of a function type must upper bound the qualifiers of the types of the function's free variables. As we will see in §8.4, this property is key to the soundness theorem.

8.3 Generic Control Effects in $\lambda^{\mathsf{URAL}}(\mathcal{C})$

Rather than add a specific control effect, such as exceptions or delimited continuations, to λ^{URAL}, I first describe a substructural type system with a general notion of control effect. Thus, this section defines a modified calculus, $\lambda^{\mathsf{URAL}}(\mathcal{C})$, parametrized by an unspecified control effect.

8.3.1 The Control Effect Parameter

Rather than incorporate a specific control effect, the definition of $\lambda^{\mathsf{URAL}}(\mathcal{C})$ is parametrized by an abstract control effect, whose form is described in this section. For the generic soundness theorem to hold, the actual control effect parameter must satisfy several properties, which are specified in §8.4. The three instances of control effects studied in §8.5 are proved to satisfy those properties.

DEFINITION 8.1 (Control effect)**.**

A control effect instance is a triple $(\mathcal{C}, \bot_{\mathcal{C}}, \otimes)$ *where* $\mathcal{C}$ *is a set of control effects (c),* $\bot_{\mathcal{C}} \in \mathcal{C}$ *is a distinguished* pure *effect that denotes no actual control, and* $\otimes \colon \mathcal{C} \times \mathcal{C} \rightharpoonup \mathcal{C}$ *is an associative, partial, binary operation denoting effect sequencing.*

For example, §8.5.3 adds exception handling to $\lambda^{\mathsf{URAL}}(\mathcal{C})$. An exception effect is the set of exceptions that may be raised by an expression, the distinguished pure effect $\bot_{\mathcal{C}}$ is the empty set, and sequencing is set union. A non-empty effect indicates that an expression may discard part of its continuation, whereas the empty effect guarantees that an expression treats its continuation linearly.

In simple cases, as with exceptions, effects form a join semilattice where sequencing is the join, but this is not necessarily true in general (§8.5.2).

k	::=		kinds
	|	CTL	control effects
	|	QUAL	qualifiers
	|	$\overline{\star}$	pretypes
	|	$\star$	types
i	::=		type-level terms
	|	c	control effect
	|	ξ	qualifier
	|	$\overline{t}$	pretype
	|	t	type
$\overline{t}$	::=		updated pretypes
	|	$\cdots$	other productions as before
	|	$t_1 \xrightarrow{c} t_2$	function with latent effect
	|	$\forall^c \alpha{:}k.\, t$	universal with latent effect
t	::=		updated types
	|	α	type variable
	|	${}^{\xi}\overline{t}$	qualified pretype

Figure 8.9: Updated syntax for $\lambda^{\mathsf{URAL}}(\mathcal{C})$

8.3.2 Updated Syntax

The syntax of $\lambda^{\mathsf{URAL}}(\mathcal{C})$ differs from that of λ^{URAL} in two ways:

- In $\lambda^{\mathsf{URAL}}(\mathcal{C})$, control effects constitute a fourth kind of type-level term, in addition to qualifiers, pretypes, and types. This requires a new kind CTL and the inclusion of abstract control effects ($c \in \mathcal{C}$) among the type-level terms.

- Function and universal pretypes now have latent effects, which record the effect that will happen when an abstraction is applied. The definition of pretypes is updated to include these latent effects; the other pretype productions remain unchanged.

The syntax of expressions, values, and qualifiers is unchanged from λ^{URAL} to $\lambda^{\mathsf{URAL}}(\mathcal{C})$.

The updated syntax for kinds, type-level terms, types, and pretypes appears in figure 8.9. For non-terminal symbols that differ between λ^{URAL} and $\lambda^{\mathsf{URAL}}(\mathcal{C})$, I use Roman letters ($t$, k, G, ...) for $\lambda^{\mathsf{URAL}}(\mathcal{C})$ to distinguish them from λ^{URAL}, where they appeared in Greek (τ, κ, Γ, ...).

8.3.3 Static Semantics of $\lambda^{\mathsf{URAL}}(\mathcal{C})$

All type system judgments from λ^{URAL} are updated for $\lambda^{\mathsf{URAL}}(\mathcal{C})$, and $\lambda^{\mathsf{URAL}}(\mathcal{C})$ adds two new judgments as well. The kinding and expression typing judgments are the only two to change significantly. The judgments for bounding types ($\mathrm{D} \vdash_{\mathcal{C}} t \preceq \xi$), bounding type contexts ($\mathrm{D} \vdash_{\mathcal{C}} \mathrm{G} \preceq \xi$), and splitting type contexts ($\mathrm{D} \vdash_{\mathcal{C}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_2$) are isomorphic to the λ^{URAL} versions of those judgments from Figures 8.5 and 8.6. They are merely updated with new non-terminals as appropriate (*i.e.,* κ to k, $\overline{\tau}$ to $\overline{t}$, and τ to t).

Kinding. Figure 8.10 shows one new kinding rule, C-K-BOT, which assigns kind CTL to the pure effect $\bot_{\mathcal{C}}$. Rules C-K-ARR and C-K-ALL are updated to account for latent effects in function and universal pretypes. The remaining kinding rules are the same as for λ^{URAL}, with non-terminals *mutatis mutandis*. Specific control effect instances (§8.5) must define additional kinding rules for their particular effects.

Control effect judgments. The first new judgment for control effects ($\mathrm{D} \vdash_{\mathcal{C}} c \succeq \xi$, figure 8.11) relates control effects to qualifiers. This gives the meaning of a control effect in terms of a lower bound on how an expression with that effect may treat its own continuation. For example, if an expression e has some effect c such that $\mathrm{D} \vdash_{\mathcal{C}} c \succeq \mathsf{A}$, this indicates that e may drop but not duplicate its continuation. Two rules appear here:

- Rule C-B-PURE says that the pure effect is bounded by any qualifier, which means that a pure expression satisfies any requirement for how it treats its continuation.

$\boxed{\mathrm{D} \vdash_{\mathcal{C}} i : k}$ *(kinding type-level terms)*

$$\textsc{C-K-Bot}\ \frac{}{\mathrm{D} \vdash_{\mathcal{C}} \bot_{\mathcal{C}} : \mathsf{CTL}} \qquad \textsc{C-K-Arr}\ \frac{\mathrm{D} \vdash_{\mathcal{C}} t_1 : \star \qquad \mathrm{D} \vdash_{\mathcal{C}} t_2 : \star \qquad \mathrm{D} \vdash_{\mathcal{C}} c : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{C}} t_1 \xrightarrow{c} t_2 : \overline{\star}}$$

$$\textsc{C-K-All}\ \frac{\mathrm{D}, \alpha{:}k \vdash_{\mathcal{C}} t : \star \qquad \mathrm{D} \vdash_{\mathcal{C}} c : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{C}} \forall^{c} \alpha{:}k.\, t : \overline{\star}}$$

Figure 8.10: $\lambda^{\mathsf{URAL}}(\mathcal{C})$ statics (i): updated kinding rules

$\boxed{\mathrm{D} \vdash_{\mathcal{C}} c \succeq \xi}$ *(qualifier bound for control effects)*

$$\textsc{C-B-Pure}\ \frac{\mathrm{D} \vdash_{\mathcal{C}} \xi : \mathsf{QUAL}}{\mathrm{D} \vdash_{\mathcal{C}} \bot_{\mathcal{C}} \succeq \xi} \qquad \textsc{C-B-Unl}\ \frac{\mathrm{D} \vdash_{\mathcal{C}} c : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{C}} c \succeq \mathsf{U}}$$

$\boxed{\mathrm{D} \vdash_{\mathcal{C}} c_1 \preceq c_2}$ *(control effect subsumption)*

$$\textsc{CSub-Refl}\ \frac{\mathrm{D} \vdash_{\mathcal{C}} c : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{C}} c \preceq c} \qquad \textsc{CSub-Trans}\ \frac{\mathrm{D} \vdash_{\mathcal{C}} c_1 \preceq c' \qquad \mathrm{D} \vdash_{\mathcal{C}} c' \preceq c_2}{\mathrm{D} \vdash_{\mathcal{C}} c_1 \preceq c_2}$$

Figure 8.11: $\lambda^{\mathsf{URAL}}(\mathcal{C})$ statics (ii): control effect judgments

$\boxed{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} e : t ; c}$ *(typing expressions)*

C-T-SUBSUME
$$\frac{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} e : t ; c' \qquad \mathrm{D} \vdash_{\mathcal{C}} c' \preceq c}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} e : t ; c}$$

C-T-WEAK
$$\frac{\mathrm{D} \vdash_{\mathcal{C}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_2 \qquad \mathrm{D} \vdash_{\mathcal{C}} \mathrm{G}_2 \preceq \mathsf{A} \qquad \mathrm{D};\mathrm{G}_1 \vdash_{\mathcal{C}} e : t ; c}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} e : t ; c}$$

C-T-VAR
$$\frac{\mathrm{D} \vdash_{\mathcal{C}} t : \star}{\mathrm{D};\bullet, x{:}t \vdash_{\mathcal{C}} x : t ; \bot_{\mathcal{C}}}$$

C-T-ABS
$$\frac{\mathrm{D} \vdash_{\mathcal{C}} \mathrm{G} \preceq \xi \qquad \mathrm{D};\mathrm{G}, x{:}t_1 \vdash_{\mathcal{C}} e : t_2 ; c}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} \lambda x.e : {}^{\xi}(t_1 \xrightarrow{c}\!\circ\, t_2) ; \bot_{\mathcal{C}}}$$

C-T-TABS
$$\frac{\mathrm{D} \vdash_{\mathcal{C}} \mathrm{G} \preceq \xi \qquad \mathrm{D}, \alpha{:}k;\mathrm{G} \vdash_{\mathcal{C}} e : t ; c}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} \Lambda.e : {}^{\xi}\forall^{c}\alpha{:}k.t ; \bot_{\mathcal{C}}}$$

C-T-INL
$$\frac{\mathrm{D} \vdash_{\mathcal{C}} t_1 \preceq \xi \qquad \mathrm{D} \vdash_{\mathcal{C}} t_2 : \star \qquad \mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} v_1 : t_1 ; \bot_{\mathcal{C}}}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} \mathsf{inl}\, v_1 : {}^{\xi}(t_1 \oplus t_2) ; \bot_{\mathcal{C}}}$$

C-T-INR
$$\frac{\mathrm{D} \vdash_{\mathcal{C}} t_1 : \star \qquad \mathrm{D} \vdash_{\mathcal{C}} t_2 \preceq \xi \qquad \mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} v_2 : t_2 ; \bot_{\mathcal{C}}}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} \mathsf{inr}\, v_2 : {}^{\xi}(t_1 \oplus t_2) ; \bot_{\mathcal{C}}}$$

C-T-PROD
$$\frac{\mathrm{D} \vdash_{\mathcal{C}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_2 \qquad \mathrm{D};\mathrm{G}_1 \vdash_{\mathcal{C}} v_1 : t_1 ; \bot_{\mathcal{C}} \qquad \mathrm{D} \vdash_{\mathcal{C}} t_1 \preceq \xi \qquad \mathrm{D};\mathrm{G}_2 \vdash_{\mathcal{C}} v_2 : t_2 ; \bot_{\mathcal{C}} \qquad \mathrm{D} \vdash_{\mathcal{C}} t_2 \preceq \xi}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} \langle v_1, v_2 \rangle : {}^{\xi}(t_1 \otimes t_2) ; \bot_{\mathcal{C}}}$$

C-T-UNIT
$$\frac{\mathrm{D} \vdash_{\mathcal{C}} \xi : \mathsf{QUAL}}{\mathrm{D};\bullet \vdash_{\mathcal{C}} \langle\rangle : {}^{\xi}1 ; \bot_{\mathcal{C}}}$$

C-T-APP
$$\frac{\mathrm{D};\mathrm{G}_1 \vdash_{\mathcal{C}} e_1 : {}^{\xi_1}(t_1 \xrightarrow{c}\!\circ\, t_2) ; c_1 \qquad \mathrm{D};\mathrm{G}_2 \vdash_{\mathcal{C}} e_2 : t_1 ; c_2 \qquad \mathrm{D} \vdash_{\mathcal{C}} \mathrm{G}_2 \preceq \xi_2 \qquad \mathrm{D} \vdash_{\mathcal{C}} c_1 \succeq \xi_2 \qquad \mathrm{D} \vdash_{\mathcal{C}} c_2 \succeq \xi_1 \qquad \mathrm{D} \vdash_{\mathcal{C}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_2 \qquad \mathrm{D} \vdash_{\mathcal{C}} c_1 \oslash c_2 \oslash c : \mathsf{CTL}}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} e_1\, e_2 : t_2 ; c_1 \oslash c_2 \oslash c}$$

C-T-TAPP
$$\frac{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} e : {}^{\xi}\forall^{c'}\alpha{:}k.t ; c \qquad \mathrm{D} \vdash_{\mathcal{C}} i : k \qquad \mathrm{D} \vdash_{\mathcal{C}} c \oslash c' : \mathsf{CTL}}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} e\ \_ : \{i/\alpha\}t ; c \oslash c'}$$

C-T-SUME
$$\frac{\mathrm{D} \vdash_{\mathcal{C}} \xi' : \mathsf{QUAL} \qquad \mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} v_1 : {}^{\xi_1}(t_1 \xrightarrow{c}\!\circ\, t) ; \bot_{\mathcal{C}} \qquad \mathrm{D} \vdash_{\mathcal{C}} \xi_1 \preceq \xi \qquad \mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} v_2 : {}^{\xi_2}(t_2 \xrightarrow{c}\!\circ\, t) ; \bot_{\mathcal{C}} \qquad \mathrm{D} \vdash_{\mathcal{C}} \xi_2 \preceq \xi}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} [v_1, v_2] : {}^{\xi}({}^{\xi'}(t_1 \oplus t_2) \xrightarrow{c}\!\circ\, t) ; \bot_{\mathcal{C}}}$$

C-T-PRODE
$$\frac{\mathrm{D} \vdash_{\mathcal{C}} \xi : \mathsf{QUAL} \qquad \mathrm{D} \vdash_{\mathcal{C}} c_1 \oslash c_2 : \mathsf{CTL} \qquad \mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} v : {}^{\xi'}(t_1 \xrightarrow{c_1}\!\circ\, {}^{\xi'}(t_2 \xrightarrow{c_2}\!\circ\, t)) ; \bot_{\mathcal{C}}}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} \mathsf{uncurry}\, v : {}^{\xi'}({}^{\xi}(t_1 \otimes t_2) \xrightarrow{c_1 \oslash c_2}\!\circ\, t) ; \bot_{\mathcal{C}}}$$

C-T-UNITE
$$\frac{\mathrm{D} \vdash_{\mathcal{C}} \xi : \mathsf{QUAL} \qquad \mathrm{D} \vdash_{\mathcal{C}} t : \star \qquad \mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} v : {}^{\xi'}1 ; \bot_{\mathcal{C}}}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} \mathsf{ignore}\, v : {}^{\xi}(t \xrightarrow{\bot_{\mathcal{C}}}\!\circ\, t) ; \bot_{\mathcal{C}}}$$

Figure 8.12: $\lambda^{\mathsf{URAL}}(\mathcal{C})$ statics (iii): typing

(continued in figure 8.13)

- Rule C-B-UNL says that all control effects are bounded by U, which means that one may assume, conservatively, that any expression might freely duplicate or drop its continuation.

Specific instances of the control effect parameter will extend this judgment to take into account the properties of a particular control effect.

The second judgment for control effects ($\mathrm{D} \vdash_{\mathcal{C}} c_1 \preceq c_2$) defines a subsumption order for control effects. This means that an expression whose effect is c_1 may be safely considered to have effect c_2. Only two rules for the judgment appear in figure 8.11, which together ensure that control effect subsumption is a preorder. As with control effect bounding, specific control effect instances will extend this judgment.

Expression typing. The expression typing judgment for $\lambda^{\mathsf{URAL}}(\mathcal{C})$, which appears in figure 8.12, assigns not only a type t but an effect c to expressions: $\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} e : t ; c$. Having seven premises, the rule for applications (C-T-APP) is unwieldy, but it likely gives the most insight into how $\lambda^{\mathsf{URAL}}(\mathcal{C})$ works:

$$\frac{\begin{array}{ll}(1) & \mathrm{D} \vdash_{\mathcal{C}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_2 \\ (2) & \mathrm{D};\mathrm{G}_1 \vdash_{\mathcal{C}} e_1 : {}^{\xi_1}(t_1 \xrightarrow{c} t_2) ; c_1 \\ (3) & \mathrm{D};\mathrm{G}_2 \vdash_{\mathcal{C}} e_2 : t_1 ; c_2 \\ (4) & \mathrm{D} \vdash_{\mathcal{C}} c_2 \succeq \xi_1 \\ (5) & \mathrm{D} \vdash_{\mathcal{C}} \mathrm{G}_2 \preceq \xi_2 \\ (6) & \mathrm{D} \vdash_{\mathcal{C}} c_1 \succeq \xi_2 \\ (7) & \mathrm{D} \vdash_{\mathcal{C}} c_1 \oslash c_2 \oslash c : \mathsf{CTL}\end{array}}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} e_1 e_2 : t_2 ; c_1 \oslash c_2 \oslash c}$$

Let us consider the premises in order:

(1) The first premise, as in λ^{URAL}, splits the type context G into G_1 for typing e_1 and G_2 for typing e_2.

(2–3) As in λ^{URAL}, these premises assign types to expressions e_1 and e_2, but they assign control effects c_1 and c_2 as well.

(continued from figure 8.12)

$\boxed{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} e : t ; c}$ *(typing expressions)*

C-T-NewUA
$$\frac{\mathsf{q} \preceq \mathsf{A} \qquad \mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} e : t ; c \qquad \mathrm{D} \vdash_{\mathcal{C}} t \preceq \mathsf{A}}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} \mathsf{new}^{\mathsf{q}}\, e : {}^{\mathsf{q}}\mathsf{ref}\, t ; c}$$

C-T-NewRL
$$\frac{\mathsf{R} \preceq \mathsf{q} \qquad \mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} e : t ; c}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} \mathsf{new}^{\mathsf{q}}\, e : {}^{\mathsf{q}}\mathsf{ref}\, t ; c}$$

C-T-SwapWeak
$$\frac{\begin{array}{c}\mathrm{D} \vdash_{\mathcal{C}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_2 \\ \mathrm{D};\mathrm{G}_1 \vdash_{\mathcal{C}} e_1 : {}^{\xi_1}\mathsf{ref}\, t ; c_1 \\ \mathrm{D};\mathrm{G}_2 \vdash_{\mathcal{C}} e_2 : t ; c_2 \qquad \mathrm{D} \vdash_{\mathcal{C}} \mathrm{G}_2 \preceq \xi_2 \\ \mathrm{D} \vdash_{\mathcal{C}} c_1 \succeq \xi_2 \qquad \mathrm{D} \vdash_{\mathcal{C}} c_2 \succeq \xi_1 \qquad \mathrm{D} \vdash_{\mathcal{C}} c_1 \oslash c_2 : \mathsf{CTL}\end{array}}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} \mathsf{swap}\, e_1\, e_2 : {}^{\mathsf{L}}({}^{\xi}\mathsf{ref}\, t \otimes t) ; c_1 \oslash c_2}$$

C-T-SwapStrong
$$\frac{\begin{array}{c}\mathrm{D} \vdash_{\mathcal{C}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_2 \\ \mathrm{D};\mathrm{G}_1 \vdash_{\mathcal{C}} e_1 : {}^{\xi_1}\mathsf{ref}\, t_1 ; c_1 \\ \mathrm{D};\mathrm{G}_2 \vdash_{\mathcal{C}} e_2 : t_2 ; c_2 \qquad \mathrm{D} \vdash_{\mathcal{C}} \mathrm{G}_2 \preceq \xi_2 \\ \mathrm{D} \vdash_{\mathcal{C}} c_1 \succeq \xi_2 \qquad \mathrm{D} \vdash_{\mathcal{C}} c_2 \succeq \xi_1 \\ \mathrm{D} \vdash_{\mathcal{C}} \mathsf{A} \preceq \xi_1 \qquad \mathrm{D} \vdash_{\mathcal{C}} t_2 \preceq \xi_1 \qquad \mathrm{D} \vdash_{\mathcal{C}} c_1 \oslash c_2 : \mathsf{CTL}\end{array}}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} \mathsf{swap}\, e_1\, e_2 : {}^{\mathsf{L}}({}^{\xi}\mathsf{ref}\, t_2 \otimes t_1) ; c_1 \oslash c_2}$$

C-T-Read
$$\frac{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} e : {}^{\xi}\mathsf{ref}\, t ; c \qquad \mathrm{D} \vdash_{\mathcal{C}} t \preceq \mathsf{R}}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} \mathsf{read}\, e : t ; c}$$

C-T-Delete
$$\frac{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} e : {}^{\xi}\mathsf{ref}\, t ; c \qquad \mathrm{D} \vdash_{\mathcal{C}} \mathsf{A} \preceq \xi}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} \mathsf{delete}\, e : t ; c}$$

Figure 8.13: $\lambda^{\mathsf{URAL}}(\mathcal{C})$ statics (iv): typing

(4) This premise relates the type of e_1 to the effect of e_2 to ensure that e_2's effect does not violate e_1's invariants. Because of λ^{URAL}'s left-to-right evaluation order, by the time e_2 gets to run, e_1 has reduced to a value of type ${}^{\xi_1}(t_1 \xrightarrow{c} t_2)$, which thus may be treated according to qualifier ξ_1. Because that value is part of e_2's continuation, this premise requires that e_2's effect, c_2, be lower-bounded by ξ_1. In other words, e_2 will treat its continuation no more liberally than ξ_1 allows.

(5–6) These premises relate the free variables of e_2 to the effect of e_1. Due to the evaluation order, e_2 appears unevaluated in e_1's continuation, which means that if e_1 drops or duplicates its continuation then e_2 may be evaluated never or more than once. Premise (5) says that the type context for typing e_2, and thus e_2's free variables, are bounded above by some qualifier ξ_2, and this qualifier thus indicates how many times it is safe to evaluate e_2. Premise (6) lower bounds e_1's effect, c_1, by ξ_2, ensuring that e_1's effect treats e_2 properly.

(7) The net effect of the application expression is a sequence of the effect of e_1 (c_1), then the effect of e_2 (c_2), and finally the latent effect of the function to which e_1 must evaluate (c): $c_1 \oslash c_2 \oslash c$. This premise checks that those three effects may be sequenced in that order according to a particular control effect's definition of sequencing and the kinding judgment.

Rules C-T-SWAPSTRONG and C-T-SWAPSTRONG (reference swap) are similar, since they need to safely sequence two subexpressions. Both rules follow rule C-T-APP in relating the effect of the first subexpression to the type context of the second and effect of the second to the qualifier of the first. Rule C-T-TAPP (type application), while dealing with only one effectful subexpression, needs to sequence the effect of evaluating the expression in a type application with the latent effect of the resulting type abstraction value.

The subsumption rule C-T-SUBSUME implements control effect subsumption, whereby an expression of effect c may also be considered to have effect c' if c is less than c' in the control effect subsumption order. C-T-WEAK, which handles weakening, is unchanged from λ^{URAL}.

The remaining rules are for typing values, which always have the pure effect $\bot_{\mathcal{C}}$. Rules C-T-UNIT, C-T-INL, and C-T-INR, for unit and sum introduction, are unchanged from λ^{URAL}, except that each of them assigns the pure effect. Rules C-T-ABS and C-T-TABS also assign the pure effect to their values, but each records the effect of the abstraction body as the latent effect in the resulting type.

8.4 The Generic Theory

To prove type soundness for $\lambda^{\mathsf{URAL}}(\mathcal{C})$, I define a type-preserving translation to λ^{URAL}. Instead of a direct reduction semantics for $\lambda^{\mathsf{URAL}}(\mathcal{C})$, its operational semantics is defined in terms of the translation and the reduction semantics of λ^{URAL} (§8.2.1). Thus, it is sufficient to show that all well-typed $\lambda^{\mathsf{URAL}}(\mathcal{C})$ programs translate to well-typed λ^{URAL} programs, in order to apply λ^{URAL}'s type soundness theorem to $\lambda^{\mathsf{URAL}}(\mathcal{C})$ as well.

The translation is into what Danvy and Filinski (1989) call *continuation-composing style* (henceforth "CCoS"). It is similar to continuation-passing style, but unlike continuation-passing style it still relies on the object language's order of evaluation.

In order to specify the translation and prove the propositions later in this section, I impose several more requirements on the abstract control effect parameter. As the semantics of $\lambda^{\mathsf{URAL}}(\mathcal{C})$ was parametrized by an abstract control effect, so is the theory of $\lambda^{\mathsf{URAL}}(\mathcal{C})$ parametrized by several definitions and properties that a control effect must satisfy.

Because the development of this section is constrained by several dependencies, this outline may be helpful:

The Translation Parameter (§8.4.1). A control effect instance must supply a few definitions to fully specify its particular CCoS translation.

The Translation (§8.4.2). The definition of the CCoS translation relies on the definitions supplied by the control effect parameter.

Parameter Properties (§8.4.3). A control effect instance must satisfy several properties on which the generic type soundness theorem relies.

Generic Type Soundness (§8.4.4). The section culminates in a generic proof of type soundness for $\lambda^{\mathsf{URAL}}(\mathcal{C})$.

8.4.1 The Translation Parameter

DEFINITION 8.2 (Translation parameter)**.**

The definition of the generic CCoS translation relies on the following effect-specific definitions:

- *a metafunction* $(\cdot)^*$ *from effects to qualifiers, such that* $\bot_{\mathcal{C}}{}^* = \mathsf{L}$ *and* $\alpha^* = \alpha$*;*
- *a value* $\mathsf{done}_{\mathcal{C}}$*, to use as the initial continuation for a CCoSed program; and*
- *a pair of* answer-type *metafunctions* $\langle\!\langle \cdot, \cdot \rangle\!\rangle_c^-$ *and* $\langle\!\langle \cdot, \cdot \rangle\!\rangle_c^+$*, each of which maps a* λ^{URAL} *type and a* $\lambda^{URAL}(\mathcal{C})$ *effect to a* λ^{URAL} *type.*

Intuitively, one may understand metafunctions $(\cdot)^*$, $\langle\!\langle \cdot, \cdot \rangle\!\rangle_c^-$, and $\langle\!\langle \cdot, \cdot \rangle\!\rangle_c^+$ as relating the effect of a $\lambda^{\mathsf{URAL}}(\mathcal{C})$ expression to the type of its translation into λ^{URAL}. Typically, the CPS translation of an expression of some type τ yields a type like

$$(\tau \to \mathsf{Answer}) \to \mathsf{Answer}.$$

Given a $\lambda^{\mathsf{URAL}}(\mathcal{C})$ expression whose translated type is τ and whose effect is c, the translation to λ^{URAL} yields type

$$^{c^*}(\tau \multimap \langle\!\langle \tau_0, c \rangle\!\rangle_{\mathcal{C}}^-) \multimap \langle\!\langle \tau_0, c \rangle\!\rangle_{\mathcal{C}}^+$$

for some answer type τ_0. That is, $(\cdot)^*$ gives the qualifier of the continuation, and the other two metafunctions give the answer types, which may depend on the nature of the control effect. Because they give the answer types in negative and positive positions, respectively, I call $\langle\!\langle \tau, c \rangle\!\rangle_{\mathcal{C}}^-$ the *negative answer type* and $\langle\!\langle \tau, c \rangle\!\rangle_{\mathcal{C}}^+$ the *positive answer type*.

$$\begin{aligned}
\mathsf{QUAL}^* &= \mathsf{QUAL} && \textit{(kinds)}\\
\overline{\star}^* &= \overline{\star}\\
\star^* &= \star\\
\mathsf{CTL}^* &= \mathsf{QUAL}\\
\bullet^* &= \bullet && \textit{(kind contexts)}\\
(\mathrm{D}, \alpha{:}k)^* &= \mathrm{D}^*, \alpha{:}k^*
\end{aligned}$$

Figure 8.14: CCoS translation (i): kinds and kind contexts

8.4.2 The Translation

This subsection gives the CCoS translation from $\lambda^{\mathsf{URAL}}(\mathcal{C})$ to λ^{URAL}. In several places, the translation relies on the definitions of c^*, $\mathsf{done}_{\mathcal{C}}$, $\langle\!\langle \tau, c \rangle\!\rangle^-_{\mathcal{C}}$, and $\langle\!\langle \tau, c \rangle\!\rangle^+_{\mathcal{C}}$ supplied by the control effect parameter.

The translation for kinds and kind contexts appears in figure 8.14. The control effect kind CTL translates to QUAL, and the other three kinds translate to themselves. The translation of a kind context merely translates each kind in its range.

Figure 8.15 presents the translation for pretypes, types, and type contexts. Most of this translation is straightforward: type variables and the unit pretype translate to themselves, sum, product, and reference types translate homomorphically, types composed of a qualifier and a pretype translate the pretype, and type contexts translate all the types in their range. The two interesting cases are for function and universal pretypes. These follow the usual CPS translation for function and universal types, with several refinements:

- Each adds an extra universal quantifier in front of its result, which is used to make (type) abstractions polymorphic in their answer types.
- Because the effect of an expression limits how it may use its continuation, the translation c^* of latent effect c becomes the qualifier of the continuation.

$$
\begin{aligned}
\alpha^* &= \alpha && \textit{(pretypes)}\\
1^* &= 1\\
(t_1 \oplus t_2)^* &= {t_1}^* \oplus {t_2}^*\\
(t_1 \otimes t_2)^* &= {t_1}^* \otimes {t_2}^*\\
(\mathsf{ref}\ t)^* &= \mathsf{ref}\ t^*\\
(t_1 \xrightarrow{c} t_2)^* &= \forall\alpha{:}\star.\,\mathsf{L}({t_1}^* \multimap \mathsf{L}({c^*}({t_2}^* \multimap \langle\!\langle \alpha, c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c \rangle\!\rangle^+_{\mathcal{C}}))\\
(\forall^c \beta{:}k.\,t)^* &= \forall\alpha{:}\star.\,\mathsf{L}\forall\beta{:}k^*.\,\mathsf{L}({c^*}(t^* \multimap \langle\!\langle \alpha, c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c \rangle\!\rangle^+_{\mathcal{C}})\\
\\
\alpha^* &= \alpha && \textit{(types)}\\
({}^{\xi}\overline{t})^* &= {}^{\xi}\overline{t}^*\\
\\
\bullet^* &= \bullet && \textit{(type contexts)}\\
(\mathrm{G}, x{:}t)^* &= \mathrm{G}^*, x{:}t^*
\end{aligned}
$$

Figure 8.15: CCoS translation (ii): type-level terms and contexts

- All other qualifiers of the translated pretype are L. (This is because the translation never needs to duplicate partially-applied continuations, so L is a sufficiently permissive qualifier for those continuations. Furthermore, because the type rules for abstractions always allow a qualifier of L, using L wherever possible simplifies the proof.)

Translation of values and expressions is defined by mutual induction in figure 8.16. Value translation (v^*) is mostly straightforward. Both value and type abstraction have an additional type abstraction added to the front, which matches the addition of the universal quantifier in the type translation, and both translate the body according to the expression translation $[\![e]\!]_{\mathcal{C}}$. The expression translation is standard except for two unusual aspects of the translation of applications and type applications:

- The result of evaluating e_1, bound to x_1, is in each case instantiated by a type application, which compensates for the new type abstraction in the translation of abstractions. For the type application case, $x_1\_$

is instantiated then again, corresponding to the instantiation from the source expression.

- Curiously, the continuation y is η-expanded to $\lambda x.\, y\, x$. While η-expanding a variable may seem useless, it is actually necessary to obtain a type-preserving translation.

In particular, the reason for this η expansion is to handle effect subsumption. Effects in $\lambda^{\mathsf{URAL}}(\mathcal{C})$ are translated to qualifiers in λ^{URAL}, and while $\lambda^{\mathsf{URAL}}(\mathcal{C})$ supports effect subsumption directly, there is no analogous qualifier subsumption in λ^{URAL}. However, qualifier subsumption for function types can be explicitly achieved using η expansion.

LEMMA 8.3 (Dereliction)**.**

If $\Delta;\Gamma \vdash v : {}^{\xi}(\tau_1 \multimap \tau_2)$ *and* $\Delta \vdash \xi \preceq \xi'$ *then* $\Delta;\Gamma \vdash \lambda x.\, v\, x : {}^{\xi'}(\tau_1 \multimap \tau_2)$.

The proof of lemma 8.3 relies on another lemma:

LEMMA 8.4 (Value strengthening)**.**

Any qualifier that upper bounds the type of a value also bounds the portion of the type context necessary for typing that value. That is, if $\Delta;\Gamma \vdash v : \tau$ *and* $\Delta \vdash \tau \preceq \xi$ *then there exist some* Γ_1 *and* Γ_2 *such that*

- $\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$,
- $\Delta;\Gamma_1 \vdash v : \tau$,
- $\Delta \vdash \Gamma_1 \preceq \xi$, *and*
- $\Delta \vdash \Gamma_2 \preceq \mathsf{A}$.

Proof. Please see p. 391. ▷

Proof of lemma 8.3. Choose type contexts Γ_1 and Γ_2 according to lemma 8.4. Then $\Delta;\Gamma_1, x{:}\tau_1 \vdash v\, x : \tau_2$ by rule T-APP. By induction on the length of Γ_1 and transitivity of qualifier subsumption, we know that $\Delta \vdash \Gamma_1 \preceq \xi'$. Then by rule T-ABS, $\Delta;\Gamma_1 \vdash \lambda x.\, v\, x : {}^{\xi'}(\tau_1 \multimap \tau_2)$, and we change Γ_1 to Γ by rule T-WEAK.

Please see p. 395 for additional details. ▷

$$
\begin{aligned}
x^* &= x && \textit{(values)}\\
(\lambda x.\, e)^* &= \Lambda.\, \lambda x.\, [\![e]\!]_{\mathcal{C}}\\
(\Lambda.\, e)^* &= \Lambda.\, \Lambda.\, [\![e]\!]_{\mathcal{C}}\\
(\mathsf{inl}\ v)^* &= \mathsf{inl}\ v^*\\
(\mathsf{inr}\ v)^* &= \mathsf{inr}\ v^*\\
[v_1, v_2]^* &= \Lambda.\, [\lambda x.\, {v_1}^*\ \_\, x, \lambda x.\, {v_2}^*\ \_\, x]\\
\langle v_1, v_2\rangle^* &= \langle {v_1}^*, {v_2}^*\rangle\\
(\mathsf{uncurry}\ v)^* &= \Lambda.\, \mathsf{uncurry}\,(\lambda x_1.\, \lambda x_2.\, [\![v\, x_1\, x_2]\!]_{\mathcal{C}})\\
\langle\rangle^* &= \langle\rangle\\
(\mathsf{ignore}\ v)^* &= \Lambda.\, \lambda x.\, \mathsf{ignore}\ v^*\ [\![x]\!]_{\mathcal{C}}
\end{aligned}
$$

$$
\begin{aligned}
[\![v]\!]_{\mathcal{C}} &= \lambda y.\, y\, v^* && \textit{(expressions)}\\
[\![e_1\, e_2]\!]_{\mathcal{C}} &= \lambda y.\, [\![e_1]\!]_{\mathcal{C}}\ (\lambda x_1.\, [\![e_2]\!]_{\mathcal{C}}\ (\lambda x_2.\, x_1\ \_\, x_2\, (\lambda x.\, y\, x)))\\
[\![e\ \_]\!]_{\mathcal{C}} &= \lambda y.\, [\![e]\!]_{\mathcal{C}}\ (\lambda x_1.\, x_1\ \_\ \_\, (\lambda x.\, y\, x))\\
[\![\mathsf{new}^{\mathsf{q}}\, e]\!]_{\mathcal{C}} &= \lambda y.\, [\![e]\!]_{\mathcal{C}}\ (\lambda x.\, y\, (\mathsf{new}^{\mathsf{q}}\, x))\\
[\![\mathsf{read}\ e]\!]_{\mathcal{C}} &= \lambda y.\, [\![e]\!]_{\mathcal{C}}\ (\lambda x.\, y\, (\mathsf{read}\ x))\\
[\![\mathsf{delete}\ e]\!]_{\mathcal{C}} &= \lambda y.\, [\![e]\!]_{\mathcal{C}}\ (\lambda x.\, y\, (\mathsf{delete}\ x))\\
[\![\mathsf{swap}\ e_1\, e_2]\!]_{\mathcal{C}} &= \lambda y.\, [\![e_1]\!]_{\mathcal{C}}\ (\lambda x_1.\, [\![e_2]\!]_{\mathcal{C}}\ (\lambda x_2.\, y\, (\mathsf{swap}\ x_1\, x_2)))
\end{aligned}
$$

Figure 8.16: CCoS translation (iii): values and expressions

Having defined the translation, we run a program e by applying the CCoS translation and passing it the initial continuation $\mathsf{done}_{\mathcal{C}}$. Define the operational semantics of $\lambda^{\mathsf{URAL}}(\mathcal{C})$ as a partial function $\mathit{eval} : \mathit{Expressions} \rightharpoonup \mathit{Values} \cup \{\textsc{Wrong}\}$:

$$\mathit{eval}(e) = \begin{cases} v & \text{if } [\![e]\!]_{\mathcal{C}}\, \mathsf{done}_{\mathcal{C}} \xrightarrow{*} v; \\ \textsc{Wrong} & \text{if } [\![e]\!]_{\mathcal{C}}\, \mathsf{done}_{\mathcal{C}} \xrightarrow{*} e' \\ & \quad \text{such that } e' \text{ is not a value} \\ & \quad \text{and } \neg(\exists e'')\, e' \longmapsto e''. \end{cases}$$

8.4.3 Parameter Properties

Having defined the CCoS translation, it is now possible to state the additional properties that the abstract control effect parameter must satisfy for the generic type soundness theorem (§8.4.4) to hold:

Parameter Property 1 (Answer types)**.**

1. *For all* τ, $\langle\!\langle \tau, \bot_{\mathcal{C}} \rangle\!\rangle^{-}_{\mathcal{C}} = \langle\!\langle \tau, \bot_{\mathcal{C}} \rangle\!\rangle^{+}_{\mathcal{C}}$.

   Rationale. For pure expressions, the negative and positive answer types agree, because a pure expression finishes by calling its continuation. Henceforth, we are justified defining the *pure answer type* $\langle\!\langle \tau \rangle\!\rangle_{\mathcal{C}} \triangleq \langle\!\langle \tau, \bot_{\mathcal{C}} \rangle\!\rangle^{+}_{\mathcal{C}}$.

2. *If* $\mathrm{D}^* \vdash \tau : \star$ *and* $\mathrm{D} \vdash_{\mathcal{C}} c : \mathsf{CTL}$ *then* $\mathrm{D}^* \vdash \langle\!\langle \tau, c \rangle\!\rangle^{-}_{\mathcal{C}} : \star$ *and* $\mathrm{D}^* \vdash \langle\!\langle \tau, c \rangle\!\rangle^{+}_{\mathcal{C}} : \star$.

   Rationale. For the translation to be well typed, well-kinded types and effects must become well-kinded answer types.

3. *For all* D, τ, $c_1 \neq \bot_{\mathcal{C}}$, *and* $c_2 \neq \bot_{\mathcal{C}}$ *such that* $\mathrm{D} \vdash_{\mathcal{C}} c_1 \oslash c_2 : \mathsf{CTL}$,

   a) $\langle\!\langle \tau, c_1 \oslash c_2 \rangle\!\rangle^{-}_{\mathcal{C}} = \langle\!\langle \tau, c_2 \rangle\!\rangle^{-}_{\mathcal{C}}$,

   b) $\langle\!\langle \tau, c_1 \oslash c_2 \rangle\!\rangle^{+}_{\mathcal{C}} = \langle\!\langle \tau, c_1 \rangle\!\rangle^{+}_{\mathcal{C}}$, *and*

   c) $\langle\!\langle \tau, c_1 \rangle\!\rangle^{-}_{\mathcal{C}} = \langle\!\langle \tau, c_2 \rangle\!\rangle^{+}_{\mathcal{C}}$.

   Rationale. Effect sequencing must maintain answer types in order for the continuations of sequenced expressions to compose.

4. *If* $\mathrm{D} \vdash_{\mathcal{C}} c_1 \preceq c_2$*, then for every type* τ *there exists some type* τ' *such that* $\langle\!\langle \tau', c_1 \rangle\!\rangle^-_{\mathcal{C}} = \langle\!\langle \tau, c_2 \rangle\!\rangle^-_{\mathcal{C}}$ *and* $\langle\!\langle \tau', c_1 \rangle\!\rangle^+_{\mathcal{C}} = \langle\!\langle \tau, c_2 \rangle\!\rangle^+_{\mathcal{C}}$.

   RATIONALE. For control effect subsumption to be valid, related control effects must generate related answer types.

PARAMETER PROPERTY 2 (Done)**.**

If $\Delta \vdash \tau \preceq \mathsf{A}$ *then* $\Delta; \bullet \vdash \mathsf{done}_{\mathcal{C}} : {}^{\mathsf{L}}(\tau \multimap \langle\!\langle \tau \rangle\!\rangle_{\mathcal{C}})$.

   RATIONALE. The $\mathsf{done}_{\mathcal{C}}$ value must be well typed for the translation of a whole program to be well typed.

PARAMETER PROPERTY 3 (Effect sequencing)**.**

If $\mathrm{D} \vdash_{\mathcal{C}} c_1 \mathbin{⊙} c_2 : \mathsf{CTL}$ *then* $\mathrm{D}^* \vdash (c_1 \mathbin{⊙} c_2)^* \preceq {c_1}^*$ *and* $\mathrm{D}^* \vdash (c_1 \mathbin{⊙} c_2)^* \preceq {c_2}^*$.

   RATIONALE. Sequencing lowers the translation of control effects in the qualifier order. This makes sense, because if either of two sequenced expressions may duplicate or discard their continuations, then the compound expression may do the same.

PARAMETER PROPERTY 4 (Bottom and lifting)**.**

1. $c_1 \mathbin{⊙} c_2 = \bot_{\mathcal{C}}$ *if and only if* $c_1 = c_2 = \bot_{\mathcal{C}}$.

   RATIONALE. Sequencing impure expressions should not result in a pure expression.

2. *If* $\mathrm{D} \vdash_{\mathcal{C}} c_1 \mathbin{⊙} c_2 : \mathsf{CTL}$ *and* $c_1 \mathbin{⊙} c_2 \neq \bot_{\mathcal{C}}$*, then there exist some* $c'_1 \neq \bot_{\mathcal{C}}$ *and* $c'_2 \neq \bot_{\mathcal{C}}$ *such that*

   - $\mathrm{D} \vdash_{\mathcal{C}} c_1 \preceq c'_1$,
   - $\mathrm{D} \vdash_{\mathcal{C}} c_2 \preceq c'_2$,
   - $c'_1 \mathbin{⊙} c'_2 = c_1 \mathbin{⊙} c_2$*, and*
   - $\mathrm{D} \vdash_{\mathcal{C}} c'_1 \mathbin{⊙} c'_2 : \mathsf{CTL}$.

   RATIONALE. This assumption is likely not necessary, but it significantly simplifies the proof by allowing the effects in a sequence to be considered either all pure or all impure.

The final property concerns four lemmas that we state and prove for the generic system in the next subsection. An actual control effect instance needs to extend these lemmas to cover any additional rules added to the relevant judgments:

PARAMETER PROPERTY 5 (New rules)**.**

1. *Lemma 8.5 (§8.4.4) must be extended, by induction on derivations, for any rules added to the kinding judgment* $\mathrm{D} \vdash_{\mathcal{C}} i : k$.

2. *Lemma 8.6 (§8.4.4) must be extended, by induction on derivations, for any rules added to the control effect bounding judgment* $\mathrm{D} \vdash_{\mathcal{C}} c \succeq \xi$.

3. *Lemma 8.7 (§8.4.4) must be extended, by induction on derivations, for any rules added to the control effect subsumption judgment* $\mathrm{D} \vdash_{\mathcal{C}} c_1 \preceq c_2$.

4. *Lemma 8.8 (§8.4.4) must be extended, by induction on derivations, for any rules added to the expression typing judgment* $\mathrm{D}; \mathrm{G} \vdash_{\mathcal{C}} e : t \,; c$.

In §8.5, I give several example control effects and show that they satisfy the above properties.

8.4.4 Generic Type Soundness

Assuming that the above properties hold of the control effect parameter, it is now possible to prove a type soundness theorem for $\lambda^{\mathsf{URAL}}(\mathcal{C})$ that leaves the control effect abstract. I sketch the proof here and provide the full details in appendix C.

This first lemma ensures that control effects translate to well-formed qualifiers.

LEMMA 8.5 (Translation of kinding)**.**

For all D*, i, and k, if* $\mathrm{D} \vdash_{\mathcal{C}} i : k$ *then* $\mathrm{D}^* \vdash i^* : k^*$.

Proof. Please see p. 396. ▷

The next two lemmas concern how the translation of control effects to qualifiers relates to qualifier subsumption. The former ensures that the control effect bound used by typing rules such as C-T-APP matches the qualifier assigned to the type of a continuation by the CCoS translation. The latter shows that a larger control effect, which indicates more liberal treatment of a continuation, maps to a smaller qualifier, which indicates more liberal treatment of any value.

LEMMA 8.6 (Translation of effect bounds)**.**

If $\mathrm{D} \vdash_{\mathcal{C}} c \succeq \xi$ *then* $\mathrm{D}^* \vdash \xi \preceq c^*$.

Proof. Please see p. 399. ▷

LEMMA 8.7 (Translation of effect subsumption)**.**

If $\mathrm{D} \vdash_{\mathcal{C}} c_1 \preceq c_2$ *then* $\mathrm{D}^* \vdash {c_2}^* \preceq {c_1}^*$.

Proof. Please see p. 400. ▷

The most difficult lemma, and the heart of the proof, is about typing translated expressions. Given a $\lambda^{\mathsf{URAL}}(\mathcal{C})$ expression whose control effect is c, the translation of the control effect, c^*, is the qualifier of the continuation of the translated expression:

LEMMA 8.8 (Translation of term typing)**.**

If $\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} e : t \,;c$ *then*

$$\mathrm{D}^*;\mathrm{G}^* \vdash [\![e]\!]_{\mathcal{C}} : {}^{\mathsf{L}}({}^{c^*}(t^* \multimap \langle\!\langle t^*, c \rangle\!\rangle^{-}_{\mathcal{C}}) \multimap \langle\!\langle t^*, c \rangle\!\rangle^{+}_{\mathcal{C}}).$$

Proof. By induction on the typing derivation, generalizing the induction hypothesis thus:

> If $\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} e : t \,;c$, then for all τ_0 such that $\mathrm{D}^* \vdash \tau_0 : \star$, and for all ξ_0 such that $\mathrm{D}^* \vdash \xi_0 \preceq c^*$, we have $\mathrm{D}^*;\mathrm{G}^* \vdash [\![e]\!]_{\mathcal{C}} : {}^{\mathsf{L}}({}^{\xi_0}(t^* \multimap \langle\!\langle \tau_0, c \rangle\!\rangle^{-}_{\mathcal{C}}) \multimap \langle\!\langle \tau_0, c \rangle\!\rangle^{+}_{\mathcal{C}})$.

Let us consider two cases here:

$$\textbf{Case}\ \frac{\mathrm{D};\mathrm{G} \vdash_{\mathbb{C}} e : t ; c' \qquad \mathrm{D} \vdash_{\mathbb{C}} c' \preceq c}{\mathrm{D};\mathrm{G} \vdash_{\mathbb{C}} e : t ; c}.$$

By property 8.4 (part 3), $\mathrm{D}^* \vdash c^* \preceq c'^*$, and thus by property 8.4 (part 4), there exists some type τ_0' such that $\langle\!\langle \tau_0', c' \rangle\!\rangle_{\mathbb{C}}^- = \langle\!\langle \tau_0, c \rangle\!\rangle_{\mathbb{C}}^-$ and $\langle\!\langle \tau_0', c' \rangle\!\rangle_{\mathbb{C}}^+ = \langle\!\langle \tau_0, c \rangle\!\rangle_{\mathbb{C}}^+$. By the lemma assumption, $\mathrm{D}^* \vdash \xi_0 \preceq c^*$, and by transitivity of qualifier subsumption, $\mathrm{D}^* \vdash \xi_0 \preceq c'^*$. Thus, we can apply the induction hypothesis at $\mathrm{D};\mathrm{G} \vdash_{\mathbb{C}} e : t ; c'$, using the same ξ_0 but with τ_0' for τ_0, yielding

$$\mathrm{D}^*;\mathrm{G}^* \vdash [\![e]\!]_{\mathbb{C}} : \mathsf{L}^{(\xi_0}(t^* \multimap \langle\!\langle \tau_0', c' \rangle\!\rangle_{\mathbb{C}}^-) \multimap \langle\!\langle \tau_0', c' \rangle\!\rangle_{\mathbb{C}}^+).$$

Then it suffices to replace $\langle\!\langle \tau_0', c' \rangle\!\rangle_{\mathbb{C}}^-$ by $\langle\!\langle \tau_0, c \rangle\!\rangle_{\mathbb{C}}^-$ and $\langle\!\langle \tau_0', c' \rangle\!\rangle_{\mathbb{C}}^+$ by $\langle\!\langle \tau_0, c \rangle\!\rangle_{\mathbb{C}}^+$, which we know to be equal by property 8.4 (part 4).

$$\textbf{Case}\ \frac{\begin{array}{c}\mathrm{D} \vdash_{\mathbb{C}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_2 \qquad \mathrm{D} \vdash_{\mathbb{C}} \mathrm{G}_2 \preceq \xi_2 \\ \mathrm{D};\mathrm{G}_1 \vdash_{\mathbb{C}} e_1 : {}^{\xi_1}(t_1 \xrightarrow{c} t_2) ; c_1 \qquad \mathrm{D} \vdash_{\mathbb{C}} c_1 \succeq \xi_2 \\ \mathrm{D};\mathrm{G}_2 \vdash_{\mathbb{C}} e_2 : t_1 ; c_2 \qquad \mathrm{D} \vdash_{\mathbb{C}} c_2 \succeq \xi_1 \\ \mathrm{D} \vdash_{\mathbb{C}} c_1 \oslash c_2 \oslash c : \mathsf{CTL}\end{array}}{\mathrm{D};\mathrm{G} \vdash_{\mathbb{C}} e_1\, e_2 : t_2 ; c_1 \oslash c_2 \oslash c}.$$

For rule C-T-APP, we want to show that $[\![e_1\, e_2]\!]_{\mathbb{C}}$ has type

$$\mathsf{L}^{(\xi_0}(t_2{}^* \multimap \langle\!\langle \tau_0, c_1 \oslash c_2 \oslash c \rangle\!\rangle_{\mathbb{C}}^-) \multimap \langle\!\langle \tau_0, c_1 \oslash c_2 \oslash c \rangle\!\rangle_{\mathbb{C}}^+).$$

Consider the translation of $e_1\, e_2$,

$$\lambda y.\, [\![e_1]\!]_{\mathbb{C}} (\lambda x_1.\, [\![e_2]\!]_{\mathbb{C}} (\lambda x_2.\, x_1 \_ x_2 (\lambda x.\, y\, x))).$$

The type derivation is very large (see p. 410), but it hinges on giving the right qualifiers to the types of continuations. We will consider the continuation passed to the whole expression and the continuations constructed for e_1, e_2, and the function application itself, in turn.

First we consider y, the continuation of the whole application expression. Given the type that we need to derive for the whole expression, the qualifier of y's type must be ξ_0. Furthermore, from the assumptions of the lemma, we know that $\mathrm{D}^* \vdash \xi_0 \preceq (c_1 \oslash c_2 \oslash c)^*$. By property 3, each of

${c_1}^*$, ${c_2}^*$, and c^* is greater than $(c_1 \oslash c_2 \oslash c)^*$, so by transitivity, ξ_0 is less than each of these.

Expression e_1 has effect c_1, so by the induction hypothesis, its continuation may have qualifier ${c_1}^*$. The continuation passed to $[\![e_1]\!]_{\mathcal{C}}$ is

$$\lambda x_1.\, [\![e_2]\!]_{\mathcal{C}}\, (\lambda x_2.\, x_1 \_ x_2\, (\lambda x.\, y\, x)),$$

whose free variables are $\{y\} \cup \mathrm{fv}(e_2)$. Thus, the qualifier of this function must upper bound both ξ_0 and the qualifiers of the types in G_2 (the type context for e_2). We have $\mathrm{D}^* \vdash \xi_0 \preceq {c_1}^*$ from the previous paragraph. Furthermore, looking at the premises of rule T-App, we see that ξ_2 upper bounds the types in G_2 and is less than ${c_1}^*$ (by property 8.4 (part 2)), so by transitivity, $\mathrm{D}^* \vdash {\mathrm{G}_2}^* \preceq {c_1}^*$, as desired.

Expression e_2 has effect c_2, so similarly, its continuation should have qualifier ${c_2}^*$. The free variables of e_2's continuation are only y and x_1, which is the value of e_1. We handle y as before. The type of x_1 is ${}^{\xi_1}((t_1 \xrightarrow{c}\!\!\circ\, t_2)^*)$, so it remains to show that $\mathrm{D}^* \vdash \xi_1 \preceq {c_2}^*$, by property 8.4 (part 2) applied to the premise $\mathrm{D} \vDash_{\mathcal{C}} c_2 \succeq \xi_1$.

Finally, given that x_1 has type ${}^{\xi_1}((t_1 \xrightarrow{c}\!\!\circ\, t_2)^*)$, it expects a continuation whose qualifier is c^*. The type of y has qualifier ξ_0, which is less than c^*. Then by lemma 8.3, the type of the η expansion $\lambda x.\, y\, x$ may be given qualifier c^*.

Please see p. 400 for the remaining cases. ▷

Corollary 8.9 (Translation of program typing)**.**

If $\mathrm{D}; \mathrm{G} \vDash_{\mathcal{C}} e : t ; \bot_{\mathcal{C}}$ *where* $\mathrm{D} \vDash_{\mathcal{C}} t \preceq \mathsf{A}$*, then*

$$\mathrm{D}^*; \mathrm{G}^* \vdash [\![e]\!]_{\mathcal{C}}\, \mathsf{done}_{\mathcal{C}} : \langle\!\langle t^* \rangle\!\rangle_{\mathcal{C}}.$$

Proof. By lemma 8.5, lemma 8.8, property 2, and rules QSub-Refl and T-App. Please see p. 423 for details. ▷

LEMMA 8.10 (λ^{URAL} soundness)**.**

If $\bullet;\bullet \vdash e_1 : \tau$ *and* $e_1 \overset{*}{\longmapsto} e_2$*, then either* $\exists v_2.e_2 \equiv v_2$ *or* $\exists e_3.e_2 \longmapsto e_3$.

Proof. See the proof in Ahmed et al. (2005). □

THEOREM 8.11 ($\lambda^{\mathsf{URAL}}(\mathcal{C})$ soundness)**.**

If $\bullet;\bullet \vdash_{\mathcal{C}} e : t ; \bot_{\mathcal{C}}$*, and* $\bullet \vdash_{\mathcal{C}} t \preceq \mathsf{A}$ *then* $eval(e) \neq$ WRONG.

Proof. By corollary 8.9, $\bullet;\bullet \vdash [\![e]\!]_{\mathcal{C}}\ \mathsf{done}_{\mathcal{C}} : \langle\!\langle t^* \rangle\!\rangle_{\mathcal{C}}$. Then by lemma 8.10, either $[\![e]\!]_{\mathcal{C}}\ \mathsf{done}_{\mathcal{C}}$ reduces to a value v, in which case $eval(e) = v$, or $[\![e]\!]_{\mathcal{C}}\ \mathsf{done}_{\mathcal{C}}$ diverges, in which case $eval(e)$ is undefined. □

8.5 Example Control Effects

In the previous section, I proved type soundness for generic $\lambda^{\mathsf{URAL}}(\mathcal{C})$, a substructural λ calculus parametrized by abstract control effects. In this section, I give three instances of control effects as described by definition 8.1 and show that they satisfy the properties on which the generic theorem depends.

It will be useful, when stating several later definitions, to have a definition for meets and joins of qualifiers. Because qualifiers include qualifier variables, the lattice on constant qualifiers is insufficient. Because λ^{URAL} lacks qualifier expressions like Alms has, meets and joins on qualifier variables are undefined in several cases.

DEFINITION 8.12 (Qualifier meets and joins)**.**

Define meets and joins of qualifiers as follows:

$$\mathsf{L} \sqcap \xi = \xi \sqcap \mathsf{L} = \xi \sqcap \xi = \xi \qquad \mathsf{U} \sqcup \xi = \xi \sqcup \mathsf{U} = \xi \sqcup \xi = \xi$$

$$\mathsf{U} \sqcap \xi = \xi \sqcap \mathsf{U} = \mathsf{A} \sqcap \mathsf{R} = \mathsf{R} \sqcap \mathsf{A} = \mathsf{U} \qquad \mathsf{L} \sqcup \xi = \xi \sqcup \mathsf{L} = \mathsf{A} \sqcup \mathsf{R} = \mathsf{R} \sqcup \mathsf{A} = \mathsf{L}$$

otherwise, $\xi \sqcap \xi'$ *is undefined* *otherwise,* $\xi \sqcup \xi'$ *is undefined*

$\boxed{\mathrm{D} \vdash_{\mathcal{D}} i : k}$ *(kinding delimited control effects)*

$$\textsc{D-K-Qual}\ \frac{\mathrm{D} \vdash_{\mathcal{D}} \xi : \mathsf{QUAL}}{\mathrm{D} \vdash_{\mathcal{D}} \overline{\xi} : \mathsf{CTL}} \qquad \textsc{D-K-Join}\ \frac{\mathrm{D} \vdash_{\mathcal{D}} d_1 : \mathsf{CTL} \qquad \mathrm{D} \vdash_{\mathcal{D}} d_2 : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{D}} d_1 \sqcup d_2 : \mathsf{CTL}}$$

$\boxed{\mathrm{D} \vdash_{\mathcal{D}} d \succeq \xi}$ *(qualifier bound for delimited control effects)*

$$\textsc{D-B-Qual}\ \frac{\mathrm{D} \vdash_{\mathcal{D}} \xi \preceq \xi'}{\mathrm{D} \vdash_{\mathcal{D}} \overline{\xi'} \succeq \xi} \qquad \textsc{D-B-Join}\ \frac{\mathrm{D} \vdash_{\mathcal{D}} d_1 \succeq \xi \qquad \mathrm{D} \vdash_{\mathcal{D}} d_2 \succeq \xi}{\mathrm{D} \vdash_{\mathcal{D}} d_1 \sqcup d_2 \succeq \xi}$$

$\boxed{\mathrm{D} \vdash_{\mathcal{D}} d_1 \preceq d_2}$ *(delimited control effect subsumption)*

$$\textsc{DSub-Bot}\ \frac{\mathrm{D} \vdash_{\mathcal{D}} d : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{D}} \bot_{\mathcal{D}} \preceq d} \qquad \textsc{DSub-Lin}\ \frac{\mathrm{D} \vdash_{\mathcal{D}} \xi : \mathsf{QUAL}}{\mathrm{D} \vdash_{\mathcal{D}} \overline{\mathsf{L}} \preceq \overline{\xi}} \qquad \textsc{DSub-Top}\ \frac{\mathrm{D} \vdash_{\mathcal{D}} d : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{D}} d \preceq \overline{\mathsf{U}}}$$

$$\textsc{DSub-Join}\ \frac{\mathrm{D} \vdash_{\mathcal{D}} d_1 \preceq d_1' \qquad \mathrm{D} \vdash_{\mathcal{D}} d_2 \preceq d_2' \qquad \mathrm{D} \vdash_{\mathcal{D}} d_1 \sqcup d_2 : \mathsf{CTL} \qquad \mathrm{D} \vdash_{\mathcal{D}} d_1' \sqcup d_2' : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{D}} d_1 \sqcup d_2 \preceq d_1' \sqcup d_2'}$$

$\boxed{\mathrm{D}; \mathrm{G} \vdash_{\mathcal{D}} e : t ; d}$ *(delimited control expression typing)*

$$\textsc{D-T-Reset}\ \frac{\mathrm{D}; \mathrm{G} \vdash_{\mathcal{D}} e : {}^{\mathsf{U}}1 ; d}{\mathrm{D}; \mathrm{G} \vdash_{\mathcal{D}} \mathsf{reset}\, e : {}^{\mathsf{U}}1 ; \bot_{\mathcal{D}}} \qquad \textsc{D-T-Shift}\ \frac{\mathrm{D}; \mathrm{G}, x :^{\xi} (t \xrightarrow{\bot_{\mathcal{D}}} {}^{\mathsf{U}}1) \vdash_{\mathcal{D}} e : {}^{\mathsf{U}}1 ; d}{\mathrm{D}; \mathrm{G} \vdash_{\mathcal{D}} \mathsf{shift}\, x \,\mathsf{in}\, e : t ; d \sqcup \overline{\xi}}$$

Figure 8.17: Statics for delimited continuation effects

8.5.1 Shift and Reset

This section defines a control effect instance for delimited continuations. In this example, answer types are restricted to the unit type ${}^{\mathsf{U}}1$, in order to keep the effects simple. In §8.5.2, I show how to define a more general control effect instance that allows answer-type modification.

We add shift and reset to $\lambda^{\mathsf{URAL}}(\mathcal{C})$ as follows. First, extend the syntax:

e	::=		new expressions
	|	$\cdots$	*extending syntax from figure 8.1*
	|	$\mathsf{reset}\, e$	delimiter
	|	$\mathsf{shift}\, x \,\mathsf{in}\, e$	control operator

We give the dynamics of the new expressions by defining their CCoS translations, which are standard:

$$[\![\mathsf{reset}\, e]\!]_{\mathcal{D}} = \lambda y.\, y([\![e]\!]_{\mathcal{D}}\, (\lambda x.\, x))$$
$$[\![\mathsf{shift}\, x \,\mathsf{in}\, e]\!]_{\mathcal{D}} = \lambda y.(\lambda x.\, [\![e]\!]_{\mathcal{D}}\, (\lambda x'.\, x'))(\Lambda.\, \lambda x.\, \lambda y'.\, y'(y\, x))$$

To type shift and reset, we define delimited continuation effects d as the dual lattice of qualifiers ξ with a new point $\bot_{\mathcal{D}}$:

d	::=		delimited continuation effects
	|	$\bot_{\mathcal{D}}$	no effect
	|	α	an effect variable
	|	$\overline{\xi}$	treats continuation like ξ
	|	$d_1 \sqcup d_2$	effect join

Let $\mathcal{D}$ be the set of delimited continuation effects (d) quotiented by the following equivalences:

$$\overline{\xi_1} \sqcup \overline{\xi_2} = \overline{(\xi_1 \sqcap \xi_2)} \text{ when } \xi_1 \sqcap \xi_2 \text{ is defined}$$
$$d \sqcup \bot_{\mathcal{D}} = \bot_{\mathcal{D}} \sqcup d = d \sqcup d = d.$$

(The quotient simplifies defining other functions and relations on delimited continuation effects.) Then delimited continuation effects are the triple $(\mathcal{D}, \bot_{\mathcal{D}}, \sqcup)$.

We extend the type system of $\lambda^{\mathsf{URAL}}(\mathcal{C})$ with the new rules in figure 8.17. The new kinding rules say that qualifiers-as-effects ($\overline{\xi}$) and joins ($d_1 \sqcup d_2$) are well-kinded if their components are. The new control effect bound rules say that a control effect $\overline{\xi'}$ is bounded by all qualifiers ξ that are less than ξ' and that any bound of both effects in a join bounds the join as well. The rules added for effect subsumption effectively axiomatize the delimited continuation effect lattice. Finally, we add two rules for typing shift and reset. To type an expression reset e, subexpression e may have any effect whatsoever, but must return type ${}^{\mathsf{U}}1$. (We lift this restriction in §8.5.2.) Then reset e is pure and also has type ${}^{\mathsf{U}}1$. To type shift x in e, we give x type ${}^{\xi}(t \xrightarrow{\perp_{\mathcal{D}}}\circ\, {}^{\mathsf{U}}1)$ for checking e, where $\overline{\xi}$ is joined with the effect of e to get the effect of the whole shift expression. That is, because shift captures its continuation and gives the reified continuation qualifier ξ, its effect must be at least $\overline{\xi}$, since that qualifier determines how it might treat its captured continuation.

Type soundness. To prove type soundness for $\lambda^{\mathsf{URAL}}(\mathcal{C})$ extended with delimited continuation effects, we need to give the translation parameter as described by definition 8.2. We define the translation parameter as follows:

$$\langle\!\langle \tau, d \rangle\!\rangle^{-}_{\mathcal{D}} = \langle\!\langle \tau, d \rangle\!\rangle^{+}_{\mathcal{D}} = {}^{\mathsf{U}}1$$

$$\mathsf{done}_{\mathcal{D}} = \lambda x.\langle\rangle$$

$$d^* = \begin{cases} \mathsf{L} & \text{if } d = \perp_{\mathcal{D}} \\ \alpha & \text{if } d = \alpha \\ \xi & \text{if } d = \overline{\xi} \\ \mathsf{U} & \text{otherwise} \end{cases}$$

Then, we must show that this definition satisfies the properties of §8.4.3:

THEOREM 8.13 (Delimited continuation properties)**.**

Delimited continuation effects $(\mathcal{D}, \perp_{\mathcal{D}}, \sqcup)$ *satisfy properties 1–5.*

Proof.

Property 1 (Answer types). We need to show that several equalities on answer types, such as that $\langle\!\langle \tau, d_1 \rangle\!\rangle^-_{\mathcal{D}} = \langle\!\langle \tau, d_2 \rangle\!\rangle^+_{\mathcal{D}}$, hold whenever $d_1 \sqcup d_2$ is well formed. All of the equalities are trivial because $\langle\!\langle \tau, d \rangle\!\rangle^-_{\mathcal{D}} = \langle\!\langle \tau, d \rangle\!\rangle^+_{\mathcal{D}} = {}^{\mathsf{U}}1$.

Property 2 (Done). We need to show that $\Delta; \bullet \vdash \mathsf{done}_{\mathcal{D}} : {}^{\mathsf{L}}(\tau \multimap \langle\!\langle \tau \rangle\!\rangle_{\mathcal{D}})$. Given the definition of $\mathsf{done}_{\mathcal{D}}$, we can show $\Delta; \bullet \vdash \lambda x.\langle\rangle : {}^{\mathsf{L}}(\tau \multimap \langle\!\langle \tau \rangle\!\rangle_{\mathcal{D}})$ by a straightforward type derivation.

Property 3 (Effect sequencing). We need to show that $\mathrm{D} \vdash_{\mathcal{D}} d_1 \sqcup d_2 : \mathsf{CTL}$ implies that $\mathrm{D}^* \vdash (d_1 \sqcup d_2)^* \preceq {d_1}^*$ and $\mathrm{D}^* \vdash (d_1 \sqcup d_2)^* \preceq {d_2}^*$. By symmetry, it suffices to show the former:

(1) $\mathrm{D} \vdash_{\mathcal{D}} d_1 \preceq d_1$	by CSUB-REFL
(2) $\mathrm{D} \vdash_{\mathcal{D}} \bot_{\mathcal{D}} \preceq d_2$	by DSUB-BOT
(3) $\mathrm{D} \vdash_{\mathcal{D}} d_1 \sqcup \bot_{\mathcal{D}} \preceq d_1 \sqcup d_2$	by (1–2), DSUB-JOIN
(4) $\mathrm{D} \vdash_{\mathcal{D}} d_1 \preceq d_1 \sqcup d_2$	by (3), $d_1 \sqcup \bot_{\mathcal{D}} = d_1$
(5) $\mathrm{D}^* \vdash (d_1 \sqcup d_2)^* \preceq {d_1}^*$	by (4), lemma 8.7.

Property 4 (Bottom and lifting).

1. To show that $d_1 \sqcup d_2 = \bot_{\mathcal{D}}$ if and only if $d_1 = d_2 = \bot_{\mathcal{D}}$, we consider the quotienting of $\mathcal{D}$.

2. We must also show that if $\mathrm{D} \vdash_{\mathcal{D}} d_1 \sqcup d_2 : \mathsf{CTL}$ and $d_1 \sqcup d_2 \neq \bot_{\mathcal{D}}$, then there exist some $d_1' \neq \bot_{\mathcal{D}}$ and $d_2' \neq \bot_{\mathcal{D}}$ with particular properties. For each ${d_i}^{(i \in \{1,2\})}$, if $d_i = \bot_{\mathcal{D}}$ then let $d_i' = \overline{\mathsf{L}}$; otherwise, let $d_i' = d_i$. This ensures that 1–2) each $\mathrm{D} \vdash_{\mathcal{D}} d_i \preceq d_i'$, 3) $d_1 \sqcup d_2 = d_1' \sqcup d_2'$, and 4) $d_1' \sqcup d_2'$ is well formed.

Property 5 (New rules).

1. We show that $\mathrm{D} \vdash_{\mathcal{D}} d \succeq \xi$ implies that $\mathrm{D}^* \vdash \xi \preceq d^*$, by induction on the derivation. The only new cases to consider are for rules D-B-QUAL and D-B-JOIN. These require a lemma about the translation of qualifier subsumption derivations.

2. We show that $\mathrm{D} \vdash_{\mathcal{D}} d_1 \preceq d_2$ implies that $\mathrm{D}^* \vdash {d_2}^* \preceq {d_1}^*$, again by induction on the derivation. The only nontrivial case is when

$$\frac{\mathrm{D} \vdash_{\mathcal{D}} d_1 \preceq d_1' \qquad \mathrm{D} \vdash_{\mathcal{D}} d_2 \preceq d_2' \qquad \mathrm{D} \vdash_{\mathcal{D}} d_1 \sqcup d_2 : \mathsf{CTL} \qquad \mathrm{D} \vdash_{\mathcal{D}} d_1' \sqcup d_2' : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{D}} d_1 \sqcup d_2 \preceq d_1' \sqcup d_2'}.$$

   We show that $\mathrm{D}^* \vdash (d_1' \sqcup d_2')^* \preceq (d_1 \sqcup d_2)^*$ by exhaustively enumerating the possibilities for d_1, d_2, d_1', and d_2' such that the premises hold.

3. For translation of kinding, we show that $\mathrm{D} \vdash_{\mathcal{C}} d : \mathsf{CTL}$ implies that $\mathrm{D}^* \vdash d^* : \mathsf{QUAL}$. We proceed, as usual, by a simple induction on the derivation, considering the two new kinding rules for delimited continuation effects.

4. For translation of typing, we use the generalized induction hypothesis as in the proof of lemma 8.8. There are two cases, for shift and reset, each of which requires a large type derivation.

Please see p. 425 for additional details. ▷

8.5.2 Shift and Reset with Answer-Type Modification

The type-and-effect system for shift and reset described in §8.5.1 requires that all *answer types*—the type of all reset expressions—be ${}^{\cup}1$. Our second example adds answer-type modification (*à la* Danvy and Filinski 1989), which allows shift to capture and compose continuations of differing types and allows the answer delivered by reset to have any type. Both the syntax and CCoS translation are as in §8.5.1, but we change the definition of control effects as follows. An answer-type control effect a is either the pure effect $\bot_{\mathcal{A}}$ or a collection of qualifiers $\xi_1, \ldots, \xi_j$ along with old and new answer types t_1 and t_2:

a	::=		answer-type modification effects
	|	$\bot_{\mathcal{A}}$	pure
	|	${}^{\Xi}(t_1 \rightarrowtail t_2)$	captures continuation
Ξ	::=	$\xi_1, \ldots, \xi_j$	qualifier collections

$\boxed{\mathrm{D} \vdash_{\mathcal{A}} i : k}$ *(kinding answer-type effects)*

$$\textsc{A-K-Effect}\ \frac{\mathrm{D} \vdash_{\mathcal{A}} \xi_1 : \mathsf{QUAL} \quad \cdots \quad \mathrm{D} \vdash_{\mathcal{A}} \xi_k : \mathsf{QUAL} \qquad \mathrm{D} \vdash_{\mathcal{A}} t_1 : \star \qquad \mathrm{D} \vdash_{\mathcal{A}} t_2 : \star}{\mathrm{D} \vdash_{\mathcal{A}} {}^{\xi_1,\ldots,\xi_k}(t_1 \rightarrowtail t_2) : \mathsf{CTL}}$$

$\boxed{\mathrm{D} \vdash_{\mathcal{A}} a \succeq \xi}$ *(qualifier bound for answer-type effects)*

$$\textsc{A-B-Qual}\ \frac{\mathrm{D} \vdash_{\mathcal{A}} \xi \preceq \xi_1 \quad \cdots \quad \mathrm{D} \vdash_{\mathcal{A}} \xi \preceq \xi_j \qquad \mathrm{D} \vdash_{\mathcal{A}} t_1 : \star \qquad \mathrm{D} \vdash_{\mathcal{A}} t_2 : \star}{\mathrm{D} \vdash_{\mathcal{A}} {}^{\xi_1,\ldots,\xi_j}(t_1 \rightarrowtail t_2) \succeq \xi}$$

$\boxed{\mathrm{D} \vdash_{\mathcal{A}} a_1 \preceq a_2}$ *(answer-type effect subsumption)*

$$\textsc{ASub-Bot}\ \frac{\mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi}(t \rightarrowtail t) : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{A}} \bot_{\mathcal{A}} \preceq {}^{\Xi}(t \rightarrowtail t)} \qquad \textsc{ASub-L}\ \frac{\mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi}(t_1 \rightarrowtail t_2) : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{A}} {}^{\mathsf{L}}(t_1 \rightarrowtail t_2) \preceq {}^{\Xi}(t_1 \rightarrowtail t_2)}$$

$$\textsc{ASub-Top}\ \frac{\mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi}(t_1 \rightarrowtail t_2) : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi}(t_1 \rightarrowtail t_2) \preceq {}^{\mathsf{U}}(t_1 \rightarrowtail t_2)}$$

$$\textsc{ASub-Join}\ \frac{\mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi_1}(t_1 \rightarrowtail t_2) \preceq {}^{\Xi'_1}(t_1 \rightarrowtail t_2) \qquad \mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi_2}(t_1 \rightarrowtail t_2) \preceq {}^{\Xi'_2}(t_1 \rightarrowtail t_2)}{\mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi_1,\Xi_2}(t_1 \rightarrowtail t_2) \preceq {}^{\Xi'_1,\Xi'_2}(t_1 \rightarrowtail t_2)}$$

$\boxed{\mathrm{D};\mathrm{G} \vdash_{\mathcal{A}} e : t \,;\, a}$ *(answer-type effect expression typing)*

$$\textsc{A-T-Reset}\ \frac{\mathrm{D};\mathrm{G} \vdash_{\mathcal{A}} e : t_0 \,;\, {}^{\Xi}(t_0 \rightarrowtail t)}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{A}} \mathsf{reset}\, e : t \,;\, \bot_{\mathcal{A}}} \qquad \textsc{A-T-Shift}\ \frac{\mathrm{D};\mathrm{G}, x{:}^{\xi}(t_1 \xrightarrow{\bot_{\mathcal{A}}\circ} t_2) \vdash_{\mathcal{A}} e : t_0 \,;\, {}^{\Xi}(t_0 \rightarrowtail t)}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{A}} \mathsf{shift}\, x \,\mathsf{in}\, e : t_1 \,;\, {}^{\Xi,\xi}(t_2 \rightarrowtail t)}$$

Figure 8.18: Statics for answer-type effects

A type derivation $\mathrm{D}; \mathrm{G} \vdash_{\mathcal{A}} e : t ; {}^{\xi_1,\ldots,\xi_j}(t_1 \rightarrowtail t_2)$ may be understood as follows:

- The collection of qualifiers $\xi_1, \ldots, \xi_j$ keeps track of all the ways that expression e may treat its context; expression e may be considered to treat its context according to any qualifier ξ that lower bounds all of $\xi_1, \ldots, \xi_j$. We need a collection of qualifiers because qualifiers do not, in the presence of qualifier variables, have greatest lower bounds.

- Evaluated in a context expecting type t whose original answer type was t_1, expression e changes the answer type to t_2. This means that our type-and-effect judgment, disregarding substructural considerations, is equivalent to the type judgment that Danvy and Filinski write as $\Gamma, t_1 \vdash e : t, t_2$.

For answer-type modification effects, we define the partial sequencing operation as follows:

$$\bot_{\mathcal{A}} \circ a = a$$
$$a \circ \bot_{\mathcal{A}} = a$$
$${}^{\Xi}(t' \rightarrowtail t_2) \circ {}^{\Xi'}(t_1 \rightarrowtail t') = {}^{\Xi,\Xi'}(t_1 \rightarrowtail t_2).$$

Any other cases are undefined. Collections of qualifiers are quotiented by the following equivalence:

$$\xi_1, \xi_2 = \xi_1 \sqcap \xi_2 \text{ when } \xi_1 \sqcap \xi_2 \text{ is defined.}$$

Then we define answer-type modification effects as the triple $(\mathcal{A}, \bot_{\mathcal{A}}, \circ)$.

The new type rules for answer-type effects appear in figure 8.18. For the most part, these rules treat the collection of qualifiers $\xi_1, \ldots, \xi_j$ similarly to the delimited continuation effect $\overline{\xi_1} \sqcup \cdots \sqcup \overline{\xi_j}$ from §8.5.1. However, there is some subtlety to the definition of answer-type effect subsumption: the only non-bottom effects related by subsumption are those whose before and after answer types match, pairwise, but the pure effect $\bot_{\mathcal{A}}$ is less than any effect whose before and after answer types match *each other* (rule ASUB-BOT). This makes sense, as pure expressions do not change the answer type.

The rules for typing shift and reset expressions are a hybrid of the rules from §8.5.1, which they follow for the qualifier portion, and the rules from Danvy and Filinski (1989), which they follow for maintaining answer types.

Type soundness. To prove type soundness for $\lambda^{\mathsf{URAL}}(\mathcal{C})$ with answer-type modification, we define the translation parameter as follows:

$$\begin{aligned}
\langle\!\langle \tau, \bot_{\mathcal{A}} \rangle\!\rangle^{-}_{\mathcal{A}} &= \tau \\
\langle\!\langle \tau, {}^{\Xi}(t_1 \rightarrowtail t_2) \rangle\!\rangle^{-}_{\mathcal{A}} &= {t_1}^* \\
\langle\!\langle \tau, \bot_{\mathcal{A}} \rangle\!\rangle^{+}_{\mathcal{A}} &= \tau \\
\langle\!\langle \tau, {}^{\Xi}(t_1 \rightarrowtail t_2) \rangle\!\rangle^{+}_{\mathcal{A}} &= {t_2}^*
\end{aligned}$$

$$\mathsf{done}_{\mathcal{A}} = \lambda x.\, x$$

$$a^* = \begin{cases} \mathsf{L} & \text{if } a = \bot_{\mathcal{A}} \\ \xi & \text{if } a = {}^{\xi}(t_1 \rightarrowtail t_2) \\ \mathsf{U} & \text{otherwise} \end{cases}$$

THEOREM 8.14 (Answer-type effect properties)**.**

Answer-type modification effects $(\mathcal{A}, \bot_{\mathcal{A}}, \circ)$ *satisfy properties 1–5.*

Proof. Please see p. 435. ▷

8.5.3 Exceptions

We add exceptions to $\lambda^{\mathsf{URAL}}(\mathcal{C})$ as follows. We assume a set *Exn* of exception names ψ and extend the syntax of expressions:

ψ	$\in$	*Exn*	exception names
e	::=		new expressions
	|	$\cdots$	*extending syntax from figure 8.1*
	|	e_1 handle $\psi \to e_2$	delimiter
	|	raise ψ	control operator

$\boxed{\mathrm{D} \vdash_{\mathfrak{X}} i : k}$ *(kinding exception effects)*

$$\textsc{X-K-Sing}\ \frac{}{\mathrm{D} \vdash_{\mathfrak{X}} \{\psi\} : \mathsf{CTL}} \qquad \textsc{X-K-Union}\ \frac{\mathrm{D} \vdash_{\mathfrak{X}} \Psi_1 : \mathsf{CTL} \qquad \mathrm{D} \vdash_{\mathfrak{X}} \Psi_2 : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathfrak{X}} \Psi_1 \cup \Psi_2 : \mathsf{CTL}}$$

$\boxed{\mathrm{D} \vdash_{\mathfrak{X}} \Psi \succeq \xi}$ *(qualifier bound for exception effects)*

$$\textsc{X-B-Raise}\ \frac{\mathrm{D} \vdash_{\mathfrak{X}} \Psi : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathfrak{X}} \Psi \succeq \mathsf{A}}$$

$\boxed{\mathrm{D}; \mathrm{G} \vdash_{\mathfrak{X}} e : t \,;\, \Psi}$ *(exception effect expression typing)*

$$\textsc{X-T-Raise}\ \frac{\mathrm{D} \vdash_{\mathfrak{X}} t : \star}{\mathrm{D}; \bullet \vdash_{\mathfrak{X}} \mathsf{raise}\ \psi : t \,;\, \{\psi\}} \qquad \textsc{X-T-Handle}\ \frac{\begin{array}{c}\mathrm{D} \vdash_{\mathfrak{X}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_2 \\ \mathrm{D}; \mathrm{G}_1 \vdash_{\mathfrak{X}} e_1 : t \,;\, \{\psi\} \cup \Psi \\ \mathrm{D}; \mathrm{G}_2 \vdash_{\mathfrak{X}} e_2 : t \,;\, \Psi \qquad \mathrm{D} \vdash_{\mathfrak{X}} \mathrm{G}_2 \preceq \mathsf{A}\end{array}}{\mathrm{D}; \mathrm{G} \vdash_{\mathfrak{X}} e_1\ \mathsf{handle}\ \psi \to e_2 : t \,;\, \Psi}$$

Figure 8.19: Statics for exception effects

While these exceptions are simple tags, it would not be difficult to have exceptions carry values. As in the previous example, we define the dynamics by the CCoS translation. However, because the CCoS translation for exceptions is type directed, we show how the type system is extended first.

To type exceptions, we instantiate $\lambda^{\mathsf{URAL}}(\mathcal{C})$ as follows. Exception effects, Ψ, are sets of primitive exception names ψ:

Ψ	::=		exception effect sets
	|	$\varnothing$	the empty effect
	|	α	an effect variable
	|	$\{\psi\}$	singleton effect
	|	$\Psi_1 \cup \Psi_2$	effect union

Let $\mathcal{X}$ be the set of exception effect sets (Ψ). Then we define exception effects as the triple $(\mathcal{X}, \varnothing, \cup)$. We consider exception effects as true sets, not merely as the free algebra generated by the syntax. Thus, the subsumption order is set containment:

$\boxed{\mathrm{D} \vdash_{\mathcal{X}} \Psi_1 \preceq \Psi_2}$ *(exception effect subsumption)*

$$\frac{\Psi_1 \subseteq \Psi_2 \qquad \mathrm{D} \vdash_{\mathcal{X}} \Psi_1 : \mathsf{CTL} \qquad \mathrm{D} \vdash_{\mathcal{X}} \Psi_2 : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{X}} \Psi_1 \preceq \Psi_2} \text{XSUB-SUBSET}$$

The other new type rules for exception effects appear in figure 8.19. Note that rule X-B-RAISE says that all exception effects are bounded below by A; this is because exceptions allow an expression to discard its context but not duplicate it. (Of course, the empty exception set $\varnothing$ is bounded by L by rule C-B-PURE.)

To define the CCoS translation, we assume a run-time representation of exceptions and exception sets as follows:

- There is an exception pretype exn such that $\Delta \vdash \mathsf{exn} : \overline{\star}$.
- Each exception ψ is represented by a λ^{URAL} value ψ^*, such that $\Delta; \bullet \vdash \psi^* : {}^{\mathsf{U}}\mathsf{exn}$.
- For each exception ψ and pair of λ^{URAL} values v_1 and v_2, there is a λ^{URAL} value $[v_1, v_2]_\psi$ such that

$$\frac{}{[v_1, v_2]_\psi \, \psi^* \longmapsto v_1 \, \psi^*} \qquad \frac{\psi \neq \psi'}{[v_1, v_2]_\psi \, \psi'^* \longmapsto v_2 \, \psi'^*}.$$

$$\frac{\Delta; \Gamma \vdash v_1 : {}^{\xi_1}({}^{\mathsf{U}}\mathsf{exn} \multimap \tau) \qquad \Delta \vdash \xi_1 \preceq \xi \qquad \Delta; \Gamma \vdash v_2 : {}^{\xi_2}({}^{\mathsf{U}}\mathsf{exn} \multimap \tau) \qquad \Delta \vdash \xi_2 \preceq \xi}{\Delta; \Gamma \vdash [v_1, v_2]_\psi : {}^{\xi}({}^{\mathsf{U}}\mathsf{exn} \multimap \tau)}$$

Intuitively, $[v_1, v_2]_\psi$ performs case analysis on exception values: when applied to exception ψ, it passes the exception to v_1, and when applied to any other exception, it passes the exception to v_2.

For exception effects, we use a typed CCoS translation that takes an extra parameter: the exception effect of the expression to be translated. We assume that the generic CCoS has been updated to translate type derivations as well in order to propagate control effects correctly. Then we can give the CCoS translation for exceptions:

$$[\![\mathsf{raise}\ \psi]\!]^{\Psi}_{\mathcal{X}} = \lambda\_.\ \mathsf{inl}\ \psi^*$$
$$[\![e_1\ \mathsf{handle}\ \psi \to e_2]\!]^{\Psi}_{\mathcal{X}} = \lambda y.[v, y]([\![e_1]\!]^{\{\psi\}\cup\Psi}_{\mathcal{X}}(\lambda x.\ \mathsf{inr}\ x))$$

$$\text{where } v = \begin{cases} \lambda\_.\,[\![e_2]\!]^{\varnothing}_{\mathcal{X}}\ y & \text{if } \Psi = \varnothing; \\ [\lambda\_.\,[\![e_2]\!]^{\Psi}_{\mathcal{X}}\ y, \lambda x.\ \mathsf{inl}\ x]_{\psi} & \text{if } \Psi \neq \varnothing. \end{cases}$$

Type soundness. To prove type soundness for $\lambda^{\mathsf{URAL}}(\mathcal{C})$ extended with exceptions, we define the translation parameter as follows:

$$\langle\!\langle \tau, \Psi \rangle\!\rangle^{-}_{\mathcal{X}} = \langle\!\langle \tau, \Psi \rangle\!\rangle^{+}_{\mathcal{X}} = {}^{\mathsf{L}}({}^{\mathsf{U}}\,\mathsf{exn} \oplus \tau)$$

$$\mathsf{done}_{\mathcal{X}} = \lambda x.\ \mathsf{inr}\ x$$

$$\Psi^* = \begin{cases} \mathsf{L} & \text{if } \Psi = \varnothing \\ \mathsf{A} & \text{if } \Psi \neq \varnothing \end{cases}$$

THEOREM 8.15 (Exception effect properties)**.**

Exception effects $(\mathcal{X}, \varnothing, \cup)$ *satisfy properties 1–5.*

Proof. Please see p. 441. ▷

8.6 Discussion

I began this chapter with the desire to add linear types to Alms, a general-purpose programming language with affine types and exceptions. The treatment of exceptions in §8.5.3 points the way toward that goal. One question that remains, however, concerns the pragmatics of checked exceptions in a higher-order language such as Alms, where latent exception effects are likely to appear on many function arrows. Weighing the cost against the benefit, I

have decided that adding linear types to Alms is not worth the complexity of a programmer-visible effect system.

One potential direction for future research is to consider how other control effects fit into my general framework. I suspect that some control operators common to imperative languages, such as *return*, *break*, and *goto*, absent first-class labels, would be straightforward. More exotic forms of control may be harder. Some control operators, such as *shift0*, are very difficult to type even in a simpler setting (Kiselyov and Shan 2007), which is why I have not considered them. Others, such as Felleisen's *prompt* and *control* (1988) may be tractable with a more expressive version of my generic type system, because effects need to reflect not only how an expression treats its continuation, but how a continuation, if captured and reinvoked, treats *its* new continuation.

CHAPTER 9

Related Work and Design Rationale

ALMS IS RELATED to a variety of other programming languages, and several of its novel features are inspired by similar features in other languages. In this chapter I discuss related research, place Alms in the context of similar work, and show how other languages influenced the design of Alms.

9.1 Substructural Type Systems

Much of the prior work on substructural type systems appears as background in chapter 2. In this section, I discuss several additional languages with substructural types and relate them to Alms.

9.1.1 System F°

Mazurak et al. (2010) describe a calculus of "lightweight linear types." Their primary motivation is similar to mine: to remove needless overhead and provide a "simple foundation for practical linear programming."

System F° and Alms share several important ideas:

- Both use kinds to distinguish linear (in Alms, affine) types from unlimited types, where F°'s kinds $\circ$ and $\star$ correspond to my A and U, and their subkinding relation $\star \leq \circ$ corresponds to my $\mathsf{U} <: \mathsf{A}$. (Kinds are used in a similar fashion by Charguéraud and Pottier (2008), who use

several kinds to distinguish unlimited values from affine capabilities, and Swamy et al. (2010), who have base kinds $\star$ and A.)

- F$^\circ$ uses existential types and subkinding to abstract unlimited types into linear types. Alms (the language) uses modules and ${}^a\lambda_{ms}$ (the calculus) uses higher-kinded type abstraction to define abstract affine types, including type constructors with parameters. Mazurak et al. mention the possibility of extending F$^\circ$ with abstraction over higher kinds but do not show the details.

- They sketch out a notation for writing linear computations, which inspired Alms's implicit threading syntax (§3.2.1). Mazurak et al.'s sugar requires explicit sequencing but not explicit threading, whereas Alms's requires neither and instead uses the sequencing determined by the evaluation order.

There are also notable differences:

- F$^\circ$ has linear types, which disallow weakening, whereas Alms has affine types, which allow it. This is a trade-off. Linear types make it possible to enforce liveness properties, which may be useful, for instance, to ensure that manual memory management does not leak. On the other hand, safely combining linearity with exceptions requires a type-and-effect system to track when raising an exception would implicitly discard linear values (chapter 8). Alms can support explicit deallocation so long as failure to do so is backed up by a garbage collector.

- Alms's unlimited-use function type is a subtype of its one-use function type. F$^\circ$ does not provide subtyping, though they do show how η expansion can explicitly perform the coercion that Alms's subtyping does implicitly. Experience with Alms confirms that dereliction subtyping is valuable, though it does come at the cost of complexity.

- F$^\circ$ requires annotating abstractions ($\lambda^\kappa x{:}\tau.e$) to specify the kind of the resulting arrow type, which may only be $\star$ or $\circ$. Alms refines this with qualifier expressions and selects the least kind automatically.

- Mazurak et al. give a resource-aware semantics and prove that they can encode regular protocols. I do neither but conjecture that my system enjoys similar properties, except that weakening makes it possible to bail out of a protocol at any point.

- Their sketch of rules for algebraic datatypes is similar to how mine work, though mine are strictly stronger. For example, an option type in F° would have two versions:

  $$\mathsf{optionLin} : \circ \Rightarrow \circ \qquad \mathsf{optionUn} : \star \Rightarrow \star.$$

  Dependent kinds in Alms allow defining one type constructor whose kind subsumes both:

  $$\mathsf{option} : \Pi(\text{`}a^{+}).\, \langle \text{`}a \rangle.$$

In general, dependent kinds and subtyping in Alms should allow for more code reuse than in F°.

9.1.2 Fine

Swamy et al.'s (2010) Fine programming language combines affine types, dependent types, and refinement types in a security-oriented language. They use affine types to track the state of authorization protocols, which ensures that a programmer cannot accidentally use a stale authorization state.

Like Alms and F°, Swamy et al. use kinds to distinguish affine types (of kind `A`) from unlimited types (of kind $\star$). Like F°, and unlike Alms, their system does not support any sort of kind polymorphism. They would seem to require separate affine and unlimited type declarations, as in the option type example for F°.

Swamy et al. also impose two restrictions on where affine types may appear. First, affine type variables may not appear in dependent types. This restriction simplifies the metatheory, but it is not clear what the implications are in practice. Second, they require that for any kind of the form $\mathtt{A} \to \kappa$, kind κ must eventually end with `A`. They claim that this restriction is necessary for

soundness,[1] but compared to Alms, this restriction is quite limiting. Consider, for example, this Alms type declaration:

$\mathsf{type}\ {}'a\ \mathsf{id} = Id\ \mathsf{of}\ {}'a \to {}'a$

In Alms, type constructor id has kind $\Pi({}'a).\,\mathsf{U}$, because the kind of the actual parameter does not determine whether an unlimited function can be duplicated. The kind restriction in Fine means that id must have either kind $\star \to \star$ or kind $\mathtt{A} \to \mathtt{A}$, seriously limiting its applicability.

Fine is implemented by a type-preserving compiler that targets a dependently typed extension of the .NET intermediate language, which means that Fine modules may be linked against code written in other .NET languages such as C#. Fine prevents foreign .NET code from violating its affine invariants by a syntactic check on the shapes of types exported to and imported from .NET, which disallows the escape of affine values (Nikhil Swamy 2011, personal communication). Fine's type checker generates a large number of proof obligations; it uses Z3, an SMT solver, to discharge them. Combining affine types with dependent and refinement types means that Fine interfaces can express a variety of invariants that Alms cannot.

9.1.3 ATS, Cyclone, and Vault

Substructural types have proven popular for tracking resources, especially memory, in safe, low-level languages.

ATS (Zhu and Xi 2005) has a notion of *stateful views*, which are linear capabilities witnessing that a particular memory location holds a value of a particular type. In ATS, stateful views support explicit memory allocation and deallocation, strong updates, and even pointer arithmetic, all without violating memory safety.

Cyclone (Grossman et al. 2002) is a type-safe dialect of C that uses regions for memory management. Early versions of Cyclone manage memory using

[1] In particular, they write that "types constructed from affine types must themselves be affine—this is standard." It is often the case that *values* constructed from affine *values* must be affine, but extending this rule to types is a very conservative approximation that gives up significant expressiveness.

using nested regions as in Tofte and Talpin (1997, see my §2.3.1). In addition, Cyclone uses a global "heap region" that is conservatively garbage collected. For cases whether neither static regions nor tracing garbage collection are suitable, later versions of Cyclone add unique pointers and temporary aliasing mechanisms, whose theory is based on substructural types.

Vault (DeLine and Fähndrich 2001) uses linear capabilities for typestate. Vault keys double as dynamic regions, which can guard dynamically-allocated objects, as in §2.3.3. Typestate in Vault is also discussed in detail in §2.2.

As shown in §4.2, Alms can express variations on Vault-style regions. It is unlikely, however, that Alms can support statically-checked pointer arithmetic, as ATS does using dependent types. Similarly, Cyclone supports subtyping between regions based on nested lifetimes, and it can often infer where this subtyping is necessary, whereas nested regions in Alms would probably require explicit management of subtyping proofs. Vault's treatment of capabilities may be more convenient to use than Alms's, because while Alms requires explicit threading of capability values, Vault's key sets are tracked automatically within function bodies. On the other hand, because capabilities in Alms appear as ordinary values, we may combine them using the native intuitionistic logic of Alms's type system. Instead, Vault provides a simple language for expressing function pre- and postconditions. For more complicated logic, Vault allows embedding capabilities in variant values and tracking them dynamically. In Alms, by contrast, the full language—algebraic data types, anonymous sums and products, open variants and records, exceptions, and reference cells—is available for managing capabilities.

9.1.4 Clean

The Clean programming language (Brus et al. 1987) relies on uniqueness typing (§2.1.2) to track side effects in a pure language. Uniqueness types enforce single-threading of mutable state, which means that mutation in Clean does not violate referential transparency. In Clean, a unique type is a subtype of a non-unique type with the same representation. For example, a unique array may be updated, and the update operation returns a new unique

array; or a unique array may become aliased, via subtyping, at which point the array becomes immutable.

There is a strong similarity between Alms's kinding judgment and Clean's uniqueness propagation rules that relate the uniqueness of data structures to that of their constituent parts. Specifically, if some complex value is shared then its components are shared as well.

While Clean supports subtyping, it does not have a subkinding relation analogous to Alms's or F$^\circ$'s. In particular, Clean requires that the uniqueness attributes declared for an abstract type in a module's interface exactly match the uniqueness attributes in the module's implementation.

9.1.5 Sing#

Microsoft's experimental Singularity operating system is written in Sing# (Fähndrich et al. 2006), a high-level systems programming language that extends Spec#. Sing# has built-in support for *channel contracts,* which are a form of session type providing static checking of communication protocols between device drivers and other services. Unlike more idealistic linear systems, the design acknowledges the need to allow for failure: Every protocol implicitly includes branches to close the channel at any point. In other words, session types in Sing#, as in Alms (§4.3), are affine.

9.2 The Spirit of ML

The overarching design of Alms puts it squarely in the ML family. Alms includes algebraic data types, exceptions, and a module system all closely modeled on other ML-like languages (Leroy et al. 2011; Milner et al. 1997). The concrete syntax of Alms is based on OCaml (Leroy et al. 2011). Its global type inference is a conservative extension of Damas and Milner's (1982), in that if a term has some type scheme under their type system—and does not violate affinity—then it has the same or a more general type scheme in Alms.

In a broader sense, Alms is an answer to the question: *What must be done to smoothly integrate substructural types with ML?* Some language design

choices are dictated directly by features imported from ML. For example, the inclusion of ML-style exception handling meant that linear types would have required a type-and-effect system (as in chapter 8), which lead to the adoption of affine types instead. Similarly, ML-style algebraic data types and abstract types in Alms lead to Alms's expressive kind system, which allows the kind of a type constructor application to depend on the kinds of the actual type parameters.

Kind subsumption in Alms is akin to Standard ML's treatment of equality types (Milner et al. 1997) and OCaml's treatment of type constructor variance (Leroy et al. 2011). In both cases, these properties of types may be considered as kinds with subkinding orders, and in both cases types may be abstracted to higher kinds. In SML, for example, **eqtype** subsumes **type**, in that signature matching can abstract an equality type to a non-equality type but not vice versa. Likewise, an ordinary (non-equality) type variable may be instantiated to an equality type, but an equality type variable may not be instantiated to a non-equality type in SML.

Dependent kinds in Alms may be seen as a generalization of Standard ML's kind system for equality types. Consider an abstract **eqtype** constructor of arity k:

eqtype ('a_1, …, 'a_k) c

In Standard ML, type (t_1, …, t_k) c is an equality type if all of types t_1, …, t_k are equality types. Given the kind order that EQTYPE is less than TYPE, we could thus say that type constructor c has kind

$$\Pi(\text{'}a_1, \ldots, \text{'}a_k).\ \langle \text{'}a_1 \rangle \sqcup \cdots \sqcup \langle \text{'}a_k \rangle.$$

In fact, in Standard ML every k-ary type constructor that admits equality has that kind, and every type constructor that does not admit equality has kind $\Pi(\text{'}a_1, \ldots, \text{'}a_k).$ TYPE, with the exception of some built-in type constructors, such as ref, which has kind $\Pi(\text{'}a).$ EQTYPE. However, there is no way to give a user-defined type a kind like ref has, which expresses the fact that reference

cells support equality regardless of the type of their contents. For example, consider this algebraic data type definition:

$$\mathsf{datatype}\ (\text{'}a, \text{'}b)\ \mathsf{d} = D\ \mathsf{of}\ \text{'}a \times \text{'}b\ \mathsf{ref}$$

In Standard ML, type (int, int → int) d is not an equality type, even though its representation type int × (int → int) ref is an equality type, because d effectively has kind $\Pi(\text{'}a_1, \text{'}a_2).\ \langle \text{'}a_1 \rangle \sqcup \langle \text{'}a_2 \rangle$. Using Alms-style dependent kinds, type constructor d could have kind $\Pi(\text{'}a_1, \text{'}a_2).\ \langle \text{'}a_1 \rangle$, which reflects the fact that equality for d does not depend on the second type parameter.

9.2.1 Beyond ML

Inferring affine types with function type qualifiers and two kinds of type variables is not very far from the type inference in ML-like languages. De Vries et al. (2007) show that after a simple analysis to discover the use constraints on variables, ordinary type inference in the style of Damas and Milner (1982) can handle substructural types. However, Alms also depends on two type system features that are not as simple to add to ML-style type inference: subtyping and first-class polymorphism.

Subtyping. Alms uses subtyping for dereliction, in order to allow supplying an unlimited function to a context that will use it but once, and extends the same subtyping to the higher-order case. Alms follows Odersky et al.'s (1999) approach to inference for subtyping based on constrained types. For solving constraints, the implementation of Alms uses techniques from Simonet (2003).

First-class polymorphism. In addition to subtyping, Alms offers rank-n polymorphism and impredicativity, together known as first-class polymorphism. ML-style prenex types are insufficient to support a common pattern for typestate with capabilities whereby a value and its capability are tied together using existential quantification (as in figure 4.2 on p. 60). Several systems for global type inference with first-class polymorphisms have been proposed, such as Jones's (1997) FCP, Le Botlan and Rémy's (2003) ML^{F}, and Russo and Vytiniotis's (2009) QML. Alms takes its approach to first-

class polymorphism from Leijen's (2008) HMF, extended with Leijen's (2006) technique for automatically packing and unpacking existential types.

9.3 From ILL to Alms

In another sense, Alms is an answer to the question: *What does it take to turn intuitionistic linear logic into a real programming language?* Recall from §2.1 that in Bierman's (1993) intuitionistic linear logic, following Girard (1987), the exponential modality ! is used to indicate resources that may be freely duplicated (or dropped), and there are rules for introducing and eliminating the exponential:

$$\text{ILL-Contraction}\quad \frac{\Gamma, !\sigma, !\sigma \vdash \tau}{\Gamma, !\sigma \vdash \tau} \qquad \text{ILL-Promotion}\quad \frac{!\Gamma \vdash \sigma}{!\Gamma \vdash !\sigma} \qquad \text{ILL-Dereliction}\quad \frac{\Gamma \vdash !\sigma}{\Gamma \vdash \sigma}$$

While ILL provides a good starting point, I contend that the treatment of the exponential in ILL is unsuitable for a high-level programming language for several reasons. Types that differ only in placement of ! are inhabited by different terms, which inhibits code reuse; at the same time, programs are cluttered with explicit dereliction and promotion syntax for managing exponentials. Furthermore, the resulting types are unnecessarily noisy, and linear types are shorter than unlimited types, which is the wrong default because unlimited values are likely to be more common than affine values. I discuss how Alms and some related work contend with each of these issues in turn.

Implicit dereliction and promotion. To avoid the clutter of explicit promotion and dereliction, exponential introduction and elimination should instead be implicit. Thus, it is necessary to identify where to derelict and promote. One answer is to promote wherever possible and derelict whenever necessary, which suggests, as in Wadler (1991), that we treat dereliction as subtyping. Then promoting whenever and as much as possible amounts to finding principal types in the dereliction-subtyping order (*cf.* theorem 5.3). Wadler showed that

adding a subtyping relation based on dereliction does not result in principal types, and he introduced use variables in order to recover them (§2.1.1).

Wadler's (1991) standard linear types and Ahmed et al.'s (2005) λ^{URAL} both modify the syntax of types and in doing so make dereliction and promotion implicit. Instead of an exponential connective that needs to be introduced and eliminated, they build use variables or qualifiers—which determine which structural operations are supported—into the syntax of types. Then promotion, as selection of use variables or qualifiers, is built into the introduction rules for other connectives, and dereliction in built into the elimination rules. Consider, for example, the rules for introducing and eliminating function types in λ^{URAL}:

$$\text{URAL-}\multimap\text{I}\quad \frac{\Delta \vdash \xi : \mathsf{QUAL} \qquad \Delta \vdash \Gamma \preceq \xi \qquad \Delta; \Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Delta; \Gamma \vdash \lambda x.e : {}^{\xi}(\tau_1 \multimap \tau_2)}$$

$$\text{URAL-}\multimap\text{E}\quad \frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Delta; \Gamma_1 \vdash e_1 : {}^{\xi}(\tau_1 \multimap \tau_2) \qquad \Delta; \Gamma_2 \vdash e_2 : \tau_1}{\Delta; \Gamma \vdash e_1\, e_2 : \tau_2}.$$

The introduction rule selects qualifier ξ based on the qualifiers of the types in environment Γ, and gives the resulting type that qualifier. This rule thus includes and generalizes promotion, because it gives a permissive qualifier to a function type when the context allows. The elimination rule allows applying a function with any qualifier, which means that dereliction to remove an exponential before applying a function is unnecessary.

Aside from the mechanics of context splitting, the rules in Alms are very similar to λ^{URAL}:

$$\text{ALMS-}\multimap\text{I}\quad \frac{\Gamma \vdash \Sigma \preceq \xi \qquad \vdash (\Gamma; \Sigma), x{:}\tau_1 \rightsquigarrow \Gamma'; \Sigma' \qquad \Gamma'; \Sigma' \rhd e : \tau_2}{\Gamma; \Sigma \rhd \lambda x{:}\tau_1.e : \tau_1 \xrightarrow{\xi} \tau_2}$$

$$\text{ALMS-}\multimap\text{E}\quad \frac{\Gamma; \Sigma_1 \rhd e_1 : \tau_1 \xrightarrow{\xi} \tau_2 \qquad \Gamma; \Sigma_2 \rhd e_2 : \tau_1}{\Gamma; \Sigma_1, \Sigma_2 \rhd e_1\, e_2 : \tau_2}$$

In particular, the function type introduction rule also includes promotion, by selecting a qualifier bounded by the qualifiers in environment Σ, and the

elimination rule includes dereliction, by eliminating function types with any qualifier. A notable difference from λ^{URAL}, however, is that promotion in Alms selects a qualifier expression which is the least upper bound of the qualifiers in Σ, rather than bounding the environment by a constant.

Use and qualifier polymorphism. Besides making promotion and dereliction implicit, both use variables and qualifier variables support parametric polymorphism over linearity. This eliminates the problem in ILL that types differing only in placement of the exponential require separate terms to inhabit them. For example, consider two possible ILL type schemes for a function that composes two functions:

$$(\rho \multimap \tau) \multimap (\sigma \multimap \rho) \multimap \sigma \multimap \tau \qquad\qquad !(\rho \multimap \tau) \multimap !(\sigma \multimap \rho) \multimap !(\sigma \multimap \tau)$$

The first type composes two one-shot functions into a one-shot function; the second composes two unlimited functions into an unlimited function. Even with promotion and dereliction made implicit, without polymorphism over exponentials, it may be necessary to have two different composition functions with these two unrelated types. With standard types or λ^{URAL}, use and qualifier polymorphism allow having one function for both cases. In Alms, parametric polymorphism using ordinary type variables, which may appear in qualifier expressions, accomplishes the same thing:

standard types	$!^{\mu}(\rho \multimap \tau) \multimap !^{\mu}(\sigma \multimap \rho) \multimap !^{\mu}(\sigma \multimap \tau)$
λ^{URAL}	$^{\alpha}(\rho \multimap \tau) \multimap {}^{\alpha}({}^{\alpha}(\sigma \multimap \rho) \multimap {}^{\alpha}(\sigma \multimap \tau))$
Alms	$(\rho \xrightarrow{`a} \tau) \to (\sigma \xrightarrow{`a} \rho) \to \sigma \xrightarrow{`a} \tau$

Qualifier kinds. Use variables and qualifier variables allow implicit dereliction and promotion and increase polymorphism, but at the cost of very verbose types, because the syntax of types includes a use or a qualifier on every type constructor. For example, the type of a function that adds two integers in λ^{URAL} might be

$$^{\mathsf{U}}({}^{\mathsf{U}}({}^{\mathsf{U}}\mathsf{int} \otimes {}^{\mathsf{U}}\mathsf{int}) \multimap {}^{\mathsf{U}}\mathsf{int}).$$

Dependent qualifier kinds arise from an observation that well-formedness rules for types imply bounds on qualifiers. For example, the product introduction rule in λ^{URAL} requires the qualifier on a product type to upper bound the

qualifiers of the components of the product:

$$\text{URAL-}\otimes\text{I}\quad \frac{\begin{array}{c}\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \\ \Delta;\Gamma_1 \vdash v_1 : \tau_1 \qquad \Delta \vdash \tau_1 \preceq \xi \\ \Delta;\Gamma_2 \vdash v_2 : \tau_2 \qquad \Delta \vdash \tau_2 \preceq \xi\end{array}}{\Delta;\Gamma \vdash \langle v_1, v_2\rangle : {}^{\xi}(\tau_1 \otimes \tau_2)}$$

If ξ_1 and ξ_2 are the least qualifiers of types τ_1 and τ_2, respectively, then this amounts to a bound

$$\xi \sqsupseteq \xi_1 \sqcup \xi_2.$$

Noting that choosing the least qualifier ξ yields the most permissive usage of the resulting product type, we might choose to set ξ equal to its bound:

$$\xi = \xi_1 \sqcup \xi_2.$$

But if ξ is set equal to the bound, then it need not be written explicitly in the type, because it can always be inferred from the qualifiers of τ_1 and τ_2. This is precisely what the dependent kind $\Pi(\text{`}a, \text{`}b).\langle \text{`}a\rangle \sqcup \langle \text{`}b\rangle$ accomplishes in Alms.

CHAPTER 10

Conclusion

SUBSTRUCTURAL TYPES ARE expressive but, in prior incarnations, unwieldy. Special-purpose stateful type systems can be practical, but they are designed for tracking specific kinds of state. Alms demonstrates that a programming language with general-purpose substructural types can be both practical and expressive.

10.1 Contributions

To support my thesis, I have designed and implemented Alms, a programming language with substructural types. I developed a model of Alms and proved that the model enjoys two beneficial properties, type soundness and principal qualifiers.

Linear type systems that rely on exponential connectives suffer from lack of polymorphism between linear and unlimited types, which can result in code duplication. Adding use variables or qualifiers ameliorates the duplication problem, but at the cost of verbose types. In the design of Alms, I combine existing language features—notably qualifier kinds and dereliction subtyping—with several novel ones in order to solve both problems and to enable the design and use of a variety of typed, stateful abstractions:

- Alms treats the standard subqualifier relation ($\mathsf{U} \sqsubseteq \mathsf{A}$) as a subkinding relation; combined with modules and sealing, subkinding supports the design of affine abstractions with unlimited representations (p. 41).

- Dereliction subtyping and principal promotion (p. 45) together ensure that functions are given qualifiers that are as permissive as possible and may be used in less permissive contexts as needed; this increases the possibilities for reusing functional abstractions with both affine and unlimited types.

- Dependent qualifier kinds (p. 44) reflect natural relationships between compound types and their components, which allows the same type constructors to be used for both affine and unlimited data, but without the need for awkward qualifier annotations.

- Arrow qualifier inference (§3.2.2 on p. 55) significantly reduces the need for qualifier annotations on function types, making the concrete syntax of types less cluttered.

- The implicit threading syntax (§3.2.1 on p. 52) makes affine values easier to use by automatically threading them through a program.

These language features together, in Alms, form a coherent design, as evidenced by the code examples in chapters 3 and 4. An important result of this combination of features is that Alms programs, despite their affine types, look similar to OCaml programs.

In addition to Alms itself, my contributions include solutions to two practical problems in the design of substructural programming languages: how to integrate code written in a substructural language with code written in a legacy language (chapter 7), and how to safely add control operators to a language with substructural types (chapter 8). While these solutions are not implemented in Alms, they should be valuable to future designers of practical substructural languages.

10.2 Future Work

Alms is available for download today, and it is already a useful platform for exploratory programming with substructural types. However, much work

remains to be done to make Alms suitable for production use and to investigate further language design possibilities suggested by my research here.

10.2.1 A Run-Time Story

While Alms's type checker is very sophisticated, the implementation of its dynamics—a mere 4% of the line count—is a weak point. Alms programs are currently evaluated by the simplest possible interpreter, with primitive operations implemented in Haskell (Peyton Jones 2003). There are two significant downsides to this approach. First, it means that Alms programs run very slowly. Second, and more importantly, it means that the run-time system currently cannot support efficient implementations of manually-managed resources, which is an important application of affine types. Consider, for example, region-based memory management. While Alms's type system is up to the task of expressing interfaces for regions, the run-time system provides no mechanism for implementing region operations in a reasonable way.

10.2.2 Affine and Uniqueness Types, Together

Affine types prevent future aliasing, whereas uniqueness types keep track of past aliasing. Both approaches are useful, because while affine types can enforce single threading of values, uniqueness types can allow aliasing while limiting some operations to unaliased values.

In §6.3.3 (p. 139), we saw that with qualifier kinds the distinction between affine and uniqueness types manifests as a difference in the monotonicity of the kinding relation: monotone kinding yields affine types and antitone kinding yields uniqueness types. The discussion at the end of that section raises the possibility of integrating the two in a single type system, but ultimate rejects that idea, in favor of admitting a constraint-solving rule that relies on a monotone kinding relation. However, a dual rule is sound in the presence of antitone kinding. With suitable kind structure, it may be possible to distinguish monotone-kinded and antitone-kinded types in the same system. Ensuring that constraint-solving rules that rely on monotonicity

are applied correctly might allow the inclusion of both affine and uniqueness types in the same language.

10.2.3 Other Applications of a Novel Idea

Some type system features in Alms are applicable beyond the world of substructural types. For example, dependent qualifier kinds can be viewed as an extension of Standard ML's (Milner et al. 1997) kind system for equality types that allows the qualifier of a type application to depend on the qualifiers of the type parameters (§9.2 on p. 250). The idea of dependent kinds can be generalized, however, to allow other properties of type applications to depend on properties of the parameters. In a language where kinds track mutability, dependent kinds can provide a precise, type-based analysis of which values have mutable components; if either component of a product type is mutable, then values of the product type are observably mutable as well. Similarly, dependent kinds might be useful in a language that uses kinds to control information flow.

10.2.4 Exploring the Design Space

Alms represents but one point in a design space of practical, general-purpose substructural type systems. For example, Alms distinguishes affine and unlimited type variables, similarly to how Standard ML distinguishes equality and non-equality type variables. Haskell abandoned equality types in favor of a more general mechanism, type classes (Wadler and Blott 1989). Type classes allow programmers to specify the equality operation for a type directly by declaring an instance of type class Eq; furthermore, the equality operation for an applied type constructor can be specified in terms of equality operations for its parameters. By analogy, instead of two kind of type variables, unlimited types in Alms could simply be members of a type class Dup, which includes a duplication operation. Then programmers could specify duplication operations for new types, and dependent product kinds would be replaced by Dup instances that specify how to duplicate a compound type in terms of

duplication operations for its components. Such an approach would integrate well with Alms's constraint-based type inference.

APPENDIX A

Additional Proofs for Chapter 5

A.1 Preliminaries

I follow these two conventions for splitting contexts throughout this chapter:

- If there is a context Σ_X and I introduce Σ_{X1} and Σ_{X2}, this means that $\Sigma_X = \Sigma_{X1}, \Sigma_{X2}$, up to permutation.
- Contexts Σ_{X1} and Σ_{X2} are taken to be disjoint.

OBSERVATION A.1 (Strengthening)**.**

1. *Type- and kind-level judgments are unaffected by variables and locations in the domain of the context. Let* Γ' *be a context with no type variable bindings, that is, there are no* $\alpha \in \operatorname{dom} \Gamma'$*. Then:*
   a) *If* $\Gamma, \Gamma' \vdash \kappa$ *then* $\Gamma \vdash \kappa$.
   b) *If* $\Gamma, \Gamma' \vdash \tau : \kappa$ *then* $\Gamma \vdash \tau : \kappa$.
   c) *If* $\Gamma, \Gamma' \vdash \tau_1 <:^{\mathfrak{v}} \tau_2$ *then* $\Gamma \vdash \tau_1 <:^{\mathfrak{v}} \tau_2$.
   d) *If* $\Gamma, \Gamma' \vdash \kappa_1 <: \kappa_2$ *then* $\Gamma \vdash \kappa_2 <: \kappa_2$.
   e) *If* $\Gamma, \Gamma' \vdash \alpha \in \tau \updownarrow \mathfrak{v}$ *then* $\Gamma \vdash \alpha \in \tau \updownarrow \mathfrak{v}$.
   f) *If* $\Gamma, \Gamma' \vdash \Sigma \preceq \xi$ *then* $\Gamma \vdash \Sigma \preceq \xi$.
   g) *If* $\vdash (\Gamma_0; \Sigma_0), \Gamma' \rightsquigarrow \Gamma; \Sigma$ *and one of the preceding judgments holds in* Γ *then it holds in* Γ_0 *as well.*

2. *The context bounding judgment is unaffected by type variables in the domain of the subject context; the context well-formedness judgments are unaffected by type variables in the domain of the affine context. Let* Γ' *be a context with no variables nor locations, only type variables. Then:*

   a) *If* $\Gamma \vdash \Sigma, \Gamma' \preceq \xi$ *then* $\Gamma \vdash \Sigma \preceq \xi$.

   b) *If* $\vdash \Gamma; \Sigma, \Gamma'$ *then* $\vdash \Gamma; \Sigma$.

   c) *If* $\Gamma; \Sigma, \Gamma' \rhd e : \tau$ *then* $\Gamma; \Sigma \rhd e : \tau$.

Justification.

1. a–f) By inspection of the rules we can see that these judgments never directly observe variables or locations in their context, and they never indirectly observe them through appeal to some other judgment not sharing this property.

   g) Observe that Γ may differ from Γ_0 only by adding the types of some variables.

2. a) By induction on the length of Γ'.

   b) By the previous subpart.

   c) Observe that no rule for the typing judgment looks up a type variable in Σ, and furthermore, the only rules to which it passes Σ are covered by the previous two subparts.

LEMMA A.2 (Qualifier substitution on kind well-formedness)**.**

If $\Gamma, \alpha{:}\langle\alpha\rangle \vdash \kappa$ *and* $\Gamma \vdash \xi$ *then* $\Gamma \vdash \{\xi/\alpha\}\kappa$.

Proof. By induction on κ, with one non-trivial case:

Case $\Pi\beta^{\upsilon}.\kappa'$**.**

By inversion of rule OK-OPER,

(1) if $\beta \in \mathrm{FTV}(\kappa')$ then $+ \sqsubseteq \upsilon$ and

(2) $\Gamma, \alpha{:}\langle\alpha\rangle, \beta{:}\langle\beta\rangle \vdash \kappa'$.

By the induction hypothesis,

(3) $\Gamma, \beta{:}\langle\beta\rangle \vdash \{\xi/\alpha\}\kappa'$.

Because β is bound with only κ' in its scope, by renaming, we know that $\beta \notin \mathrm{FTV}(\xi)$ and $\beta \neq \alpha$, so $\beta \in \mathrm{FTV}(\{\xi/\alpha\}\kappa')$ if and only if $\beta \in \mathrm{FTV}(\kappa')$; Thus,

(4) if $\beta \in \mathrm{FTV}(\{\xi/\alpha\}\kappa')$ then $+ \sqsubseteq \mathfrak{v}$.

Then by rule OK-OPER. □

LEMMA A.3 (Context splitting well-formedness)**.**

$\vdash \Gamma; \Sigma_1, \Sigma_2$ *if and only if* $\vdash \Gamma; \Sigma_1$ *and* $\vdash \Gamma; \Sigma_2$.

Proof. By induction on Σ_2. □

LEMMA A.4 (Regularity)**.**

Most judgments in ${}^{a}\lambda_{ms}$ *are defined only over well-formed terms:*

1. *If* $\Gamma \vdash \kappa$ *then* $\vdash \Gamma$.
2. *If* $\vdash \Gamma$ *then* $\Gamma \vdash \kappa$ *for all* $\alpha{:}\kappa \in \Gamma$.
3. *If* $\Gamma \vdash \xi_1 <: \xi_2$ *then* $\Gamma \vdash \xi_1$ *and* $\Gamma \vdash \xi_2$.
4. *If* $\Gamma \vdash \tau : \kappa$ *then* $\Gamma \vdash \kappa$.
5. *If* $\Gamma \vdash \alpha \in \tau \updownarrow \mathfrak{v}$ *then* $\Gamma \vdash \tau : \kappa$ *for some* κ.
6. *If* $\Gamma \vdash \tau_1 <:^{\mathfrak{v}} \tau_2$ *then* $\Gamma \vdash \tau_1 : \kappa_1$ *and* $\Gamma \vdash \tau_2 : \kappa_2$ *for some* κ_1 *and* κ_2.
7. *If* $\Gamma \vdash \Sigma \preceq \xi$ *then* $\Gamma \vdash \xi$ *and* $\vdash \Gamma, \Sigma$.
8. *If* $\vdash \Gamma; \Sigma$ *then* $\Gamma \vdash \Gamma \preceq \cup$ *and* $\Gamma \vdash \Sigma \preceq \xi$ *for some* ξ
9. *If* $\vdash (\Gamma_0; \Sigma_0), \Sigma' \rightsquigarrow \Gamma; \Sigma$ *then* $\vdash \Gamma_0; \Sigma_0$ *if and only if* $\vdash \Gamma; \Sigma$.
10. *If* $\Gamma; \Sigma \rhd e : \tau$ *then* $\vdash \Gamma; \Sigma$ *and* $\Gamma \vdash \tau : \xi$ *for some* ξ.

Proof.

1. By induction on κ.

2. By induction on the derivation.

3. By inversion of rule OK-QUAL.

4. By induction on the derivation, using lemma A.2 for the K-APP and K-ALL cases.

5. By induction on the derivation.

6. By induction on the derivation, using part (3) for the TSUB-ARR case.

7. By induction on the derivation:

   Case $\dfrac{\vdash \Gamma}{\Gamma \vdash \bullet \preceq \mathsf{U}}$.

   Then $\Gamma \vdash \mathsf{U}$ by rule OK-QUAL and $\vdash \Gamma, \bullet$ from the premise.

   Case $\dfrac{\Gamma \vdash \Sigma' \preceq \xi_1 \qquad \Gamma \vdash \tau : \xi_2}{\Gamma \vdash \Sigma', x{:}\tau \preceq \xi_1 \sqcup \xi_2}$.

   By the induction hypothesis, $\Gamma \vdash \xi_1$ and $\Gamma \vdash \xi_2$, so $\Gamma \vdash \xi_1 \sqcup \xi_2$ by rule OK-JOIN.

   By the induction hypothesis, $\vdash \Gamma, \Sigma'$, so $\vdash \Gamma, \Sigma', x{:}\tau$ by rule WF-CONSX.

   Case $\dfrac{\Gamma \vdash \Sigma' \preceq \xi_1 \qquad \Gamma \vdash \tau : \xi_2}{\Gamma \vdash \Sigma', \ell{:}\tau \preceq \mathsf{A}}$.

   By the induction hypothesis, part (1), and rules OK-QUAL and WF-CONSL.

   Case $\dfrac{\Gamma \vdash \Sigma' \preceq \xi \qquad \Gamma \vdash \kappa}{\Gamma \vdash \Sigma', \alpha{:}\kappa \preceq \xi}$.

   By the induction hypothesis, $\Gamma \vdash \xi$. By the induction hypothesis and rule WF-CONSA, $\vdash \Gamma, \Sigma', \alpha{:}\kappa$.

8. By inversion of rule WF.

9. By induction on the derivation:

   Case $\dfrac{\vdash \Gamma;\Sigma}{\vdash (\Gamma;\Sigma), \bullet \rightsquigarrow \Gamma;\Sigma}$.

   Trivial.

   Case $\dfrac{\Gamma_0 \vdash \tau : \mathsf{U} \qquad \vdash (\Gamma_0, x{:}\tau; \Sigma_0), \Sigma'' \rightsquigarrow \Gamma;\Sigma}{\vdash (\Gamma_0;\Sigma_0), x{:}\tau, \Sigma'' \rightsquigarrow \Gamma;\Sigma}$.

   Suppose that $\vdash \Gamma_0;\Sigma_0$. Then by inversion of rule WF,

   (1) $\Gamma_0 \vdash \Gamma_0 \preceq \mathsf{U}$ and

   (2) $\Gamma_0 \vdash \Sigma_0 \preceq \xi$

   for some ξ. Then,

   (3) $\Gamma_0 \vdash \Gamma_0, x{:}\tau \preceq \mathsf{U}$ by (1), B-ConsX

   (4) $\Gamma_0, x{:}\tau \vdash \Gamma_0, x{:}\tau \preceq \mathsf{U}$ by weakening

   (5) $\Gamma_0, x{:}\tau \vdash \Sigma_0 \preceq \xi$ by (2), weakening

   (6) $\vdash \Gamma_0, x{:}\tau; \Sigma_0$ by WF

   (7) $\vdash \Gamma;\Sigma$ by I.H.

   Conversely, suppose that $\vdash \Gamma;\Sigma$. Then,

   (1) $\vdash \Gamma_0, x{:}\tau; \Sigma_0$ by I.H.

   (2) $\Gamma_0, x{:}\tau \vdash \Gamma_0, x{:}\tau \preceq \mathsf{U}$ by inv. WF

   (3) $\Gamma_0, x{:}\tau \vdash \Sigma_0 \preceq \xi$ by inv. WF

   (4) $\Gamma_0 \vdash \Gamma_0, x{:}\tau \preceq \mathsf{U}$ by observation A.1

   (5) $\Gamma_0 \vdash \Sigma_0 \preceq \xi$ by observation A.1

   (6) $\Gamma_0 \vdash \Gamma_0 \preceq \mathsf{U}$ by inv. B-ConsX

   (7) $\vdash \Gamma_0;\Sigma_0$ by WF.

   Case $\dfrac{\Gamma_0 \vdash \tau : \xi \qquad \vdash (\Gamma_0;\Sigma_0, x{:}\tau), \Sigma'' \rightsquigarrow \Gamma;\Sigma}{\vdash (\Gamma_0;\Sigma_0), x{:}\tau, \Sigma'' \rightsquigarrow \Gamma;\Sigma}$.

   As in the previous case.

10. By induction on the derivation, using part (9) and lemma A.3 as necessary. □

A.2 Principal Qualifiers

DEFINITION A.5 (Kind semilattices)**.**

Partition the kinds by arity, where κ^j are the kinds of arity j:

$$\kappa^0 ::= \xi$$
$$\kappa^{j+1} ::= \Pi\alpha^{\mathfrak{v}}.\kappa^j.$$

Then, subject to a context Δ mapping type variables to kinds, $(\kappa^j, <:, \sqcup^j, \bot^j)$ is a bounded join semilattice, defined inductively for each arity j by:

$$\xi_1 \sqcup^0 \xi_2 = \xi_1 \sqcup \xi_2$$
$$(\Pi\alpha^{\mathfrak{v}_1}.\kappa_1^j) \sqcup^{j+1} (\Pi\alpha^{\mathfrak{v}_2}.\kappa_2^j) = \Pi\alpha^{\mathfrak{v}_1 \sqcup \mathfrak{v}_2}.(\kappa_1^j \sqcup^j \kappa_2^j)$$

$$\bot^0 = \mathsf{U}$$
$$\bot^{j+1} = \Pi\alpha^{\phi}.\bot^j.$$

I omit the arity superscript j when it is clear from context.

Proof. The previous definition imposes a proof obligation that the defined structures are in fact bounded join semilattices. By induction on j:

Case 0**.**

When Γ is empty, qualifier expressions ξ form the free bounded semilattice over uninterpreted type variables ($\langle\alpha\rangle$), with join ($\sqcup$) and bottom U all as given in the syntax of qualifier constants and expressions. When Γ is non-empty, quotient the semilattice as follows: for each type variable $\alpha{:}\mathsf{U} \in \Gamma$, add the constraint that $\langle\alpha\rangle = \mathsf{U}$.

Case $i+1$**.**

The semilattice on κ^{i+1} is isomorphic to the product semilattice on $\mathfrak{v} \times \kappa^i$. This follows from definition A.5 and by inspection of rule KSUB-OPER, noting that the addition of $\alpha{:}\langle\alpha\rangle$ to the context does not constrain $\langle\alpha\rangle$ as in the previous case. □

LEMMA A.6 (Well-formed kind semilattice)**.**

For kinds κ_1 and κ_2 of the same arity j, if $\Gamma \vdash \kappa_1$ and $\Gamma \vdash \kappa_2$ then $\Gamma \vdash \kappa_1 \sqcup \kappa_2$.

Proof. By induction on j:

Case 0**.**

By rule OK-QUAL.

Case $j' + 1$**.**

Then there are some κ_1', κ_2', $\mathfrak{v}_1$, and $\mathfrak{v}_2$ such that

(1) $\kappa_1 = \Pi\alpha^{\mathfrak{v}_1}.\kappa_1'$ and

(2) $\kappa_2 = \Pi\alpha^{\mathfrak{v}_2}.\kappa_2'$.

Then by definition A.5,

(3) $\kappa_1 \sqcup \kappa_2 = \Pi\alpha^{\mathfrak{v}_1 \sqcup \mathfrak{v}_2}.\kappa_1' \sqcup \kappa_2'$.

By inversion of rule OK-OPER,

(4) $\Gamma \vdash \kappa_1'$,

(5) if $\alpha \in \mathrm{FTV}(\kappa_1')$ then $+ \sqsubseteq \mathfrak{v}_1$,

(6) $\Gamma \vdash \kappa_2'$,

(7) if $\alpha \in \mathrm{FTV}(\kappa_2')$ then $+ \sqsubseteq \mathfrak{v}_2$.

By the induction hypothesis,

(8) $\Gamma \vdash \kappa_1' \sqcup \kappa_2'$.

Finally, if $\alpha \in \mathrm{FTV}(\kappa_1' \sqcup \kappa_2')$, it must be in at least κ_1' or κ_2', which means that either $+ \sqsubseteq \mathfrak{v}_1$ or $+ \sqsubseteq \mathfrak{v}_2$, which means that $+ \sqsubseteq \mathfrak{v}_1 \sqcup \mathfrak{v}_2$. □

LEMMA A.7 (Unique kinds and unique variances)**.**

1. *If $\Gamma \vdash \tau : \kappa$ and $\Gamma \vdash \tau : \kappa'$ then $\kappa = \kappa'$.*

2. *If* $\Gamma \vdash \alpha \in \tau \updownarrow \mathfrak{v}$ *and* $\Gamma \vdash \alpha \in \tau \updownarrow \mathfrak{v}'$ *then* $\mathfrak{v} = \mathfrak{v}'$.

Proof. By induction on the structure of τ.

1. For kinding:

Case α**.**

The only applicable rule is K-VAR. By inversion, it must be the case that $\alpha{:}\kappa \in \Gamma$. and $\alpha{:}\kappa' \in \Gamma$. Since contexts do not admit repetition, $\kappa = \kappa'$.

Case $\lambda\alpha.\tau_1$**.**

The only applicable rule is K-ABS. By inversion, there must be some κ_1 and $\mathfrak{v}_1$ where $\kappa = \Pi\alpha^{\mathfrak{v}_1}.\kappa_1$ such that

(1) $\Gamma, \alpha{:}\langle\alpha\rangle \vdash \tau_1 : \kappa_1$ and

(2) $\Gamma, \alpha{:}\langle\alpha\rangle \vdash \alpha \in \tau \updownarrow \mathfrak{v}_1$.

Likewise, there must be some κ'_1 and $\mathfrak{v}'_1$ where $\kappa' = \Pi\alpha^{\mathfrak{v}'_1}.\kappa'_1$ such that

(3) $\Gamma, \alpha{:}\langle\alpha\rangle \vdash \tau_1 : \kappa'_1$ and

(4) $\Gamma, \alpha{:}\langle\alpha\rangle \vdash \alpha \in \tau \updownarrow \mathfrak{v}'_1$.

By the induction hypothesis, $\kappa_1 = \kappa'_1$, and by the induction hypothesis part (2), $\mathfrak{v}_1 \pm \mathfrak{v}'_1$. Therefore, $\kappa = \kappa'$.

Case $\tau_1\,\tau_2$**.**

By inversion of rule K-APP and the induction hypothesis twice, relying on the fact that substitution is a function.

Case $\forall\alpha{:}\kappa_1.\tau_1$**.**

By inversion of rule K-ALL and the induction hypothesis, relying on the fact that substitution is a function.

Case $\tau_1 \overset{\xi}{\multimap} \tau_2$**.**

Then $\kappa = \xi = \kappa'$.

Case χ**.**

The only applicable rule is one of K-SUM, K-PROD, K-UNIT, or K-REF, depending on the form of χ.

2. For variance:

Case β.

If $\alpha = \beta$ then $\mathfrak{v} = \mathfrak{v}' = +$. Otherwise, $\mathfrak{v} = \mathfrak{v}' = \phi$.

Case $\lambda\beta.\tau_1$.

By inversion of rule V-ABS,

(1) $\Gamma, \beta{:}\langle\beta\rangle \vdash \alpha \in \tau_1 \updownarrow \mathfrak{v}$ and

(2) $\Gamma, \beta{:}\langle\beta\rangle \vdash \alpha \in \tau_1 \updownarrow \mathfrak{v}'$.

By the induction hypothesis, $\mathfrak{v} \pm \mathfrak{v}'$.

Case $\tau_1\,\tau_2$.

By inversion of rule V-APP,

(1) $\Gamma \vdash \alpha \in \tau_1 \updownarrow \mathfrak{v}_1$,

(2) $\Gamma \vdash \alpha \in \tau_2 \updownarrow \mathfrak{v}_2$, and

(3) $\Gamma \vdash \tau_1 : \Pi\beta^{\mathfrak{v}_3}.\kappa_3$

where $\mathfrak{v} \pm \mathfrak{v}_1 \sqcup (\mathfrak{v}_2 \cdot \mathfrak{v}_3)$. Likewise by inversion,

(4) $\Gamma \vdash \alpha \in \tau_1 \updownarrow \mathfrak{v}'_1$,

(5) $\Gamma \vdash \alpha \in \tau_2 \updownarrow \mathfrak{v}'_2$, and

(6) $\Gamma \vdash \tau_1 : \Pi\beta^{\mathfrak{v}'_3}.\kappa'_3$

where $\mathfrak{v}' \pm \mathfrak{v}'_1 \sqcup (\mathfrak{v}'_2 \cdot \mathfrak{v}'_3)$.

By the induction hypothesis twice, $\mathfrak{v}_1 \pm \mathfrak{v}'_1$ and $\mathfrak{v}_2 \pm \mathfrak{v}'_2$. By the induction hypothesis at part (1), $\Pi\beta^{\mathfrak{v}_3}.\kappa_3 = \Pi\beta^{\mathfrak{v}'_3}.\kappa'_3$, and thus $\mathfrak{v}_3 \pm \mathfrak{v}'_3$. Therefore, $\mathfrak{v} \pm \mathfrak{v}'$.

Case $\forall\alpha{:}\kappa_1.\tau_1$.

As for $\lambda\alpha.\tau_1$, but with rule V-ALL.

Case $\tau_1 \xrightarrow{\xi}_{\circ} \tau_2$.

As for $\tau_1\,\tau_2$, but with rule V-ARR.

Case χ.

Then $\mathfrak{v} = \mathfrak{v}' = \phi$ by rule V-CON. □

LEMMA A.8 (Unique context bounds)**.**

If $\Gamma \vdash \Sigma \preceq \xi$ *and* $\Gamma \vdash \Sigma \preceq \xi'$ *then* $\xi = \xi'$.

Proof. By induction on the structure of Σ, using lemma A.7 for the rule B-ConsX case. □

LEMMA A.9 (Context bounding)**.**

1. *If* $\Gamma \vdash \Sigma_1 \preceq \xi_1$ *and* $\Gamma \vdash \Sigma_2 \preceq \xi_2$ *then* $\Gamma \vdash \Sigma_1, \Sigma_2 \preceq \xi_1 \sqcup \xi_2$.
2. *If* $\Gamma \vdash \Sigma \preceq \xi$ *and* $\Gamma \vdash \tau : \xi'$ *where* $x{:}\tau \in \Sigma$ *then* $\Gamma \vdash \xi' <: \xi$.

Proof.

1. By induction on the structure of Σ_2.
2. By induction on the structure of Σ. □

THEOREM 5.3 (Principal qualifiers, restated from p. 118)**.**

If $\Gamma; \Sigma \rhd \lambda x{:}\tau.\, e : \tau_1 \xrightarrow{\xi}\circ\, \tau_2$*, then it has a least qualifier expression* ξ_0*; that is,*

- $\Gamma; \Sigma \rhd \lambda x{:}\tau.\, e : \tau_1 \xrightarrow{\xi_0}\circ\, \tau_2$ *and*
- $\Gamma \vdash \xi_0 <: \xi'$ *for all* ξ' *such that* $\Gamma; \Sigma \rhd \lambda x{:}\tau.\, e : \tau_1 \xrightarrow{\xi'}\circ\, \tau_2$.

Proof. Please see p. 272. ▷

A.3 Type Soundness

A.3.1 Type Substitutions

DEFINITION A.10 (Type substitution)**.**

I define ***type substitution*** *on a variety of syntactic classes—types* ($\{\tau/\alpha\}\tau'$)*, terms* ($\{\tau/\alpha\}e$)*, and contexts* ($\{\tau/\alpha\}\Gamma$)*—in the standard homomorphic, binding-respecting way, but only when* τ *is closed. To define type substitution on types*

requires defining type substitution on function types ($\{\tau/\alpha\}(\tau_1 \xrightarrow{\xi} \tau_2)$), which requires defining type substitution on kinds: $\{\tau/\alpha\}\kappa$.

For such a substitution to be defined, τ must be closed and well-formed; in particular, $\bullet \vdash \tau : \kappa'$ for some kind κ' (uniquely determined by lemma A.7). If κ' is a dependent product kind, then define $\{\tau/\alpha\}\kappa = \kappa$. On the other hand, if κ' is a qualifier expression ξ', then define $\{\tau/\alpha\}\kappa = \{\xi'/\alpha\}\kappa$.

LEMMA A.11 (Type substitution on kind well-formedness)**.**

Suppose that $\Gamma \vdash \tau : \kappa'$. Then:

1. *If $\Gamma, \alpha{:}\kappa' \vdash \kappa$ then $\Gamma \vdash \{\tau/\alpha\}\kappa$.*
2. *If $\Gamma \vdash \kappa$ then $\{\tau/\alpha\}\Gamma \vdash \kappa$.*
3. *If $\vdash \Gamma$ then $\vdash \{\tau/\alpha\}\Gamma$.*

Proof.

1. By induction on κ:

   Case $\mathfrak{q}$**.**

   By rule OK-QUAL.

   Case $\xi_1 \sqcup \xi_2$ **or** $\xi_1 \sqcap \xi_2$**.**

   By the induction hypothesis, twice, and rule OK-JOIN or rule OK-MEET, respectively.

   Case β**.**

   If $\alpha = \beta$, then by inversion of rule OK-VAR, $\kappa' = \xi$ for some ξ. So $\{\tau/\alpha\}\kappa = \xi$. Then because $\Gamma \vdash \tau : \xi$ and by lemma A.4, $\Gamma \vdash \xi$.

   If $\alpha \neq \beta$, then $\{\tau/\alpha\}\langle\beta\rangle = \langle\beta\rangle$. By inversion of rule OK-VAR, we know that $\beta{:}\xi \in \Gamma$ for some ξ. Then by rule OK-VAR.

   Case $\Pi\beta^{\upsilon}.\kappa''$**.**

   By inversion of rule OK-OPER and the induction hypothesis,

   (1) $\Gamma \vdash \{\tau/\alpha\}\kappa''$.

Note that because β is bound with only κ'' in its scope, we know that $\beta \notin \mathrm{FTV}(\tau)$, so $\beta \in \mathrm{FTV}(\kappa'')$ if and only if $\beta \in \mathrm{FTV}(\{\tau/\alpha\}\kappa'')$. Thus, the premise of rule OK-OPER that $\beta \in \mathrm{FTV}(\{\tau/\alpha\}\kappa'')$ implies that $+ \sqsubseteq \mathfrak{v}$ remains satisfied.

2–3. By mutual induction on the derivations. Note that the judgment $\Gamma \vdash \kappa$ only looks at Γ to make sure that it maps the free type variables of κ to well-formed qualifiers, and substitution preserves well-formed qualifiers by the previous part. □

LEMMA A.12 (Qualifier substitution on qualifier subsumption)**.**

If $\Gamma, \alpha{:}\xi \models \xi_1 \sqsubseteq \xi_2$ *then* $\{\xi/\alpha\}\Gamma \models \{\xi/\alpha\}\xi_1 \sqsubseteq \{\xi/\alpha\}\xi_2$.

Proof. Let $\mathcal{V}$ be any valuation consistent with $\{\xi/\alpha\}\Gamma$. That means that for all $\beta{:}\xi'' \in \{\xi/\alpha\}\Gamma$, $\mathcal{V}(\beta) \sqsubseteq \mathcal{V}(\xi'')$. Equivalently, for all $\beta{:}\xi' \in \Gamma$, we know that $\mathcal{V}(\beta) \sqsubseteq \mathcal{V}(\{\xi/\alpha\}\xi')$. (Note that $\beta \neq \alpha$ or else $\Gamma, \alpha{:}\xi$ would be ill formed.)

Now let $\mathcal{V}' = \mathcal{V}\{\alpha \mapsto \mathcal{V}(\xi)\}$. Note that $\mathcal{V}'(\beta) = \mathcal{V}(\beta)$ and $\mathcal{V}'(\xi') = \mathcal{V}(\{\xi/\alpha\}\xi')$. Then $\mathcal{V}'(\beta) \sqsubseteq \mathcal{V}'(\xi')$ for all $\beta{:}\xi' \in \Gamma$. Furthermore, we defined $\mathcal{V}'$ so that $\mathcal{V}'(\alpha) = \mathcal{V}'(\xi)$, so $\mathcal{V}'(\alpha) \sqsubseteq \mathcal{V}'(\xi)$. Thus, $\mathcal{V}'$ is consistent with $\Gamma, \alpha{:}\xi$.

By definition 5.2, since $\Gamma, \alpha{:}\xi \models \xi_1 \sqsubseteq \xi_2$, we now know that $\mathcal{V}'(\xi_1) \sqsubseteq \mathcal{V}'(\xi_2)$. Note that $\mathcal{V}'(\xi_1) = \mathcal{V}(\{\xi/\alpha\}\xi_1)$ and $\mathcal{V}'(\xi_2) = \mathcal{V}(\{\xi/\alpha\}\xi_2)$. Then we know that $\mathcal{V}(\{\xi/\alpha\}\xi_1) \sqsubseteq \mathcal{V}(\{\xi/\alpha\}\xi_2)$. Since $\mathcal{V}$ is an arbitrary valuation consistent with $\{\xi/\alpha\}\Gamma$, this means that $\{\xi/\alpha\}\Gamma \models \{\xi/\alpha\}\xi_1 \sqsubseteq \{\xi/\alpha\}\xi_2$. □

COROLLARY A.13 (Type substitution on subkinding)**.**

If $\Gamma, \alpha{:}\kappa \vdash \kappa_1 <: \kappa_2$ *and* $\bullet \vdash \tau : \kappa$ *then* $\{\tau/\alpha\}\Gamma \vdash \{\tau/\alpha\}\kappa_1 <: \{\tau/\alpha\}\kappa_2$.

Proof. By cases on the subkinding derivation:

Case $\dfrac{\mathfrak{v}_1 \sqsubseteq \mathfrak{v}_2 \qquad \Gamma, \beta{:}\langle\beta\rangle, \alpha{:}\kappa \vdash \kappa_1' <: \kappa_2'}{\Gamma, \alpha{:}\kappa \vdash \Pi\beta^{\mathfrak{v}_1}.\kappa_1' <: \Pi\beta^{\mathfrak{v}_2}.\kappa_2'}$.

By the induction hypothesis and rule KSUB-OPER.

Case $\dfrac{\Gamma, \alpha{:}\kappa \models \xi_1 \sqsubseteq \xi_2 \qquad \Gamma, \alpha{:}\kappa \vdash \xi_1 \qquad \Gamma, \alpha{:}\kappa \vdash \xi_2}{\Gamma, \alpha{:}\kappa \vdash \xi_1 <: \xi_2}$.

By lemma A.11, $\{\tau/\alpha\}\Gamma \vdash \{\tau/\alpha\}\xi_1$ and $\{\tau/\alpha\}\Gamma \vdash \{\tau/\alpha\}\xi_2$.

We need to show that $\{\tau/\alpha\}\Gamma \models \{\tau/\alpha\}\xi_1 \sqsubseteq \{\tau/\alpha\}\xi_2$. If κ (the kind of τ) is not a base kind, then this is trivially the case. Otherwise, κ is a qualifier ξ such that $\Gamma, \alpha{:}\xi \models \xi_1 \sqsubseteq \xi_2$. Then by lemma A.12, $\{\xi/\alpha\}\Gamma \models \{\xi/\alpha\}\xi_1 \sqsubseteq \{\xi/\alpha\}\xi_2$. □

LEMMA A.14 (Non-free type variables do not vary)**.**

If $\alpha \notin \mathrm{FTV}(\tau)$ *then* $\Gamma \vdash \alpha \in \tau \updownarrow \phi$.

Proof. By induction on the structure of τ. □

LEMMA A.15 (Type substitution on kinding and variance)**.**

For any type τ *and kind* κ *such that* $\bullet \vdash \tau : \kappa$*,*

1. *If* $\Gamma, \alpha{:}\kappa \vdash \tau' : \kappa'$ *then* $\{\tau/\alpha\}\Gamma \vdash \{\tau/\alpha\}\tau' : \{\tau/\alpha\}\kappa'$.
2. *If* $\Gamma, \beta{:}\langle\beta\rangle, \alpha{:}\kappa \vdash \beta \in \tau' \updownarrow \mathfrak{v}$*, where* $\beta \notin \mathrm{FTV}(\tau)$ *and* $\beta \notin \mathrm{FTV}(\Gamma, \alpha{:}\kappa)$*, then* $\{\tau/\alpha\}\Gamma, \beta{:}\langle\beta\rangle \vdash \beta \in \{\tau/\alpha\}\tau' \updownarrow \mathfrak{v}$.

Proof. In each case, because $\Gamma, \alpha{:}\kappa$ is well formed (by lemma A.4), we know that $\alpha \notin \mathrm{dom}\,\Gamma$. Since $\vdash \Gamma$, we know that $\alpha \notin \mathrm{FTV}(\Gamma)$, and thus $\{\tau/\alpha\}\Gamma = \Gamma$. Note also that $\bullet \vdash \kappa$ by the same lemma, which means that κ is closed.

We then proceed by mutual induction on the kinding and variance derivations.

1. For kinding, there is one interesting case:

   Case $\dfrac{\alpha'{:}\kappa' \in \Gamma, \alpha{:}\kappa \qquad \vdash \Gamma, \alpha{:}\kappa}{\Gamma, \alpha{:}\kappa \vdash \alpha' : \kappa'}$.

   If $\alpha = \alpha'$ then $\kappa = \kappa'$ and $\{\tau/\alpha\}\alpha' = \tau$. We know that κ is closed, so $\{\tau/\alpha\}\kappa = \kappa$. Because $\bullet \vdash \tau : \kappa$, and by weakening, $\Gamma \vdash \tau : \kappa$.

If $\alpha \neq \alpha'$ then $\{\tau/\alpha\}\alpha' = \alpha'$. Furthermore, we know that $\alpha' {:} \kappa' \in \Gamma$, and because $\alpha \notin \text{FTV}(\Gamma)$, we know that $\{\tau/\alpha\}\kappa' = \kappa'$. Then by rule WF-CONSA or WF-CONSAREC.

2. For variance, the two variable cases are non-trivial:

Case $\dfrac{\Gamma \vdash \beta : \kappa}{\Gamma, \beta{:}\langle\beta\rangle, \alpha{:}\kappa \vdash \beta \in \beta \updownarrow +}$.

Since $\alpha \neq \beta$, $\{\tau/\alpha\}\beta = \beta$, so $\{\tau/\alpha\}\Gamma, \beta{:}\langle\beta\rangle \vdash \beta \in \beta \updownarrow +$.

Case $\dfrac{\Gamma \vdash \beta' : \kappa}{\Gamma, \beta{:}\langle\beta\rangle, \alpha{:}\kappa \vdash \beta \in \beta' \updownarrow \phi}$.

If $\beta' \neq \alpha$ then $\{\tau/\alpha\}\beta' = \beta'$, so $\{\tau/\alpha\}\Gamma, \beta{:}\langle\beta\rangle \vdash \beta \in \{\tau/\alpha\}\beta' \updownarrow \phi$.

If $\beta' = \alpha$, then note that $\beta \notin \text{FTV}(\tau)$. Thus by lemma A.14, we know that $\{\tau/\alpha\}\Gamma, \beta{:}\langle\beta\rangle \vdash \beta \in \tau \updownarrow \phi$. □

LEMMA A.16 (Type substitution on type equivalence)**.**

If $\tau_1 \equiv \tau_2$ *then* $\{\tau/\alpha\}\tau_1 \equiv \{\tau/\alpha\}\tau_2$.

Proof. By induction on the type equivalence derivation $\tau_1 \equiv \tau_2$. There is only one non-trivial case:

Case $(\lambda\beta.\tau_1')\tau_2' \equiv \{\tau_2'/\beta\}\tau_1'$**.**

Then

(1) $\{\tau/\alpha\}\tau_1 = (\lambda\beta.\{\tau/\alpha\}\tau_1')\{\tau/\alpha\}\tau_2'$ and

(2) $\{\tau/\alpha\}\tau_2 = \{\{\tau/\alpha\}\tau_2'/\beta\}\{\tau/\alpha\}\tau_1'$,

and finally

(3) $(\lambda\beta.\{\tau/\alpha\}\tau_1')\{\tau/\alpha\}\tau_2' \equiv \{\{\tau/\alpha\}\tau_2'/\beta\}\{\tau/\alpha\}\tau_1'$.

by rule E-BETA. □

LEMMA A.17 (Type substitution on subtyping)**.**

If $\Gamma, \alpha{:}\kappa \vdash \tau_1 <:^{\mathfrak{v}} \tau_2$ *and* $\bullet \vdash \tau : \kappa$ *then* $\{\tau/\alpha\}\Gamma \vdash \{\tau/\alpha\}\tau_1 <:^{\mathfrak{v}} \{\tau/\alpha\}\tau_2$.

Proof. By a simple induction on the derivation of $\Gamma, \alpha{:}\kappa \vdash \tau_1 <:^{\mathfrak{v}} \tau_2$, using corollary A.13, lemma A.15, and lemma A.16. □

LEMMA A.18 (Type substitution on context bounding)**.**

If $\Gamma, \alpha{:}\kappa \vdash \Sigma \preceq \xi$ *and* $\bullet \vdash \tau : \kappa$ *then* $\{\tau/\alpha\}\Gamma \vdash \{\tau/\alpha\}\Sigma \preceq \{\tau/\alpha\}\xi$.

Proof. Straightforward induction on the derivation of $\Gamma, \alpha{:}\kappa \vdash \Sigma \preceq \xi$, using lemma A.15. □

LEMMA A.19 (Type substitution on context well-formedness)**.**

If $\vdash \Gamma, \alpha{:}\kappa; \Sigma$ *and* $\bullet \vdash \tau : \kappa$ *then* $\vdash \{\tau/\alpha\}\Gamma; \{\tau/\alpha\}\Sigma$.

Proof. By inversion of rule WF, lemma A.18 twice, and rule WF. □

LEMMA A.20 (Type substitution on context extension)**.**

If $\vdash (\Gamma_0, \alpha{:}\kappa; \Sigma_0), \Sigma' \rightsquigarrow \Gamma_1, \alpha{:}\kappa; \Sigma_1$ *and* $\bullet \vdash \tau : \kappa$ *then*

$$\vdash (\{\tau/\alpha\}\Gamma_0; \{\tau/\alpha\}\Sigma_0), \{\tau/\alpha\}\Sigma' \rightsquigarrow \{\tau/\alpha\}\Gamma_1; \{\tau/\alpha\}\Sigma_1.$$

Proof. Simple induction on the derivation of $\vdash (\Gamma, \alpha{:}\kappa; \Sigma), \Sigma_1 \rightsquigarrow \Gamma', \alpha{:}\kappa; \Sigma'$ using lemma A.15. □

LEMMA A.21 (Type substitution on typing)**.**

If $\Gamma, \alpha{:}\kappa; \Sigma \rhd e : \tau$ *and* $\bullet \vdash \tau' : \kappa$ *then* $\{\tau'/\alpha\}\Gamma; \{\tau'/\alpha\}\Sigma \rhd \{\tau'/\alpha\}e : \{\tau'/\alpha\}\tau$.

Proof. By induction on the *height* of the typing derivation:

Case $\dfrac{\Gamma, \alpha{:}\kappa; \Sigma \rhd e : \tau'' \qquad \Gamma, \alpha{:}\kappa \vdash \tau'' <:^{+} \tau \qquad \Gamma, \alpha{:}\kappa \vdash \tau : \xi}{\Gamma, \alpha{:}\kappa; \Sigma \rhd e : \tau}$.

By the induction hypothesis,

(1) $\{\tau'/\alpha\}\Gamma; \{\tau'/\alpha\}\Sigma \rhd \{\tau'/\alpha\}e : \{\tau'/\alpha\}\tau''$.

By lemma A.17 and lemma A.15,

(2) $\{\tau'/\alpha\}\Gamma \vdash \{\tau'/\alpha\}\tau'' <:^{+} \{\tau'/\alpha\}\tau$ and

(3) $\{\tau'/\alpha\}\Gamma \vdash \{\tau'/\alpha\}\tau : \{\tau'/\alpha\}\xi$.

Then by rule T-SUBSUME,

(4) $\{\tau'/\alpha\}\Gamma; \{\tau'/\alpha\}\Sigma \rhd \{\tau'/\alpha\}e : \{\tau'/\alpha\}\tau$.

Case $\dfrac{\Gamma_1'; \Sigma_1 \rhd e : \tau \qquad \vdash \Gamma_1, \Gamma_2, \alpha{:}\kappa; \Sigma_1, \Sigma_2}{\Gamma_1, \Gamma_2, \alpha{:}\kappa; \Sigma_1, \Sigma_2 \rhd e : \tau}$.

There are two possibilities, depending on whether $\alpha \in \operatorname{dom}\Gamma_1'$:

Case $\Gamma_1' = \Gamma_1$.

Then by weakening, $\Gamma_1, \alpha{:}\kappa; \Sigma_1 \rhd e : \tau$. This derivation establishing this judgment has the same height as the one for $\Gamma_1; \Sigma_1 \rhd e : \tau$, so we can apply the induction hypothesis as in the next case.

Case $\Gamma_1' = \Gamma_1, \alpha{:}\kappa$.

Then by the induction hypothesis,

(1) $\{\tau'/\alpha\}\Gamma_1; \{\tau'/\alpha\}\Sigma_1 \rhd \{\tau'/\alpha\}e : \{\tau'/\alpha\}\tau$,

by lemma A.19,

(2) $\vdash \{\tau'/\alpha\}(\Gamma_1, \Gamma_2); \{\tau'/\alpha\}(\Sigma_1, \Sigma_2)$,

and by weakening,

(3) $\{\tau'/\alpha\}(\Gamma_1, \Gamma_2); \{\tau'/\alpha\}(\Sigma_1, \Sigma_2) \rhd \{\tau'/\alpha\}e : \{\tau'/\alpha\}\tau$.

Case $\dfrac{x{:}\tau \in \Gamma, \alpha{:}\kappa, \Sigma \qquad \Gamma \vdash \tau : \xi \qquad \vdash \Gamma, \alpha{:}\kappa; \Sigma}{\Gamma, \alpha{:}\kappa; \Sigma \rhd x : \tau}$.

Then $x{:}\tau \in \Gamma, \Sigma$, and thus

(1) $(x{:}\{\tau'/\alpha\}\tau) \in \{\tau'/\alpha\}\Gamma, \{\tau'/\alpha\}\Sigma$.

By lemma A.15 and lemma A.19,

(2) $\{\tau'/\alpha\}\Gamma \vdash \{\tau'/\alpha\}\tau : \{\tau'/\alpha\}\xi$ and

(3) $\vdash \{\tau'/\alpha\}\Gamma; \{\tau'/\alpha\}\Sigma$.

Note that $\{\tau'/\alpha\}x = x$. Then by rule T-VAR,

(4) $\{\tau'/\alpha\}\Gamma; \{\tau'/\alpha\}\Sigma \rhd x : \{\tau'/\alpha\}\tau$.

Case $\dfrac{\ell{:}\tau_1 \in \Sigma \qquad \bullet \vdash \tau_1 : \xi \qquad \vdash \Gamma, \alpha{:}\kappa; \Sigma}{\Gamma, \alpha{:}\kappa; \Sigma \rhd \ell : \mathsf{ref}\ \tau_1}$.

Since τ_1 types in the empty context, α is not free in τ_1, so $\{\tau'/\alpha\}\tau_1 = \tau_1$. Furthermore,

(1) $\{\tau'/\alpha\}\ell = \ell$ and

(2) $\ell{:}\tau_1 \in \{\tau'/\alpha\}\Sigma$.

By lemma A.19,

(3) $\vdash \{\tau'/\alpha\}\Gamma; \{\tau'/\alpha\}\Sigma$.

Then by rule T-PTR,

(4) $\{\tau'/\alpha\}\Gamma; \{\tau'/\alpha\}\Sigma \rhd \{\tau'/\alpha\}\ell : \{\tau'/\alpha\}(\mathsf{ref}\ \tau_1)$.

Case $\dfrac{\begin{array}{c}\vdash (\Gamma, \alpha{:}\kappa; \Sigma), x{:}\tau_1 \rightsquigarrow \Gamma'; \Sigma' \\ \Gamma'; \Sigma' \rhd e_2 : \tau_2 \qquad \Gamma, \alpha{:}\kappa \vdash \Sigma \preceq \xi \qquad \Gamma, \alpha{:}\kappa \vdash \tau_1 : \xi_1\end{array}}{\Gamma, \alpha{:}\kappa; \Sigma \rhd \lambda x{:}\tau_1.\, e_2 : \tau_1 \xrightarrow{\xi} \tau_2}$.

Let $\Gamma'', \alpha{:}\kappa = \Gamma'$, and note that $\vdash (\Gamma; \Sigma), x{:}\tau_1 \rightsquigarrow \Gamma''; \Sigma'$. By the induction hypothesis,

(1) $\{\tau'/\alpha\}\Gamma''; \{\tau'/\alpha\}\Sigma' \rhd \{\tau'/\alpha\}e_2 : \{\tau'/\alpha\}\tau_2$.

By lemma A.20, lemma A.18, and lemma A.15,

(2) $\vdash (\{\tau'/\alpha\}\Gamma; \{\tau'/\alpha\}\Sigma), x{:}\{\tau'/\alpha\}\tau_1 \rightsquigarrow \{\tau'/\alpha\}\Gamma''; \{\tau'/\alpha\}\Sigma'$,

(3) $\{\tau'/\alpha\}\Gamma \vdash \{\tau'/\alpha\}\Sigma \preceq \{\tau'/\alpha\}\xi$, and

(4) $\{\tau'/\alpha\}\Gamma \vdash \{\tau'/\alpha\}\tau_1 : \{\tau'/\alpha\}\xi_1$.

Then by rule T-ABS,

(5) $\{\tau'/\alpha\}\Gamma; \{\tau'/\alpha\}\Sigma \rhd \lambda x{:}\{\tau'/\alpha\}\tau_1.\, \{\tau'/\alpha\}e_2 : \{\tau'/\alpha\}(\tau_1 \xrightarrow{\xi}_\circ \tau_2)$.

Case $\dfrac{\Gamma, \alpha{:}\kappa; \Sigma_1 \rhd e_1 : \tau_1 \xrightarrow{\xi}_\circ \tau_2 \qquad \Gamma, \alpha{:}\kappa; \Sigma_2 \rhd e_2 : \tau_1}{\Gamma, \alpha{:}\kappa; \Sigma_1, \Sigma_2 \rhd e_1\, e_2 : \tau_2}$.

By the induction hypothesis twice,

(1) $\{\tau'/\alpha\}\Gamma; \{\tau'/\alpha\}\Sigma_1 \rhd \{\tau'/\alpha\}e_1 : \{\tau'/\alpha\}\tau_1 \xrightarrow{\{\tau'/\alpha\}\xi}_\circ \{\tau'/\alpha\}\tau_2$ and

(2) $\{\tau'/\alpha\}\Gamma; \{\tau'/\alpha\}\Sigma_2 \rhd \{\tau'/\alpha\}e_2 : \{\tau'/\alpha\}\tau_1$.

Then by rule T-APP,

(3) $\{\tau'/\alpha\}\Gamma; \{\tau'/\alpha\}(\Sigma_1, \Sigma_2) \rhd \{\tau'/\alpha\}(e_1\, e_2) : \{\tau'/\alpha\}\tau_2$.

Case $\dfrac{\Gamma, \alpha{:}\kappa, \alpha_1{:}\kappa_1; \Sigma \rhd e_1 : \tau_1 \qquad \Gamma, \alpha{:}\kappa \vdash \kappa_1}{\Gamma, \alpha{:}\kappa; \Sigma \rhd \Lambda\alpha_1{:}\kappa_1.\, v_1 : \forall \alpha_1{:}\kappa_1.\, \tau_1}$.

By lemma A.11 and the induction hypothesis,

(1) $\{\tau'/\alpha\}\Gamma \vdash \{\tau'/\alpha\}\kappa_1$ and

(2) $\{\tau'/\alpha\}\Gamma, \alpha_1{:}\{\tau'/\alpha\}\kappa_1; \{\tau'/\alpha\}\Sigma \rhd \{\tau'/\alpha\}v_1 : \{\tau'/\alpha\}\tau_1$,

and by rule T-TABS,

(3) $\{\tau'/\alpha\}\Gamma; \{\tau'/\alpha\}\Sigma \rhd \{\tau'/\alpha\}(\Lambda\alpha_1{:}\kappa_1.\, v_1) : \{\tau'/\alpha\}(\forall \alpha_1{:}\kappa_1.\, \tau_1)$.

Case $\dfrac{\Gamma, \alpha{:}\kappa; \Sigma \rhd e_1 : \forall \alpha_1{:}\kappa_1.\, \tau_2 \qquad \Gamma, \alpha{:}\kappa \vdash \tau_1 : \kappa_1}{\Gamma, \alpha{:}\kappa; \Sigma \rhd e_1\, \tau_1 : \{\tau_1/\alpha_1\}\tau_2}$.

By the induction hypothesis,

(1) $\{\tau'/\alpha\}\Gamma; \{\tau'/\alpha\}\Sigma \rhd \{\tau'/\alpha\}e_1 : \forall \alpha_1{:}\{\tau'/\alpha\}\kappa_1.\, \{\tau'/\alpha\}\tau_2$,

and by lemma A.15,

(2) $\{\tau'/\alpha\}\Gamma \vdash \{\tau'/\alpha\}\tau_1 : \{\tau'/\alpha\}\kappa_1$.

Note that $\{\{\tau'/\alpha\}\tau_1/\alpha_1\}\{\tau'/\alpha\}\tau_2 = \{\tau'/\alpha\}\{\tau_1/\alpha_1\}\tau_2$. Then by rule T-TAPP,

(3) $\{\tau'/\alpha\}\Gamma;\{\tau'/\alpha\}\Sigma \rhd \{\tau'/\alpha\}(e_1\,\tau_1) : \{\tau'/\alpha\}\{\tau_1/\alpha_1\}\tau_2$.

Case $\dfrac{\Gamma,\alpha{:}\kappa;\Sigma \rhd e_1 : \tau \xrightarrow{\mathsf{U}}_{\circ} \tau}{\Gamma,\alpha{:}\kappa;\Sigma \rhd \mathsf{fix}\, e_1 : \tau}$.

By the induction hypothesis,

(1) $\{\tau'/\alpha\}\Gamma;\{\tau'/\alpha\}\Sigma \rhd \{\tau'/\alpha\}e_1 : \{\tau'/\alpha\}\tau \xrightarrow{\mathsf{U}}_{\circ} \{\tau'/\alpha\}\tau$.

Then by rule T-FIX,

(2) $\{\tau'/\alpha\}\Gamma;\{\tau'/\alpha\}\Sigma \rhd \{\tau'/\alpha\}(\mathsf{fix}\, e_1) : \{\tau'/\alpha\}\tau$.

Case $\dfrac{\vdash \Gamma,\alpha{:}\kappa;\Sigma}{\Gamma,\alpha{:}\kappa;\Sigma \rhd \langle\rangle : 1}$.

By lemma A.19,

(1) $\vdash \{\tau'/\alpha\}\Gamma;\{\tau'/\alpha\}\Sigma$,

and by rule T-UNIT, $\{\tau'/\alpha\}\Gamma;\{\tau'/\alpha\}\Sigma \rhd \langle\rangle : 1$.

Case $\dfrac{\Gamma,\alpha{:}\kappa;\Sigma \rhd e_1 : \tau_1 \qquad \Gamma,\alpha{:}\kappa \vdash \tau_2 : \xi}{\Gamma,\alpha{:}\kappa;\Sigma \rhd \mathsf{inl}\, e_1 : \tau_1 \oplus \tau_2}$.

By the induction hypothesis,

(1) $\{\tau'/\alpha\}\Gamma;\{\tau'/\alpha\}\Sigma \rhd \{\tau'/\alpha\}e_1 : \{\tau'/\alpha\}\tau_1$,

and by lemma A.15,

(2) $\{\tau'/\alpha\}\Gamma \vdash \{\tau'/\alpha\}\tau_2 : \{\tau'/\alpha\}\xi$.

By rule T-INL,

(3) $\{\tau'/\alpha\}\Gamma;\{\tau'/\alpha\}\Sigma \rhd \{\tau'/\alpha\}(\mathsf{inl}\, e_1) : \{\tau'/\alpha\}(\tau_1 \oplus \tau_2)$.

Case $\dfrac{\Gamma, \alpha{:}\kappa;\Sigma \rhd e_2 : \tau_2 \qquad \Gamma, \alpha{:}\kappa \vdash \tau_1 : \xi}{\Gamma, \alpha{:}\kappa;\Sigma \rhd \mathsf{inr}\, e_2 : \tau_1 \oplus \tau_2}$.

As in the previous case.

Case $\dfrac{\begin{array}{c}\Gamma, \alpha{:}\kappa;\Sigma_1 \rhd e' : \tau_1 \oplus \tau_2 \\ \vdash (\Gamma, \alpha{:}\kappa;\Sigma_2), x_1{:}\tau_1 \rightsquigarrow \Gamma_1;\Sigma_{21} \qquad \Gamma_1;\Sigma_{21} \rhd e_1 : \tau \\ \vdash (\Gamma, \alpha{:}\kappa;\Sigma_2), x_2{:}\tau_2 \rightsquigarrow \Gamma_2;\Sigma_{22} \qquad \Gamma_2;\Sigma_{22} \rhd e_2 : \tau\end{array}}{\Gamma, \alpha{:}\kappa;\Sigma_1, \Sigma_2 \rhd \mathsf{case}\, e' \,\mathsf{of}\, \mathsf{inl}\, x_1 \to e_2; \mathsf{inr}\, x_2 \to e_2 : \tau}$.

Let $\Gamma_1', \alpha{:}\kappa = \Gamma_1$ and $\Gamma_2', \alpha{:}\kappa = \Gamma_2$. Then by the induction hypothesis,

(1) $\{\tau'/\alpha\}\Gamma;\{\tau'/\alpha\}\Sigma_1 \rhd \{\tau'/\alpha\}e' : \{\tau'/\alpha\}\tau_1 \oplus \{\tau'/\alpha\}\tau_2$,

(2) $\{\tau'/\alpha\}\Gamma_1';\{\tau'/\alpha\}\Sigma_{21} \rhd \{\tau'/\alpha\}e_1 : \{\tau'/\alpha\}\tau$, and

(3) $\{\tau'/\alpha\}\Gamma_2';\{\tau'/\alpha\}\Sigma_{22} \rhd \{\tau'/\alpha\}e_2 : \{\tau'/\alpha\}\tau$.

By lemma A.20,

(4) $\vdash (\{\tau'/\alpha\}\Gamma;\{\tau'/\alpha\}\Sigma_2), x_1{:}\{\tau'/\alpha\}\tau_1 \rightsquigarrow \{\tau'/\alpha\}\Gamma_1';\{\tau'/\alpha\}\Sigma_{21}$ and

(5) $\vdash (\{\tau'/\alpha\}\Gamma;\{\tau'/\alpha\}\Sigma_2), x_2{:}\{\tau'/\alpha\}\tau_2 \rightsquigarrow \{\tau'/\alpha\}\Gamma_1';\{\tau'/\alpha\}\Sigma_{22}$.

Then by rule T-CASE,

(6) $\{\tau'/\alpha\}\Gamma;\{\tau'/\alpha\}(\Sigma_1, \Sigma_2) \rhd \{\tau'/\alpha\}(\mathsf{case}\, e' \,\mathsf{of}\, \mathsf{inl}\, x_1 \to e_2; \mathsf{inr}\, x_2 \to e_2) : \{\tau'/\alpha\}\tau$.

Case $\dfrac{\Gamma, \alpha{:}\kappa;\Sigma_1 \rhd v_1 : \tau_1 \qquad \Gamma, \alpha{:}\kappa;\Sigma_2 \rhd v_2 : \tau_2}{\Gamma, \alpha{:}\kappa;\Sigma_1, \Sigma_2 \rhd \langle v_1, v_2\rangle : \tau_1 \otimes \tau_2}$.

By the induction hypothesis twice and rule T-PAIR.

$$\textbf{Case}\ \frac{\begin{array}{c}\Gamma, \alpha{:}\kappa; \Sigma_1 \rhd e' : \tau_1 \otimes \tau_2 \\ \vdash (\Gamma, \alpha{:}\kappa; \Sigma_2), x_1{:}\tau_1, x_2{:}\tau_2 \rightsquigarrow \Gamma'; \Sigma' \qquad \Gamma'; \Sigma' \rhd e_1 : \tau\end{array}}{\Gamma, \alpha{:}\kappa; \Sigma_1, \Sigma_2 \rhd \mathsf{let}\ \langle x_1, x_2\rangle = e'\ \mathsf{in}\ e_1 : \tau}.$$

Let $\Gamma'', \alpha{:}\kappa = \Gamma''$. Then by the induction hypothesis twice and lemma A.20,

(1) $\{\tau'/\alpha\}\Gamma; \{\tau'/\alpha\}\Sigma_1 \rhd \{\tau'/\alpha\}e' : \{\tau'/\alpha\}\tau_1 \otimes \{\tau'/\alpha\}\tau_2$,

(2) $\{\tau'/\alpha\}\Gamma''; \{\tau'/\alpha\}\Sigma' \rhd \{\tau'/\alpha\}e_1 : \{\tau'/\alpha\}\tau$, and

(3) $\vdash (\{\tau'/\alpha\}\Gamma; \{\tau'/\alpha\}\Sigma_2), x_1{:}\{\tau'/\alpha\}\tau_1, x_2{:}\{\tau'/\alpha\}\tau_2 \rightsquigarrow \{\tau'/\alpha\}\Gamma''; \{\tau'/\alpha\}\Sigma'$

Then by rule T-UNPAIR,

(4) $\{\tau'/\alpha\}\Gamma; \{\tau'/\alpha\}(\Sigma_1, \Sigma_2) \rhd \{\tau'/\alpha\}(\mathsf{let}\ \langle x_1, x_2\rangle = e'\ \mathsf{in}\ e_1) : \{\tau'/\alpha\}\tau$.

$$\textbf{Case}\ \frac{\Gamma, \alpha{:}\kappa; \Sigma \rhd e_1 : \tau_1}{\Gamma, \alpha{:}\kappa; \Sigma \rhd \mathsf{new}\ e_1 : \mathsf{ref}\ \tau_1}.$$

By the induction hypothesis and rule T-NEW.

$$\textbf{Case}\ \frac{\Gamma, \alpha{:}\kappa; \Sigma_1 \rhd e_1 : \mathsf{ref}\ \tau_1 \qquad \Gamma, \alpha{:}\kappa; \Sigma_2 \rhd e_2 : \tau_2}{\Gamma, \alpha{:}\kappa; \Sigma_1, \Sigma_2 \rhd \mathsf{swap}\ e_1\, e_2 : \mathsf{ref}\ \tau_2 \otimes \tau_1}.$$

By the induction hypothesis twice and rule T-SWAP.

$$\textbf{Case}\ \frac{\Gamma, \alpha{:}\kappa; \Sigma \rhd e_1 : \mathsf{ref}\ \tau_1}{\Gamma, \alpha{:}\kappa; \Sigma \rhd \mathsf{delete}\ e_1 : 1}.$$

By the induction hypothesis and rule T-DELETE. □

A.3.1.1 Properties of Contexts

LEMMA A.22 (Contexts close terms)**.**

If $\Gamma; \Sigma \rhd e : \tau$ *and* $x \in \mathrm{FV}(e)$ *then there exists some* τ' *such that* $x{:}\tau' \in \Gamma, \Sigma$.

Proof. By inspection of the typing rules, we see that x can only be typed if it occurs in the context, and furthermore the typing rules must type every subterm of e. □

LEMMA A.23 (Coalescing of context extension)**.**

$\vdash (\Gamma_0;\Sigma_0),\Sigma_1' \rightsquigarrow \Gamma_1;\Sigma_1$ *and* $\vdash (\Gamma_1;\Sigma_1),\Sigma_2' \rightsquigarrow \Gamma_2;\Sigma_2$ *iff* $\vdash (\Gamma_0;\Sigma_0),\Sigma_1',\Sigma_2' \rightsquigarrow \Gamma_2;\Sigma_2$.

Proof. By induction on the structure of Σ_1':

Case $\bullet$.

Then

(1) $\vdash (\Gamma_0;\Sigma_0),\bullet \rightsquigarrow \Gamma_0;\Sigma_0$,

(2) $\vdash (\Gamma_0;\Sigma_0),\Sigma_2' \rightsquigarrow \Gamma_2;\Sigma_2$, and

(3) $\vdash (\Gamma_0;\Sigma_0),\bullet,\Sigma_2' \rightsquigarrow \Gamma_2;\Sigma_2$.

Case $\alpha{:}\kappa,\Sigma_1''$.

There is no rule for adding $\alpha{:}\kappa$, so both sides of the bi-implication are false.

Case $\ell{:}\tau,\Sigma_1''$.

There is no rule for adding $\ell{:}\tau$, so both sides of the bi-implication are false.

Case $x{:}\tau,\Sigma_1''$.

Note that Γ_0, Γ_1 and Γ_2 all support the same type- and kind-level judgments, by observation A.1.

If there is no ξ such that $\Gamma_0 \vdash \tau : \xi$, there is no rule for adding $x{:}\tau$, so both sides of the bi-implication are false.

If there exists some ξ such that $\Gamma_0 \vdash \tau : \xi$, then by inversion of rule X-CONSA twice, it suffices to show that $\vdash (\Gamma_0;\Sigma_0,x{:}\tau),\Sigma_1'' \rightsquigarrow \Gamma_1;\Sigma_1$ and $\vdash (\Gamma_1;\Sigma_1),\Sigma_2' \rightsquigarrow \Gamma_2;\Sigma_2$ if and only if $\vdash (\Gamma_0;\Sigma_0,x{:}\tau),\Sigma_1'',\Sigma_2' \rightsquigarrow \Gamma_2;\Sigma_2$. This holds by the induction hypothesis.

If $\Gamma_0 \vdash \tau : \mathsf{U}$, then by inversion of rule X-CONSU twice, it suffices to show that $\vdash (\Gamma_0,x{:}\tau;\Sigma_0),\Sigma_1'' \rightsquigarrow \Gamma_1;\Sigma_1$ and $\vdash (\Gamma_1;\Sigma_1),\Sigma_2' \rightsquigarrow \Gamma_2;\Sigma_2$ if and only if $\vdash (\Gamma_0,x{:}\tau;\Sigma_0),\Sigma_1'',\Sigma_2' \rightsquigarrow \Gamma_2;\Sigma_2$. This too holds by the induction hypothesis. □

A.3.1.2 Qualifier Soundness

LEMMA A.24 (Variance coherence)**.**

Suppose that $\Gamma, \beta{:}\langle\beta\rangle \vdash \tau : \kappa$ *and* $\Gamma, \beta{:}\langle\beta\rangle \vdash \beta \in \tau \updownarrow \mathfrak{v}$. *If* $\beta \in \mathrm{FTV}(\kappa)$ *then* $+ \sqsubseteq \mathfrak{v}$.

Proof. By induction on the kinding derivation:

Case $\dfrac{\alpha{:}\kappa \in \Gamma, \beta{:}\langle\beta\rangle \qquad \vdash \Gamma, \beta{:}\langle\beta\rangle}{\Gamma, \beta{:}\langle\beta\rangle \vdash \alpha : \kappa}$**.**

If $\alpha = \beta$ then $\mathfrak{v} = +$.

If $\alpha \neq \beta$ then $\alpha{:}\kappa \in \Gamma$. By inversion of rule WF-CONSAREC, $\vdash \Gamma$, and thus by lemma A.4, $\Gamma \vdash \kappa$. Since $\beta \notin \mathrm{dom}\,\Gamma$, we know that $\beta \notin \mathrm{FTV}(\kappa)$, which contradicts the lemma's assumptions.

Case $\dfrac{\Gamma, \beta{:}\langle\beta\rangle, \alpha{:}\langle\alpha\rangle \vdash \tau' : \kappa' \qquad \Gamma, \beta{:}\langle\beta\rangle, \alpha{:}\langle\alpha\rangle \vdash \alpha \in \tau' \updownarrow \mathfrak{v}'}{\Gamma, \beta{:}\langle\beta\rangle \vdash \lambda\alpha.\tau' : \Pi\alpha^{\mathfrak{v}'}.\kappa'}$**.**

By inversion of rule V-ABS,

(1) $\Gamma, \beta{:}\langle\beta\rangle, \alpha{:}\langle\alpha\rangle \vdash \beta \in \tau' \updownarrow \mathfrak{v}$.

If $\beta \in \mathrm{FTV}(\Pi\alpha^{\mathfrak{v}'}.\kappa')$ then $\beta \in \mathrm{FTV}(\kappa')$. Then by the induction hypothesis, $+ \sqsubseteq \mathfrak{v}$.

Case $\dfrac{\Gamma, \beta{:}\langle\beta\rangle \vdash \tau_1 : \Pi\alpha^{\mathfrak{v}_3}.\kappa_3 \qquad \Gamma, \beta{:}\langle\beta\rangle \vdash \tau_2 : \xi}{\Gamma, \beta{:}\langle\beta\rangle \vdash \tau_1\,\tau_2 : \{\xi/\alpha\}\kappa_3}$**.**

By inversion of rule V-APP, there exist some $\mathfrak{v}_1$ and $\mathfrak{v}_2$ such that

(1) $\Gamma, \beta{:}\langle\beta\rangle \vdash \beta \in \tau_1 \updownarrow \mathfrak{v}_1$,

(2) $\Gamma, \beta{:}\langle\beta\rangle \vdash \beta \in \tau_2 \updownarrow \mathfrak{v}_2$, and

(3) $\mathfrak{v} = \mathfrak{v}_1 \sqcup \mathfrak{v}_2\mathfrak{v}_3$.

If $\beta \in \mathrm{FTV}(\{\xi/\alpha\}\kappa_3)$ then either $\beta \in \mathrm{FTV}(\kappa_3)$ or both $\beta \in \mathrm{FTV}(\xi)$ and $\alpha \in \mathrm{FTV}(\kappa_3)$:

- If $\beta \in \mathrm{FTV}(\kappa_3)$, then $\beta \in \mathrm{FTV}(\Pi\alpha^{\mathfrak{v}_3}.\kappa_3)$. Then by the induction hypothesis, $+ \sqsubseteq \mathfrak{v}_1$, so $+ \sqsubseteq \mathfrak{v}_1 \sqcup \mathfrak{v}_2\mathfrak{v}_3$ as well.

- If $\beta \in \mathrm{FTV}(\xi)$ and $\alpha \in \mathrm{FTV}(\kappa_3)$, then by the induction hypothesis, $+ \sqsubseteq \mathfrak{v}_2$, and because $\Pi\alpha^{\mathfrak{v}_3}.\kappa_3$ is well-formed, $+ \sqsubseteq \mathfrak{v}_3$. Then $+ \sqsubseteq \mathfrak{v}_1 \sqcup \mathfrak{v}_2\mathfrak{v}_3$.

Case $\dfrac{\Gamma, \beta{:}\langle\beta\rangle, \alpha{:}\kappa' \vdash \tau' : \xi \qquad \Gamma, \beta{:}\langle\beta\rangle \vdash \kappa'}{\Gamma, \beta{:}\langle\beta\rangle \vdash \forall\alpha{:}\kappa'.\,\tau' : \{\mathsf{A}/\alpha\}\,\xi}$.

By inversion of rule V-ALL,

(1) $\Gamma, \beta{:}\langle\beta\rangle, \alpha{:}\kappa' \vdash \beta \in \tau' \updownarrow \mathfrak{v}_1$ and

(2) $\mathfrak{v}_2 = \begin{cases} \pm & \beta \in \mathrm{FTV}(\kappa') \\ \phi & \beta \notin \mathrm{FTV}(\kappa') \end{cases}$

where $\mathfrak{v} = \mathfrak{v}_1 \sqcup \mathfrak{v}_2$. If $\beta \in \mathrm{FTV}(\{\mathsf{A}/\alpha\}\,\xi)$ then $\beta \in \mathrm{FTV}(\xi)$. Then by the induction hypothesis $+ \sqsubseteq \mathfrak{v}_1$, which means that $+ \sqsubseteq \mathfrak{v}_1 \sqcup \mathfrak{v}_2$.

Case $\dfrac{\Gamma, \beta{:}\langle\beta\rangle \vdash \tau_1 : \xi_1 \qquad \Gamma, \beta{:}\langle\beta\rangle \vdash \tau_2 : \xi_2 \qquad \Gamma, \beta{:}\langle\beta\rangle \vdash \xi}{\Gamma, \beta{:}\langle\beta\rangle \vdash \tau_1 \xrightarrow{\xi} \tau_2 : \xi}$.

By inversion of rule V-ARR, there exist some $\mathfrak{v}_1$, $\mathfrak{v}_2$, and $\mathfrak{v}_3$ such that

(1) $\Gamma, \beta{:}\langle\beta\rangle \vdash \beta \in \tau_1 \updownarrow \mathfrak{v}_1$,

(2) $\Gamma, \beta{:}\langle\beta\rangle \vdash \beta \in \tau_2 \updownarrow \mathfrak{v}_2$,

(3) $\mathfrak{v}_3 = \begin{cases} + & \beta \in \mathrm{FTV}(\xi) \\ \phi & \beta \notin \mathrm{FTV}(\xi) \end{cases}$, and

(4) $\mathfrak{v} = -\mathfrak{v}_1 \sqcup \mathfrak{v}_2 \sqcup \mathfrak{v}_3$.

If $\beta \in \mathrm{FTV}(\xi)$ then $\mathfrak{v}_3 = +$, so $+ <: -\mathfrak{v}_1 \sqcup \mathfrak{v}_2 \sqcup \mathfrak{v}_3$.

Case $\dfrac{\vdash \Gamma, \beta{:}\langle\beta\rangle}{\Gamma, \beta{:}\langle\beta\rangle \vdash \chi : \xi}$.

$\beta \notin \mathrm{FTV}(\xi)$. □

LEMMA A.25 (Valuations and substitution)**.**

For any valuation $\mathcal{V}$, $\mathcal{V}[\alpha \mapsto \mathcal{V}(\xi')](\xi) = \mathcal{V}(\{\xi'/\alpha\}\,\xi)$.

Proof. By induction on ξ, with one non-trivial case:

$\boxed{\Gamma \vdash \kappa_1 \lessapprox \kappa_2}$ *(coarse subkinding)*

$$\frac{\Gamma \models \xi_1 \sqsubseteq \xi_2}{\Gamma \vdash \xi_1 \lessapprox \xi_2}\ \textsc{CKSub-Qual} \qquad \frac{\Gamma, \alpha{:}\langle\alpha\rangle \vdash \kappa_1 \lessapprox \kappa_2}{\Gamma \vdash \Pi\alpha^{\mathfrak{v}_1}.\kappa_1 \lessapprox \Pi\alpha^{\mathfrak{v}_2}.\kappa_2}\ \textsc{CKSub-Oper}$$

Figure A.1: Coarse subkinding relation for definition A.26

Case β**.**

If $\beta = \alpha$, then $\mathcal{V}[\alpha \mapsto \mathcal{V}(\xi')](\langle\beta\rangle) = \mathcal{V}(\xi') = \mathcal{V}(\{\xi'/\alpha\}\langle\beta\rangle)$.

If $\beta \neq \alpha$, then $\mathcal{V}[\alpha \mapsto \mathcal{V}(\xi')](\langle\beta\rangle) = \mathcal{V}(\langle\beta\rangle) = \mathcal{V}(\{\xi'/\alpha\}\langle\beta\rangle)$. □

For the next two lemmas, it will be useful to have the following definition:

DEFINITION A.26 (Coarse subkinding)**.**

The ***coarse subkinding*** *relation, defined in figure A.1, is an extension of the subkinding relation that is insensitive to variance. It should be clear that the new relation is a preorder. Note that for well-formed qualifiers, coarse subkinding corresponds to the usual subkinding relation.*

LEMMA A.27 (Coarse subkinding substitution)**.**

If $\Gamma \vdash \xi_1 \lessapprox \xi_2$ *and* $\Gamma, \alpha{:}\langle\alpha\rangle \vdash \kappa_1 \lessapprox \kappa_2$ *then* $\Gamma \vdash \{\xi_1/\alpha\}\kappa_1 \lessapprox \{\xi_2/\alpha\}\kappa_2$.

Proof. By induction on the derivation of $\Gamma, \alpha{:}\langle\alpha\rangle \vdash \kappa_1 \lessapprox \kappa_2$:

Case $\dfrac{\Gamma, \alpha{:}\langle\alpha\rangle \models \xi_1' \sqsubseteq \xi_2'}{\Gamma, \alpha{:}\langle\alpha\rangle \vdash \xi_1' \lessapprox \xi_2'}$**.**

Let $\mathcal{V}$ be an arbitrary valuation consistent with Γ. Then

(1) $\mathcal{V}(\xi_1) \sqsubseteq \mathcal{V}(\xi_2)$.

Let $\mathcal{V}_1 = \mathcal{V}[\alpha \mapsto \mathcal{V}(\xi_1)]$. By reflexivity, $\mathcal{V}_1(\langle\alpha\rangle) \sqsubseteq \mathcal{V}_1(\langle\alpha\rangle)$, so $\mathcal{V}_1$ is consistent with $\Gamma, \alpha{:}\langle\alpha\rangle$, which means that

(2) $\mathcal{V}_1(\xi_1') \sqsubseteq \mathcal{V}_1(\xi_2')$.

Let $\mathcal{V}_2 = \mathcal{V}[\alpha \mapsto \mathcal{V}(\xi_2)]$. Because $\mathcal{V}(\xi_1) \sqsubseteq \mathcal{V}(\xi_2)$, and by induction on the structure of ξ_2', this means that

(3) $\mathcal{V}_1(\xi_2') \sqsubseteq \mathcal{V}_2(\xi_2')$,

and by transitivity,

(4) $\mathcal{V}_1(\xi_1') \sqsubseteq \mathcal{V}_2(\xi_2')$.

Then by lemma A.25,

(5) $\mathcal{V}_1(\xi_1') = \mathcal{V}(\{\xi_1/\alpha\}\xi_1')$ and

(6) $\mathcal{V}_2(\xi_2') = \mathcal{V}(\{\xi_2/\alpha\}\xi_2')$.

Thus,

(7) $\mathcal{V}(\{\xi_1/\alpha\}\xi_1') \sqsubseteq \mathcal{V}(\{\xi_2/\alpha\}\xi_2')$,

and since $\mathcal{V}$ is an arbitrary valuation consistent with Γ,

(8) $\Gamma \vdash \{\xi_1/\alpha\}\xi_1' \lessapprox \{\xi_2/\alpha\}\xi_2'$.

Case $\dfrac{\Gamma, \alpha{:}\langle\alpha\rangle, \beta{:}\langle\beta\rangle \vdash \kappa_1' \lessapprox \kappa_2'}{\Gamma, \alpha{:}\langle\alpha\rangle \vdash \Pi\beta^{\mathfrak{v}_1}.\kappa_1' \lessapprox \Pi\beta^{\mathfrak{v}_2}.\kappa_2'}$.

By the induction hypothesis,

(1) $\Gamma, \beta{:}\langle\beta\rangle \vdash \{\xi_1/\alpha\}\kappa_1' \lessapprox \{\xi_2/\alpha\}\kappa_2'$.

Then by rule CKSUB-OPER. □

LEMMA 5.4 (Monotonicity of kinding, restated from p. 119)**.**

If $\Gamma \vdash \tau_1 <:^+ \tau_2$ *where* $\Gamma \vdash \tau_1 : \xi_1$ *and* $\Gamma \vdash \tau_2 : \xi_2$*, then* $\Gamma \vdash \xi_1 <: \xi_2$.

Proof. Generalize the induction hypothesis as follows:

If $\Gamma \vdash \tau_1 <:^+ \tau_2$, $\Gamma \vdash \tau_1 : \kappa_1$, and $\Gamma \vdash \tau_2 : \kappa_2$, then $\Gamma \vdash \kappa_1 \lessapprox \kappa_2$.

Note that for well-formed qualifier expressions, subkinding and coarse subkinding are identical, which means that the generalized induction hypothesis implies the original lemma.

Now by induction on the subtyping derivation:

Case $\dfrac{\tau_1 \equiv \tau_2 \qquad \Gamma \vdash \tau_1 : \kappa \qquad \Gamma \vdash \tau_2 : \kappa}{\Gamma \vdash \tau_1 <:^+ \tau_2}$.

By lemma A.7, $\kappa_2 = \kappa = \kappa_1$; then by reflexivity of coarse subkinding.

Case $\dfrac{\Gamma \vdash \tau_1 <:^+ \tau_3 \qquad \Gamma \vdash \tau_3 <:^+ \tau_2 \qquad \Gamma \vdash \tau_3 : \kappa_3}{\Gamma \vdash \tau_1 <:^+ \tau_2}$.

By the induction hypothesis twice and transitivity.

Case $\dfrac{\Gamma \vdash \tau_2 <:^+ \tau_1}{\Gamma \vdash \tau_1 <:^- \tau_2}$.

Vacuous.

Case $\dfrac{\Gamma, \beta{:}\langle\beta\rangle \vdash \tau_1' <:^+ \tau_2'}{\Gamma \vdash \lambda\beta.\tau_1' <:^+ \lambda\beta.\tau_2'}$.

By inversion of rule K-ABS, $\kappa_1 = \Pi\beta^{\mathfrak{v}_1}.\kappa_1'$ for some κ_1' and $\mathfrak{v}_1$ such that

(1) $\Gamma, \beta{:}\langle\beta\rangle \vdash \beta \in \tau_1' \updownarrow \mathfrak{v}_1$ and

(2) $\Gamma, \beta{:}\langle\beta\rangle \vdash \tau_1' : \kappa_1'$.

Likewise, where $\kappa_2 = \Pi\beta^{\mathfrak{v}_2}.\kappa_2'$ such that

(3) $\Gamma, \beta{:}\langle\beta\rangle \vdash \beta \in \tau_2' \updownarrow \mathfrak{v}_2$ and

(4) $\Gamma, \beta{:}\langle\beta\rangle \vdash \tau_2' : \kappa_2'$.

By the induction hypothesis,

(5) $\Gamma, \beta{:}\langle\beta\rangle \vdash \kappa_1' \lessapprox \kappa_2'$,

which is sufficient to show that

(6) $\Gamma \vdash \Pi\beta^{\mathfrak{v}_1}.\kappa_1' \lessapprox \Pi\beta^{\mathfrak{v}_2}.\kappa_2'$.

$$\textbf{Case}\ \frac{\Gamma \vdash \tau_{11} : \Pi\beta^{\mathfrak{v}_1}.\kappa'_1 \qquad \Gamma \vdash \tau_{21} : \Pi\beta^{\mathfrak{v}_2}.\kappa'_2 \quad \Gamma \vdash \tau_{11} <:^{+} \tau_{21} \qquad \Gamma \vdash \tau_{12} <:^{\mathfrak{v}_1 \sqcup \mathfrak{v}_2} \tau_{22}}{\Gamma \vdash \tau_{11}\,\tau_{12} <:^{+} \tau_{21}\,\tau_{22}}.$$

By inversion of rule K-APP twice, there are some ξ_1 and ξ_2 such that

(1) $\Gamma \vdash \tau_{12} : \xi_1$,

(2) $\Gamma \vdash \tau_{22} : \xi_2$,

(3) $\kappa_1 = \{\xi_1/\beta\}\kappa'_1$, and

(4) $\kappa_2 = \{\xi_2/\beta\}\kappa'_2$.

By the induction hypothesis,

(5) $\Gamma \vdash \Pi\beta^{\mathfrak{v}_1}.\kappa'_1 \lessapprox \Pi\beta^{\mathfrak{v}_2}.\kappa'_2$,

which means that

(6) $\Gamma, \beta{:}\langle\beta\rangle \vdash \kappa'_1 \lessapprox \kappa'_2$.

Now by cases on $\mathfrak{v}_1 \sqcup \mathfrak{v}_2$:

Case $+$.

That is, $\Gamma \vdash \tau_{12} <:^{+} \tau_{22}$. By the induction hypothesis, $\Gamma \vdash \xi_1 \lessapprox \xi_2$, and thus by lemma A.27, $\Gamma \vdash \{\xi_1/\beta\}\kappa'_1 \lessapprox \{\xi_2/\beta\}\kappa'_2$.

Case $-$.

By lemma A.4, $\Gamma \vdash \Pi\beta^{\mathfrak{v}_1}.\kappa'_1$ and $\Gamma \vdash \Pi\beta^{\mathfrak{v}_2}.\kappa'_2$. By inversion of rule OK-OPER, this means that if $\beta \in \mathrm{FTV}(\kappa'_1)$ then $+ \sqsubseteq \mathfrak{v}_1$, and likewise, if $\beta \in \mathrm{FTV}(\kappa'_2)$ then $+ \sqsubseteq \mathfrak{v}_2$. Since $\mathfrak{v}_1 \sqcup \mathfrak{v}_2 = -$, neither $+ \sqsubseteq \mathfrak{v}_1$ nor $+ \sqsubseteq \mathfrak{v}_2$, which means that $\beta \notin \mathrm{FTV}(\kappa'_1)$ and $\beta \notin \mathrm{FTV}(\kappa'_2)$. Thus $\{\xi_1/\beta\}\kappa'_1 = \kappa'_1$ and $\{\xi_2/\beta\}\kappa'_2 = \kappa'_2$. By (6), $\Gamma, \beta{:}\langle\beta\rangle \vdash \{\xi_1/\beta\}\kappa'_1 \lessapprox \{\xi_2/\beta\}\kappa'_2$, and since every valuation consistent with Γ is also consistent with $\Gamma, \beta{:}\langle\beta\rangle$, we have that $\Gamma \vdash \{\xi_1/\beta\}\kappa'_1 \lessapprox \{\xi_2/\beta\}\kappa'_2$.

Case ϕ**.**

As in the previous case, because $\mathfrak{v}_1 \sqcup \mathfrak{v}_2 = \phi$ means that neither $+ \sqsubseteq \mathfrak{v}_1$ nor $+ \sqsubseteq \mathfrak{v}_2$.

Case $\pm$**.**

That is, $\Gamma \vdash \tau_{12} <:^{\pm} \tau_{22}$. By a simple induction on the subtyping derivation, this means that $\tau_{12} \equiv \tau_{22}$, $\Gamma \vdash \tau_{12} : \kappa$, and $\Gamma \vdash \tau_{22} : \kappa$. By rule TSUB-EQ, $\Gamma \vdash \tau_{12} <:^{+} \tau_{22}$, so as in the $\mathfrak{v}_1 \sqcup \mathfrak{v}_2 = +$ case above.

Case $\dfrac{\Gamma, \beta{:}\kappa' \vdash \tau_1' <:^{+} \tau_2'}{\Gamma \vdash \forall \beta{:}\kappa'.\, \tau_1' <:^{+} \forall \beta{:}\kappa'.\, \tau_2'}$**.**

By inversion of rule K-ALL twice and the induction hypothesis.

Case $\dfrac{\Gamma \vdash \tau_{11} <:^{-} \tau_{21} \qquad \Gamma \vdash \tau_{12} <:^{+} \tau_{22} \qquad \Gamma \vdash \xi_1 <: \xi_2}{\Gamma \vdash \tau_{11} \xrightarrow{\xi_1}\!\!\circ\, \tau_{12} <:^{+} \tau_{21} \xrightarrow{\xi_2}\!\!\circ\, \tau_{22}}$**.**

By the premise that $\Gamma \vdash \xi_1 <: \xi_2$. □

LEMMA A.28 (Location coverage)**.**

If $\Gamma; \Sigma \rhd e : \tau$ *then* $\mathrm{FL}(e) \subseteq \mathrm{dom}\, \Sigma$.

Proof. By induction on the typing derivation. □

LEMMA 5.5 (Kinding finds locations, restated from p. 120)**.**

Suppose that $\Gamma; \Sigma \rhd v : \tau$ *and* $\Gamma \vdash \tau : \xi$. *If any locations appear in value* v *then* $\Gamma \vdash \mathsf{A} <: \xi$. *Contrapositively, if* $\xi = \mathsf{U}$ *then* $\mathrm{FL}(v) = \emptyset$.

Proof. Assume that $\mathrm{FL}(v) \neq \emptyset$. By induction on the typing derivation $\Gamma; \Sigma \rhd v : \tau$:

Case $\dfrac{\Gamma; \Sigma \rhd v : \tau' \qquad \Gamma \vdash \tau' <:^{+} \tau \qquad \Gamma \vdash \tau : \xi}{\Gamma; \Sigma \rhd v : \tau}$**.**

By lemma A.4, there is some ξ' such that $\Gamma \vdash \tau' : \xi'$, and by the induction hypothesis, $\Gamma \vdash \mathsf{A} <: \xi'$. By lemma 5.4, $\Gamma \vdash \xi' <: \xi$, and then by transitivity.

Case $\dfrac{\Gamma;\Sigma \rhd v : \tau \qquad \vdash \Gamma,\Gamma';\Sigma,\Sigma'}{\Gamma,\Gamma';\Sigma,\Sigma' \rhd v : \tau}$.

By the induction hypothesis.

Case $\dfrac{\ell{:}\tau \in \Sigma \qquad \bullet \vdash \tau : \xi \qquad \vdash \Gamma;\Sigma}{\Gamma;\Sigma \rhd \ell : \mathsf{ref}\ \tau}$.

Then $\Gamma \vdash \mathsf{ref}\ \tau : \mathsf{A}$.

Case $\dfrac{\vdash (\Gamma;\Sigma), x{:}\tau_1 \rightsquigarrow \Gamma';\Sigma' \qquad \Gamma';\Sigma' \rhd e : \tau_2 \qquad \Gamma \vdash \Sigma \preceq \xi \qquad \Gamma \vdash \tau_1 : \xi_1}{\Gamma;\Sigma \rhd \lambda x{:}\tau_1.e : \tau_1 \xrightarrow{\xi} \tau_2}$.

By lemma A.28, $\mathrm{FL}(\lambda x{:}\tau_1.e) \subseteq \mathrm{dom}\,\Sigma$, and since $\mathrm{FL}(\lambda x{:}\tau_1.e) \neq \emptyset$, we know that dom Σ must contain some locations. Then by rule B-ConsL, $\xi = \mathsf{A}$.

Case $\dfrac{\Gamma, \alpha{:}\kappa;\Sigma \rhd v' : \tau \qquad \Gamma \vdash \kappa}{\Gamma;\Sigma \rhd \Lambda\alpha{:}\kappa.v' : \forall\alpha{:}\kappa.\tau}$.

By the induction hypothesis $\Gamma \vdash \tau : \mathsf{A}$, and $\xi = \{\mathsf{A}/\alpha\}\,\mathsf{A} = \mathsf{A}$.

Case $\dfrac{\vdash \Gamma;\Sigma}{\Gamma;\Sigma \rhd \langle\rangle : 1}$.

Vacuous, as $\mathrm{FL}(\langle\rangle) = \emptyset$.

Case $\dfrac{\Gamma;\Sigma \rhd v' : \tau_1 \qquad \Gamma \vdash \tau_2 : \xi_2}{\Gamma;\Sigma \rhd \mathsf{inl}\ v' : \tau_1 \oplus \tau_2}$.

By lemma A.4, there is some ξ_1 such that $\Gamma \vdash \tau_1 : \xi_1$. If $\mathrm{FL}(\mathsf{inl}\ v') \neq \emptyset$ then $\mathrm{FL}(v') \neq \emptyset$. Then by the induction hypothesis $\Gamma \vdash \mathsf{A} <: \xi_1$. By rule K-Sum and rule K-App twice, $\Gamma \vdash \tau_1 \oplus \tau_2 : \xi_1 \sqcup \xi_2$. Then $\Gamma \vdash \xi_1 <: \xi_1 \sqcup \xi_2$, and by transitivity.

Case $\dfrac{\Gamma;\Sigma \rhd v' : \tau_2 \qquad \Gamma \vdash \tau_1 : \xi}{\Gamma;\Sigma \rhd \mathsf{inr}\ v' : \tau_1 \oplus \tau_2}$.

As in the previous case.

Case $\dfrac{\Gamma;\Sigma_1 \rhd v_1 : \tau_1 \qquad \Gamma;\Sigma_2 \rhd v_2 : \tau_2}{\Gamma;\Sigma_1,\Sigma_2 \rhd \langle v_1, v_2\rangle : \tau_1 \otimes \tau_2}$**.**

By lemma A.4, there are some ξ_1 and ξ_2 such that $\Gamma \vdash \tau_1 : \xi_1$ and $\Gamma \vdash \tau_2 : \xi_2$.

If $\mathrm{FL}(\langle v_1, v_2\rangle) \neq \emptyset$ then $\mathrm{FL}(v_1) \neq \emptyset$ or $\mathrm{FL}(v_2) \neq \emptyset$:

Case $\mathrm{FL}(v_1) \neq \emptyset$**.**

By the induction hypothesis, $\Gamma \vdash A <: \xi_1$. Then by rule K-PROD, rule K-APP twice, and transitivity.

Case $\mathrm{FL}(v_2) \neq \emptyset$**.**

By symmetry.

Otherwise.

The remaining cases do not apply to values. □

A.3.1.3 Preservation

LEMMA 5.6 (Substitution, restated from p. 120)**.**

If

- $\vdash (\Gamma;\Sigma_1), x{:}\tau' \rightsquigarrow \Gamma';\Sigma_1'$,
- $\Gamma';\Sigma_1' \rhd e : \tau$, *and*
- $\bullet;\Sigma_2 \rhd v : \tau'$, *where*
- *the domain of* Σ_2 *contains only locations,*

then $\Gamma;\Sigma_1,\Sigma_2 \rhd \{v/x\}\, e : \tau$.

Proof. In several cases, we will need to know that $\vdash \Gamma;\Sigma_1,\Sigma_2$, typically in order to use rule T-WEAK. By lemma A.4, we have $\vdash \Gamma';\Sigma_1'$ and $\vdash \bullet;\Sigma_2$. By lemma A.4 again, $\vdash \Gamma;\Sigma_1$; by weakening, $\vdash \Gamma;\Sigma_2$. Then by lemma A.3, $\vdash \Gamma;\Sigma_1,\Sigma_2$.

Now by induction on the derivation of $\Gamma';\Sigma_1' \rhd e : \tau$:

Case $\dfrac{\Gamma';\Sigma_1' \rhd e : \tau'' \qquad \Gamma' \vdash \tau'' <:^+ \tau \qquad \Gamma' \vdash \tau : \xi'}{\Gamma';\Sigma_1' \rhd e : \tau}$.

By the induction hypothesis,

(1) $\Gamma;\Sigma_1,\Sigma_2 \rhd \{v/x\}\, e : \tau''$.

By observation A.1,

(2) $\Gamma \vdash \tau'' <:^+ \tau$ and

(3) $\Gamma \vdash \tau : \xi'$.

Then by rule T-SUBSUME.

Case $\dfrac{\Gamma_1';\Sigma_{11}' \rhd e : \tau \qquad \vdash \Gamma_1',\Gamma_2';\Sigma_{11}',\Sigma_{12}'}{\Gamma_1',\Gamma_2';\Sigma_{11}',\Sigma_{12}' \rhd e : \tau}$.

We do not know whether $x{:}\tau'$ is in Γ_1', Γ_2', Σ_{11}', or Σ_{12}':

- If $x{:}\tau' \in \Gamma_1',\Sigma_{11}'$, then there exist some Γ_1 and Σ_{11} such that

  (1) $\vdash (\Gamma_1;\Sigma_{11}), x{:}\tau' \rightsquigarrow \Gamma_1';\Sigma_{11}'$ and

  (2) $\vdash (\Gamma_1,\Gamma_2';\Sigma_{11},\Sigma_{12}'), x{:}\tau' \rightsquigarrow \Gamma_1',\Gamma_2';\Sigma_{11}',\Sigma_{12}'$.

  Then by the induction hypothesis,

  (3) $\Gamma_1;\Sigma_{11},\Sigma_2 \rhd \{v/x\}\, e : \tau$,

  and by rule T-WEAK,

  (4) $\Gamma_1,\Gamma_2;\Sigma_{11},\Sigma_{12},\Sigma_2 \rhd \{v/x\}\, e : \tau$.

- If $x{:}\tau' \in \Gamma_2',\Sigma_{12}'$, then by lemma A.22, $x \notin \mathrm{FV}(e)$. This means that $\{v/x\}\, e = e$, so

  (5) $\Gamma_1';\Sigma_{11}' \rhd \{v/x\}\, e : \tau$.

  Then by rule T-WEAK.

Case $\dfrac{y{:}\tau \in \Gamma',\Sigma_1' \qquad \Gamma' \vdash \tau : \xi' \qquad \vdash \Gamma';\Sigma_1'}{\Gamma';\Sigma_1' \rhd y : \tau}$.

If $x = y$, then $\{v/x\}\, y = v$ and $\tau = \tau'$. Thus,

(1) $\Gamma;\Sigma_2 \rhd \{v/x\}\, y : \tau$.

Then by rule T-WEAK.

If $x \neq y$, then $\{v/x\}\, y = y$. Furthermore, this means that $y{:}\tau \in \Gamma, \Sigma_1$, since the only difference between Γ, Σ_1 and Γ', Σ_1' is $x{:}\tau'$. Then by rule T-VAR,

(2) $\Gamma; \Sigma_1 \rhd \{v/x\}\, y : \tau$.

Then by rule T-WEAK.

Case $\dfrac{\ell{:}\tau_1 \in \Sigma_1' \qquad \bullet \vdash \tau_1 : \xi_1 \qquad \vdash \Gamma'; \Sigma_1'}{\Gamma'; \Sigma_1' \rhd \ell : \mathsf{ref}\ \tau_1}$.

Since the only difference between Σ_1 and Σ_1' may be $x{:}\tau'$, we know that $\ell{:}\tau_1 \in \Sigma_1$. Furthermore, $\{v/x\}\, \ell = \ell$, so by rule T-PTR,

(1) $\bullet; \Sigma_1 \rhd \ell : \mathsf{ref}\ \tau_1$.

Then by rule T-WEAK.

Case $\dfrac{\begin{array}{c}\vdash (\Gamma'; \Sigma_1'), y{:}\tau_1 \rightsquigarrow \Gamma''; \Sigma_1'' \\ \Gamma''; \Sigma_1'' \rhd e_2 : \tau_2 \qquad \Gamma' \vdash \Sigma_1' \preceq \xi_1' \qquad \Gamma' \vdash \tau_1 : \xi_1\end{array}}{\Gamma'; \Sigma_1' \rhd \lambda y{:}\tau_1.\, e_2 : \tau_1 \xrightarrow{\xi_1'} \tau_2}$.

Note that Γ, Γ', and Γ'' differ only by variable bindings, so by observation A.1, we can use Γ in suitable judgments throughout.

By lemma A.4, there exists some ξ' such that $\Gamma \vdash \tau' : \xi'$. If $x{:}\tau' \in \Sigma_1'$, then by inversion of rule X-CONSA, there exists some ξ such that $\Gamma \vdash \Sigma_1 \preceq \xi$ and $\xi_1' = \xi \sqcup \xi'$. Otherwise, $\Sigma_1 = \Sigma_1'$, so let $\xi = \xi_1'$. In both cases,

(1) $\Gamma \vdash \xi <: \xi_1'$ and

(2) $\Gamma \vdash \Sigma_1 \preceq \xi$.

Let us consider whether there exists some $\ell \in \mathrm{FL}(v)$:

- If so, then by lemma 5.5, $\xi' = \mathsf{A}$. In order to type v, it must be that $\ell \in \mathrm{dom}\ \Sigma_2$, which means that $\Gamma \vdash \Sigma_2 \preceq \mathsf{A}$. Let $\Sigma_2' = \Sigma_2$ and $\xi_2' = \mathsf{A}$. Furthermore, since $x{:}\tau' \in \Sigma_1'$, by lemma A.9, $\Gamma \vdash \Sigma_1' \preceq \mathsf{A}$. Then $\xi_1' = \mathsf{A}$, so $\Gamma \vdash \xi \sqcup \xi_2' <: \xi_1'$.

- If not, then since Σ_2 contains only locations, $\bullet;\bullet \rhd v : \tau'$. Let $\Sigma_2' = \bullet$ and $\xi_2' = \mathsf{U}$. Then $\xi \sqcup \xi_2' = \xi$, so $\Gamma \vdash \xi \sqcup \xi_2' <: \xi_1'$.

In either case,

(3) $\Gamma \vdash \xi \sqcup \xi_2' <: \xi_1'$.

Since $\Gamma \vdash \Sigma_1 \preceq \xi$ and $\Gamma \vdash \Sigma_2' \preceq \xi_2'$, we have by lemma A.9 that

(4) $\Gamma \vdash \Sigma_1, \Sigma_2' \preceq \xi \sqcup \xi_2'$.

By lemma A.23,

(5) $\vdash (\Gamma; \Sigma_1), x{:}\tau', y{:}\tau_1 \rightsquigarrow \Gamma''; \Sigma_1''$,

and since we identify environments up to permutation,

(6) $\vdash (\Gamma; \Sigma_1), y{:}\tau_1, x{:}\tau' \rightsquigarrow \Gamma''; \Sigma_1''$.

By the same lemma, let Γ''' and Σ_1''' be such that

(7) $\vdash (\Gamma; \Sigma_1), y{:}\tau_1 \rightsquigarrow \Gamma'''; \Sigma_1'''$ and

(8) $\vdash (\Gamma'''; \Sigma_1'''), x{:}\tau' \rightsquigarrow \Gamma''; \Sigma_1''$.

By the induction hypothesis,

(9) $\Gamma'''; \Sigma_1''', \Sigma_2' \rhd \{v/x\}\, e_2 : \tau_2$.

Note that by sorting τ_1 the same way as above, we get

(10) $\vdash (\Gamma; \Sigma_1, \Sigma_2'), y{:}\tau_1 \rightsquigarrow \Gamma'''; \Sigma_1''', \Sigma_2'$,

and by observation A.1,

(11) $\Gamma \vdash \tau_1 : \xi_1$.

Then by rule T-ABS and (4, 9–11),

(12) $\Gamma;\Sigma_1,\Sigma_2' \rhd \lambda y{:}\tau_1.e_1 : \tau_1 \xrightarrow{\xi \sqcup \xi_2'} \tau_2$.

By rule K-ARR and (3),

(13) $\Gamma \vdash \tau_1 \xrightarrow{\xi \sqcup \xi_2'} \tau_2 <:^+ \tau_1 \xrightarrow{\xi_1'} \tau_2$.

Noting that $\Sigma_2' \subseteq \Sigma_2$, by rule T-WEAK and rule T-SUBSUME,

(14) $\Gamma;\Sigma_1,\Sigma_2 \rhd \lambda y{:}\tau_1.e_1 : \tau_1 \xrightarrow{\xi_1'} \tau_2$.

Case $\dfrac{\Gamma';\Sigma_{11}' \rhd e_1 : \tau_1 \xrightarrow{\xi} \tau \qquad \Gamma';\Sigma_{12}' \rhd e_2 : \tau_1}{\Gamma';\Sigma_{11}',\Sigma_{12}' \rhd e_1\, e_2 : \tau}$.

Let Σ_{11} and Σ_{12} be Σ_{11}' and Σ_{12}', respectively, but without $x{:}\tau'$. We know that $x{:}\tau'$ is in one of Γ', Σ_{11}', or Σ_{12}':

Case $x{:}\tau' \in \Gamma'$**.**

Then

(1) $\vdash (\Gamma;\Sigma_{11}), x{:}\tau' \rightsquigarrow \Gamma';\Sigma_{11}$ and

(2) $\vdash (\Gamma;\Sigma_{12}), x{:}\tau' \rightsquigarrow \Gamma';\Sigma_{12}$.

Furthermore, by lemma A.4 and inversion of rule WF,

(3) $\Gamma' \vdash \Gamma' \preceq \mathsf{U}$.

By lemma A.9, this means that

(4) $\Gamma' \vdash \tau' : \mathsf{U}$.

By lemma 5.5, $\mathrm{FL}(v) = \varnothing$. Since we know that Σ_2 contains only locations, none of it is relevant to v, so

(5) $\bullet;\bullet \rhd v : \tau'$.

Then by the induction hypothesis twice,

(6) $\Gamma;\Sigma_{11} \rhd \{v/x\}e_1 : \tau_1 \xrightarrow{\xi} \tau$ and

(7) $\Gamma;\Sigma_{12} \rhd \{v/x\}e_2 : \tau_1$.

Then by rule T-APP and rule T-WEAK.

Case $x{:}\tau' \in \Sigma'_{11}$**.**

Then

(1) $\vdash (\Gamma; \Sigma_{11}), x{:}\tau' \rightsquigarrow \Gamma; \Sigma'_{11}$.

Then by the induction hypothesis,

(2) $\Gamma; \Sigma_{11}, \Sigma_2 \rhd \{v/x\} e_1 : \tau_1 \xrightarrow{\xi}_{\circ} \tau_2$.

Furthermore, since $x \notin \mathrm{dom}(\Gamma', \Sigma'_{12})$, we know that $x \notin \mathrm{FV}(e_2)$, so

(3) $\Sigma_{12} = \Sigma'_{12}$,

(4) $\Gamma = \Gamma'$, and

(5) $\{v/x\} e_2 = e_2$.

Substituting those equalities into the appropriate premise, we get

(6) $\Gamma; \Sigma_{12} \rhd \{v/x\} e_2 : \tau_1$.

Then by rule T-APP.

Case $x{:}\tau' \in \Sigma'_{12}$**.**

Then by symmetry with the previous case.

Case $\dfrac{\Gamma', \alpha{:}\kappa; \Sigma'_1 \rhd v_1 : \tau_1}{\Gamma'; \Sigma'_1 \rhd \Lambda\alpha{:}\kappa.\, v_1 : \forall \alpha{:}\kappa.\, \tau_1}$**.**

Since $\vdash (\Gamma; \Sigma_1), x{:}\tau' \rightsquigarrow \Gamma'; \Sigma'_1$, we know that $\vdash (\Gamma, \alpha{:}\kappa; \Sigma_1), x{:}\tau' \rightsquigarrow \Gamma', \alpha{:}\kappa; \Sigma'_1$.
Then by the induction hypothesis,

(1) $\Gamma, \alpha{:}\kappa; \Sigma_1, \Sigma_2 \rhd \{v/x\} v_1 : \tau_1$.

Then by rule T-TABS.

Case $\dfrac{\Gamma'; \Sigma'_1 \rhd e_1 : \forall \alpha{:}\kappa_1.\, \tau_2 \qquad \Gamma' \vdash \tau_1 : \kappa_1}{\Gamma'; \Sigma'_1 \rhd e_1\, \tau_1 : \{\tau_1/\alpha\} \tau_2}$**.**

By the induction hypothesis and observation A.1,

(1) $\Gamma; \Sigma_1, \Sigma_2 \rhd \{v/x\} e_1 : \forall \alpha{:}\kappa_1.\, \tau_2$ and

(2) $\Gamma \vdash \tau_1 : \kappa_1$.

Then by rule T-TAPP, noting that $\{v/x\}(e_1\, \tau_1) = (\{v/x\} e_1)\, \tau_1$.

Case $\dfrac{\Gamma';\Sigma'_1 \rhd e_1 : \tau \xrightarrow{\mathsf{U}}_{\circ} \tau}{\Gamma';\Sigma'_1 \rhd \mathsf{fix}\, e_1 : \tau}$.

By the induction hypothesis and rule T-FIX.

Case $\dfrac{\vdash \Gamma';\Sigma'_1}{\Gamma';\Sigma'_1 \rhd \langle\rangle : 1}$.

By lemma A.19 and rule T-UNIT.

Case $\dfrac{\Gamma';\Sigma'_1 \rhd e_1 : \tau_1 \qquad \Gamma' \vdash \tau_2 : \xi_2}{\Gamma';\Sigma'_1 \rhd \mathsf{inl}\, e_1 : \tau_1 \oplus \tau_2}$.

By the induction hypothesis, observation A.1, and rule T-INL.

Case $\dfrac{\Gamma';\Sigma'_1 \rhd e_2 : \tau_2 \qquad \Gamma' \vdash \tau_1 : \xi_1}{\Gamma';\Sigma'_1 \rhd \mathsf{inr}\, e_2 : \tau_1 \oplus \tau_2}$.

By the induction hypothesis, observation A.1, and rule T-INR.

Case $\dfrac{\begin{array}{c}\Gamma';\Sigma'_{11} \rhd e' : \tau_1 \oplus \tau_2 \\ \vdash (\Gamma';\Sigma'_{12}), x_1{:}\tau_1 \rightsquigarrow \Gamma'_1;\Sigma'_{121} \qquad \Gamma'_1;\Sigma'_{121} \rhd e_1 : \tau \\ \vdash (\Gamma';\Sigma'_{12}), x_2{:}\tau_2 \rightsquigarrow \Gamma'_2;\Sigma'_{122} \qquad \Gamma'_2;\Sigma'_{122} \rhd e_2 : \tau\end{array}}{\Gamma';\Sigma'_{11},\Sigma'_{12} \rhd \mathsf{case}\, e' \,\mathsf{of}\, \mathsf{inl}\, x_1 \to e_2; \mathsf{inr}\, x_2 \to e_2 : \tau}$.

Let Σ_{11} and Σ_{12} be Σ'_{11} and Σ'_{12}, respectively, but without $x{:}\tau'$. By lemma A.23,

(1) $\vdash (\Gamma;\Sigma_{12}), x{:}\tau', x_1{:}\tau_1 \rightsquigarrow \Gamma'_1;\Sigma'_{121}$ and

(2) $\vdash (\Gamma;\Sigma_{12}), x{:}\tau', x_2{:}\tau_2 \rightsquigarrow \Gamma'_2;\Sigma'_{122}$.

Then by the same lemma, let Γ''_1, Γ''_2, Σ''_{121}, and Σ''_{122} be such that

(3) $\vdash (\Gamma;\Sigma_{12}), x_1{:}\tau_1 \rightsquigarrow \Gamma''_1;\Sigma''_{121}$,

(4) $\vdash (\Gamma;\Sigma_{12}), x_2{:}\tau_2 \rightsquigarrow \Gamma''_2;\Sigma''_{122}$,

(5) $\vdash (\Gamma''_1;\Sigma''_{121}), x{:}\tau' \rightsquigarrow \Gamma'_1;\Sigma'_{121}$, and

(6) $\vdash (\Gamma''_2;\Sigma''_{122}), x{:}\tau' \rightsquigarrow \Gamma'_2;\Sigma'_{122}$.

We know that $x{:}\tau'$ is in one of Γ', Σ'_{11}, or Σ'_{12}:

Case $x{:}\tau' \in \Gamma'$**.**

By lemma A.4 and inversion of rule WF,

(1) $\Gamma' \vdash \Gamma' \preceq \mathsf{U}$.

Then by lemma A.9, this means that

(2) $\Gamma' \vdash \tau' : \mathsf{U}$.

By lemma 5.5, $\mathrm{FL}(v) = \varnothing$. Since we know that Σ_2 contains only locations,

(3) $\bullet;\bullet \rhd v : \tau'$.

By weakening,

(4) $\Gamma_1'';\bullet \rhd v : \tau'$ and

(5) $\Gamma_2'';\bullet \rhd v : \tau'$.

Then by the induction hypothesis three times,

(6) $\Gamma;\Sigma_{11},\bullet \rhd \{v/x\}e : \tau_1 \oplus \tau_2$,

(7) $\Gamma_1'';\Sigma_{121}'',\bullet \rhd \{v/x\}e_1 : \tau$, and

(8) $\Gamma_2'';\Sigma_{122}'',\bullet \rhd \{v/x\}e_2 : \tau$.

By rule T-APP,

(9) $\Gamma;\Sigma_{11},\Sigma_{12} \rhd \{v/x\}(\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x_1 \to e_1; \mathsf{inr}\ x_2 \to e_2) : \tau$.

Then by rule T-WEAK.

Case $x{:}\tau' \in \Sigma_{11}'$**.**

Then

(1) $\vdash (\Gamma;\Sigma_{11}),x{:}\tau' \rightsquigarrow \Gamma;\Sigma_{11}'$,

so by the induction hypothesis,

(2) $\Gamma;\Sigma_{11},\Sigma_2 \rhd \{v/x\}e' : \tau_1 \oplus \tau_2$.

Furthermore,

- $\Gamma = \Gamma'$,
- $\Gamma_1' = \Gamma_1''$,
- $\Gamma_2' = \Gamma_2''$,
- $\Sigma_{12} = \Sigma_{12}'$,
- $\Sigma_{121}' = \Sigma_{121}''$, and
- $\Sigma_{122}' = \Sigma_{122}''$.

Since $x{:}\tau \in \Sigma'_{11}$, $x \notin \mathrm{dom}(\Gamma', \Sigma'_{12})$, which means that $x \notin \mathrm{dom}(\Gamma'_1, \Sigma'_{121})$ and $x \notin \mathrm{dom}(\Gamma'_2, \Sigma'_{122})$. This means that $x \notin \mathrm{FV}(e_1)$ and $x \notin \mathrm{FV}(e_2)$. Thus, $\{v/x\}e_1 = e_1$ and $\{v/x\}e_2 = e_2$, which gives us

(3) $\Gamma''_1; \Sigma''_{121} \rhd \{v/x\}e_1 : \tau$ and

(4) $\Gamma''_2; \Sigma''_{122} \rhd \{v/x\}e_2 : \tau$.

Then by rule T-CASE.

Case $x{:}\tau' \in \Sigma'_{12}$**.**

This means that $\Gamma' = \Gamma$ and $\Sigma'_{11} = \Sigma_{11}$. Furthermore, $x \notin \mathrm{dom}(\Gamma', \Sigma'_{11})$, which means that $x \notin \mathrm{FV}(e')$. Thus, we know that $\{v/x\}e' = e'$, so

(1) $\Gamma; \Sigma_{11} \rhd \{v/x\}e' : \tau_1 \oplus \tau_2$.

From our assumptions, we have $\bullet; \Sigma_2 \rhd v : \tau'$. By the induction hypothesis twice,

(2) $\Gamma''_1; \Sigma''_{121}, \Sigma_2 \rhd \{v/x\}e_1 : \tau$ and

(3) $\Gamma''_2; \Sigma''_{122}, \Sigma_2 \rhd \{v/x\}e_2 : \tau$.

Note that

(4) $\vdash (\Gamma; \Sigma_{12}, \Sigma_2), x_1{:}\tau_1 \rightsquigarrow \Gamma''_1; \Sigma''_{121}, \Sigma_2$ and

(5) $\vdash (\Gamma; \Sigma_{12}, \Sigma_2), x_2{:}\tau_2 \rightsquigarrow \Gamma''_2; \Sigma''_{122}, \Sigma_2$.

Then by rule T-CASE.

Case $\dfrac{\Gamma'; \Sigma'_{11} \rhd v_1 : \tau_1 \qquad \Gamma'; \Sigma'_{12} \rhd v_2 : \tau_2}{\Gamma'; \Sigma'_{11}, \Sigma'_{12} \rhd \langle v_1, v_2 \rangle : \tau_1 \otimes \tau_2}$**.**

As in the T-APP case.

Case $\dfrac{\begin{array}{c}\Gamma'; \Sigma'_{11} \rhd e' : \tau_1 \otimes \tau_2 \\ \vdash (\Gamma'; \Sigma'_{12}), x_1{:}\tau_1, x_2{:}\tau_2 \rightsquigarrow \Gamma''; \Sigma''_{12} \qquad \Gamma''; \Sigma''_{12} \rhd e_1 : \tau\end{array}}{\Gamma'; \Sigma'_{11}, \Sigma'_{12} \rhd \mathsf{let}\ \langle x_1, x_2 \rangle = e'\ \mathsf{in}\ e_1 : \tau}$**.**

As in the T-CASE case.

Case $\dfrac{\Gamma; \Sigma \rhd e_1 : \tau_1}{\Gamma; \Sigma \rhd \mathsf{new}\ e_1 : \mathsf{ref}\ \tau_1}$**.**

By the induction hypothesis and rule T-NEW.

Case $\dfrac{\Gamma;\Sigma_1 \rhd e_1 : \mathsf{ref}\ \tau_1 \qquad \Gamma;\Sigma_2 \rhd e_2 : \tau_2}{\Gamma;\Sigma_1,\Sigma_2 \rhd \mathsf{swap}\ e_1\ e_2 : \mathsf{ref}\ \tau_2 \otimes \tau_1}$**.**

As in the T-APP case.

Case $\dfrac{\Gamma;\Sigma \rhd e_1 : \mathsf{ref}\ \tau_1}{\Gamma;\Sigma \rhd \mathsf{delete}\ e_1 : 1}$**.**

By the induction hypothesis and rule T-DELETE. □

LEMMA A.29 (Replacement)**.**

If $\bullet;\Sigma \rhd E[e] : \tau$ *then there exist some contexts* Σ_1 *and* Σ_2 *and some type* τ' *such that*

- $\bullet;\Sigma_1 \rhd e : \tau'$ *and*
- $\bullet;\Sigma'_1,\Sigma_2 \rhd E[e'] : \tau$ *for any* e' *such that* $\bullet;\Sigma'_1 \rhd e' : \tau'$.

Proof. By induction on the structure of E:

Case []**.**

Let $\Sigma_1 = \Sigma$ and $\Sigma_2 = \bullet$.

Case $E'\ e_2$**.**

That is,

(1) $\bullet;\Sigma \rhd E'[e]\ e_2 : \tau$.

By inversion of rule T-APP,

(2) $\bullet;\Sigma'_1 \rhd E'[e] : \tau_2 \xrightarrow{\xi}_{\circ} \tau$ and

(3) $\bullet;\Sigma'_2 \rhd e_2 : \tau_2$

for some Σ'_1, Σ'_2, ξ, and τ_2. By the induction hypothesis at E', there exist some contexts Σ_1 and Σ'_{12} and some type τ' such that

(4) $\bullet;\Sigma_1 \rhd e : \tau'$ and

(5) $\bullet;\Sigma'_1,\Sigma'_{12} \rhd E'[e'] : \tau_2 \xrightarrow{\xi}_{\circ} \tau$.

Let $\Sigma_2 = \Sigma'_{12}, \Sigma'_2$. Then by rule T-APP,

(6) $\bullet; \Sigma'_1, \Sigma'_{12}, \Sigma'_2 \rhd E'[e']\, e_2 : \tau$.

Case $v_1\, E'$**.**

As in the previous case, *mutatis mutandem*.

Case $E'\, \tau_2$**.**

That is,

(1) $\bullet; \Sigma \rhd E'[e]\, \tau_2 : \{\tau_2/\alpha\}\tau_1$

where $\tau = \{\tau_2/\alpha\}\tau_1$. By inversion of rule T-TAPP,

(2) $\bullet; \Sigma \rhd E'[e] : \forall \alpha{:}\kappa.\, \tau_1$ and

(3) $\bullet \vdash \tau_2 : \kappa$.

By the induction hypothesis at E', there exist some contexts Σ_1 and Σ_2 and some type τ' such that

(4) $\bullet; \Sigma_1 \rhd e : \tau''$ and

(5) $\bullet; \Sigma'_1, \Sigma_2 \rhd E'[e'] : \forall \alpha{:}\kappa.\, \tau_1$.

Then by rule T-TAPP,

(6) $\bullet; \Sigma'_1, \Sigma_2 \rhd E'[e']\, \tau_2 : \{\tau_2/\alpha\}\tau_1$.

Case $\mathsf{fix}\, E'$**.**

As in the previous case.

Case $\mathsf{inl}\, E'$**.**

As in the previous case.

Case $\mathsf{inr}\, E'$**.**

As in the previous case.

Case $\mathsf{case}\, E'\, \mathsf{of}\, \mathsf{inl}\, x_1 \to e_1; \mathsf{inr}\, x_2 \to e_2$**.**

As in the previous case.

Case $\langle E', e_2 \rangle$**.**

As in the $E'\, e_2$ case.

Case $\langle v_1, E' \rangle$**.**

As in the $v_1 E'$ case.

Case $\mathsf{let}\, \langle x_1, x_2 \rangle = E'\, \mathsf{in}\, e_1$**.**

As in the $E'\, \tau$ case.

Case $\mathsf{new}\, E'$**.**

As in the $E'\, \tau$ case.

Case $\mathsf{swap}\, E'\, e_2$**.**

As in the $E'\, e_2$ case.

Case $\mathsf{swap}\, v_1 E'$**.**

As in the $v_1 E'$ case.

Case $\mathsf{delete}\, E'$**.**

As in the $E'\, \tau$ case. □

LEMMA 5.8 (Preservation, restated from p. 121)**.**

If $\triangleright (s, e) : \tau$ *and* $(s, e) \longmapsto (s', e')$ *then* $\triangleright (s', e') : \tau$.

Proof. Without loss of generality, we consider only the case of rule CXT, $(s, E[e]) \longmapsto (s', E[e'])$, where $(s, e) \longmapsto (s', e')$ not by rule CXT. (All derivations may have exactly one instance of rule CXT at the root because the empty context is an evaluation context and the composition of two evaluation contexts is an evaluation context.)

By inversion of rule CONF, there must be some Σ_1 and Σ_2 such that $\Sigma_1 \rhd s : \Sigma_1, \Sigma_2$ and $\bullet; \Sigma_2 \rhd E[e] : \tau$. Then by lemma A.29, there are some τ', Σ_{21}, and Σ_{22} such that:

- $\bullet; \Sigma_{21} \rhd e : \tau'$ and
- $\bullet; \Sigma'_{21}, \Sigma_{22} \rhd E[e''] : \tau$ for any e'' such that $\bullet; \Sigma'_{21} \rhd e'' : \tau'$.

In cases where $s = s'$, it is sufficient to show that $\bullet; \Sigma_{21} \rhd e' : \tau'$, which allows us to replace e with e' and reconstruct the same configuration typing. For cases where $s \neq s'$, we will need to rederive the configuration typing using the new store.

We proceed by cases on the reduction relation, in each case inverting the typing relation. We need not consider the non–syntax-directed rules T-SUBSUME and T-WEAK:

- If the final rule is T-SUBSUME, then there must be some τ'' such that $\bullet; \Sigma_{21} \rhd e : \tau''$ and $\bullet \vdash \tau'' <:^+ \tau'$. If we can show that τ'' is preserved, then we can reapply rule T-SUBSUME to get τ'.
- If the final rule is T-WEAK, we can push it upward in the derivation—and thus ignore it—unless we are typing an abstraction, since rule T-ABS is the only rule affected by unused elements in the affine environment.

Now by cases on the reduction relation:

Case $(s, (\lambda x{:}\tau_2.\, e_1)\, v_2) \longmapsto (s, \{v_2/x\}\, e_1)$**.**

By inversion of rule T-APP, there exist some contexts Σ_{211} and Σ_{212} and some qualifier expression ξ such that

(1) $\bullet; \Sigma_{211} \rhd \lambda x{:}\tau_2.\, e_1 : \tau_2 \xrightarrow{\xi}\!\!\circ\, \tau'$ and

(2) $\bullet; \Sigma_{212} \rhd v_2 : \tau_2$.

Without loss of generality, split Σ_{21} so that Σ_{211} contains the bare minimum to type $\lambda x{:}\tau_2.\, e_1$, so that typing the abstraction does not require weakening.

By inversion of rule T-ABS,

(3) $\vdash (\bullet;\Sigma_{211}), x{:}\tau_2 \rightsquigarrow \Gamma';\Sigma'_{211}$,

(4) $\Gamma';\Sigma'_{211} \rhd e_1 : \tau'$,

(5) $\bullet \vdash \Sigma_{211} \preceq \xi$, and

(6) $\bullet \vdash \tau_2 : \xi_2$.

By lemma 5.6, $\bullet;\Sigma_{211},\Sigma_{212} \rhd \{v_2/x\}\, e_1 : \tau'$.

Case $(s,(\Lambda\alpha{:}\kappa.\, v)\,\tau_1) \longmapsto (s,\{\tau_1/\alpha\}\, v)$**.**

By inversion of rule T-TAPP, there exists some type τ_2 such that

(1) $\bullet \vdash \tau_1 : \kappa$ and

(2) $\bullet;\Sigma_{21} \rhd \Lambda\alpha{:}\kappa.\, v : \forall\alpha{:}\kappa.\, \tau_2$

where $\tau' = \{\tau_1/\alpha\}\,\tau_2$. Then by inversion of rule T-TABS,

(3) $\alpha{:}\kappa;\Sigma_{21} \rhd v : \tau_2$.

By lemma A.21, $\bullet;\Sigma_{21} \rhd \{\tau_1/\alpha\}\, v : \{\tau_1/\alpha\}\,\tau_2$.

Case $(s, \mathsf{fix}\, v_1\, v_2) \longmapsto (s, v_1\,(\mathsf{fix}\, v_1)\, v_2)$**.**

By inversion of rule T-APP, there exist some contexts Σ_{211} and Σ_{212} and some qualifier expression ξ such that

(1) $\bullet;\Sigma_{211} \rhd \mathsf{fix}\, v_1 : \tau_2 \xrightarrow{\xi}_{\circ} \tau'$ and

(2) $\bullet;\Sigma_{212} \rhd v_2 : \tau_2$.

It suffices to show that $\bullet;\Sigma_{211} \rhd v_1\,(\mathsf{fix}\, v_1) : \tau_2 \xrightarrow{\xi}_{\circ} \tau'$ as well.

By inversion of rule T-FIX,

(3) $\bullet;\Sigma_{211} \rhd v_1 : (\tau_2 \xrightarrow{\xi}_{\circ} \tau') \xrightarrow{\cup}_{\circ} \tau_2 \xrightarrow{\xi}_{\circ} \tau'$.

Because Σ_{211} came from the store typing of s, and a store typing does not contain variable bindings, we can apply lemma 5.5 and strengthen to get

(4) $\bullet;\bullet \rhd v_1 : (\tau_2 \xrightarrow{\xi}_{\circ} \tau') \xrightarrow{\cup}_{\circ} \tau_2 \xrightarrow{\xi}_{\circ} \tau'$.

Then,

$$\dfrac{\dfrac{\dfrac{(4)}{\bullet;\bullet \rhd \mathsf{fix}\, v_1 : \tau_2 \xrightarrow{\xi}_{\circ} \tau'}\ \text{T-FIX} \quad (4)}{\bullet;\bullet \rhd v_1(\mathsf{fix}\, v_1) : \tau_2 \xrightarrow{\xi}_{\circ} \tau'}\ \text{T-APP} \qquad \vdash \bullet;\Sigma_{211}}{\bullet;\Sigma_{211} \rhd v_1(\mathsf{fix}\, v_1) : \tau_2 \xrightarrow{\xi}_{\circ} \tau'}\ \text{T-WEAK.}$$

Case $(s, \mathsf{case}\ \mathsf{inl}\ v\ \mathsf{of}\ \mathsf{inl}\ x_1 \to e_1; \mathsf{inr}\ x_2 \to e_2) \longmapsto (s, \{v/x_1\}e_1)$**.**

By inversion of rule T-CASE, there exist some contexts Σ_{211} and Σ_{212} and some types τ_1 and τ_2 such that

(1) $\bullet;\Sigma_{211} \rhd \mathsf{inr}\ v : \tau_1 \oplus \tau_2$,

(2) $\vdash (\bullet;\Sigma_{212}), x_1{:}\tau_1 \rightsquigarrow \Gamma_1;\Sigma_{2121}$,

(3) $\Gamma_1;\Sigma_{2121} \rhd e_1 : \tau'$,

(4) $\vdash (\bullet;\Sigma_{212}), x_2{:}\tau_2 \rightsquigarrow \Gamma_2;\Sigma_{2122}$, and

(5) $\Gamma_2;\Sigma_{2122} \rhd e_2 : \tau'$.

By inversion of rule T-INL, $\bullet;\Sigma_{211} \rhd v : \tau_1$.

By lemma 5.6, $\bullet;\Sigma_{211},\Sigma_{212} \rhd \{v/x_1\}e_1 : \tau'$.

Case $(s, \mathsf{case}\ \mathsf{inr}\ v\ \mathsf{of}\ \mathsf{inl}\ x_1 \to e_1; \mathsf{inr}\ x_2 \to e_2) \longmapsto (s, \{v/x_2\}e_2)$**.**

As in the previous case.

Case $(s, \mathsf{let}\ \langle x_1, x_2\rangle = \langle v_1, v_2\rangle\ \mathsf{in}\ e) \longmapsto (s, \{v_1/x_1\}\{v_2/x_2\}e)$**.**

By inversion of rule T-CASE, there exist some contexts Σ_{211} and Σ_{212} and some types τ_1 and τ_2 such that

(1) $\bullet;\Sigma_{211} \rhd \langle v_1, v_2\rangle : \tau_1 \otimes \tau_2$,

(2) $\vdash (\bullet;\Sigma_{212}), x_1{:}\tau_1, x_2{:}\tau_2 \rightsquigarrow \Gamma';\Sigma'_{212}$, and

(3) $\Gamma';\Sigma'_{212} \rhd e_1 : \tau'$.

By inversion of rule T-PAIR, there exist some contexts Σ_{2111} and Σ_{2112} such that

(4) $\bullet; \Sigma_{2111} \rhd v_1 : \tau_1$ and

(5) $\bullet; \Sigma_{2112} \rhd v_2 : \tau_2$.

Now consider $\vdash (\bullet; \Sigma_{212}), x_1{:}\tau_1, x_2{:}\tau_2 \rightsquigarrow \Gamma'; \Sigma'_{212}$. This must be derived by either rule X-ConsA or X-ConsU. By cases:

Case X-ConsA**.**

Then $\vdash (\bullet; \Sigma_{212}, x_1{:}\tau_1), x_2{:}\tau_2 \rightsquigarrow \Gamma'; \Sigma'_{212}$. By lemma 5.6,

(1) $\bullet; \Sigma_{2112}, \Sigma_{212}, x_1{:}\tau_1 \rhd \{v_2/x_2\}\, e_1 : \tau'$.

Since $\vdash (\bullet; \Sigma_{2112}, \Sigma_{212}), x_1{:}\tau_1 \rightsquigarrow \bullet; \Sigma_{2112}, \Sigma_{212}, x_1{:}\tau_2$, by lemma 5.6 again,

(2) $\bullet; \Sigma_{2111}, \Sigma_{2112}, \Sigma_{212} \rhd \{v_1/x_1\}\{v_2/x_2\}\, e_1 : \tau'$.

Case X-ConsU**.**

Then $\vdash (x_1{:}\tau_1; \Sigma_{212}), x_2{:}\tau_2 \rightsquigarrow \Gamma'; \Sigma'_{212}$, and by rule T-Weak,

(1) $x_1{:}\tau_1; \Sigma_{2112} \rhd v_2 : \tau_2$.

By lemma 5.6,

(2) $x_1{:}\tau_1; \Sigma_{2112}, \Sigma_{212} \rhd \{v_2/x_2\}\, e_1 : \tau'$.

Since $\vdash (\bullet; \Sigma_{2112}, \Sigma_{212}), x_1{:}\tau_1 \rightsquigarrow x_1{:}\tau_2; \Sigma_{2112}, \Sigma_{212}$, then by lemma 5.6 again,

(3) $\bullet; \Sigma_{2111}, \Sigma_{2112}, \Sigma_{212} \rhd \{v_1/x_1\}\{v_2/x_2\}\, e_1 : \tau'$.

Case $(s, \mathsf{new}\ v) \longmapsto (s \uplus \{\ell \mapsto v\}, \ell)$**.**

By rule T-New, $\tau' = \mathsf{ref}\ \tau''$, where $\bullet; \Sigma_{21} \rhd v : \tau''$. Then by rule S-Cons,

(1) $\Sigma_1, \Sigma_{21} \rhd s \uplus \{\ell \mapsto v\} : \Sigma_1, \Sigma_{21}, \Sigma_{22}, \ell{:}\tau''$,

and by rule T-Ptr,

(2) $\bullet; \ell{:}\tau'' \rhd \ell : \mathsf{ref}\ \tau''$.

By lemma A.29,

(3) $\bullet; \ell{:}\tau'', \Sigma_{22} \rhd E[\ell] : \tau$,

and by rule CONF,

(4) $\triangleright (s \uplus \{\ell \mapsto v\}, E[\ell]) : \tau$.

Case $(s_1 \uplus \{\ell \mapsto v_1\}, \mathsf{swap}\ \ell\ v_2) \longrightarrow (s_1 \uplus \{\ell \mapsto v_2\}, \langle \ell, v_1 \rangle)$**.**

By inversion of rules T-SWAP and T-PTR,

(1) $\bullet; \Sigma'_{211}, \ell{:}\tau_1 \triangleright \ell : \mathsf{ref}\ \tau_1$ and

(2) $\bullet; \Sigma_{212} \triangleright v_2 : \tau_2$ where

(3) $\Sigma_{21} = \Sigma'_{211}, \ell{:}\tau_1, \Sigma_{212}$ and

(4) $\tau' = \mathsf{ref}\ \tau_2 \otimes \tau_1$.

Since $s = s_1 \uplus \{\ell \mapsto v_1\}$, we have that

(5) $\Sigma_1 \triangleright s_1 \uplus \{\ell \mapsto v_1\} : \Sigma_1, \Sigma'_{211}, \Sigma_{212}, \Sigma_{22}, \ell{:}\tau_1$.

By inversion of rule S-CONS,

(6) $\Sigma_{11} \triangleright s_1 : \Sigma_{11}, \Sigma_{12}, \Sigma'_{211}, \Sigma_{212}, \Sigma_{22}$ and

(7) $\bullet; \Sigma_{12} \triangleright v_1 : \tau_1$.

Then by rule S-CONS again,

(8) $\Sigma_{11}, \Sigma_{212} \triangleright s_1 \uplus \{\ell \mapsto v_2\} : \Sigma_{11}, \Sigma_{12}, \Sigma'_{211}, \Sigma_{212}, \Sigma_{22}, \ell{:}\tau_2$.

By rule T-PTR,

(9) $\bullet; \Sigma'_{211}, \ell{:}\tau_2 \triangleright \ell : \mathsf{ref}\ \tau_2$,

and by rule T-PAIR,

(10) $\bullet; \Sigma_{12}, \Sigma'_{211}, \ell{:}\tau_2 \triangleright \langle \ell, v_1 \rangle : \mathsf{ref}\ \tau_2 \otimes \tau_1$.

By lemma A.29,

(11) $\bullet; \Sigma_{12}, \Sigma_{22}, \Sigma'_{211}, \ell{:}\tau_2 \triangleright E[\langle \ell, v_1 \rangle] : \tau$,

and by rule CONF,

(12) $\triangleright (s_1 \uplus \{\ell \mapsto v_2\}, E[\langle \ell, v_1 \rangle]) : \tau$.

Case $(s' \uplus \{\ell \mapsto v\}, \mathsf{delete}\ \ell) \longrightarrow (s', \langle\rangle)$**.**

By inversion of rule T-DELETE,

(1) $\bullet; \Sigma_{21} \triangleright \ell : \mathsf{ref}\ \tau''$

for some type τ''; by inversion of rule T-PTR, $\ell{:}\tau'' \in \Sigma_{21}$. Without loss of generality, let $\Sigma'_{21}, \ell{:}\tau'' = \Sigma_{21}$.

Since $s = s' \uplus \{\ell \mapsto v\}$, we have that

(2) $\Sigma_1 \triangleright s' \uplus \{\ell \mapsto v\} : \Sigma_1, \Sigma'_{21}, \Sigma_{22}, \ell{:}\tau''$.

Then by inversion of rule S-CONS, we have that

(3) $\Sigma_{11} \triangleright s' : \Sigma_1, \Sigma'_{21}, \Sigma_{22}$.

By rule T-UNIT,

(4) $\bullet; \Sigma_{12}, \Sigma'_{21} \triangleright \langle\rangle : 1$,

by lemma A.29,

(5) $\bullet; \Sigma_{12}, \Sigma'_{21}, \Sigma_{22} \triangleright E[\langle\rangle] : \tau$,

and finally

(6) $\triangleright (s', \langle\rangle) : \tau$

by rule CONF. □

$\boxed{\tau_1 \Rrightarrow \tau_2}$ *(parallel reduction)*

$$\text{PR-REFL}\ \frac{}{\tau \Rrightarrow \tau} \qquad \text{PR-ARR}\ \frac{\tau_{11} \Rrightarrow \tau_{21} \qquad \tau_{12} \Rrightarrow \tau_{22}}{\tau_{11} \xrightarrow{\xi_1}\!\circ\ \tau_{12} \Rrightarrow \tau_{21} \xrightarrow{\xi_2}\!\circ\ \tau_{22}} \qquad \text{PR-ALL}\ \frac{\tau_1 \Rrightarrow \tau_2}{\forall\alpha{:}\kappa.\,\tau_1 \Rrightarrow \forall\alpha{:}\kappa.\,\tau_2}$$

$$\text{PR-ABS}\ \frac{\tau_1 \Rrightarrow \tau_2}{\lambda\alpha.\,\tau_1 \Rrightarrow \lambda\alpha.\,\tau_2} \qquad \text{PR-APP}\ \frac{\tau_{11} \Rrightarrow \tau_{21} \qquad \tau_{12} \Rrightarrow \tau_{22}}{\tau_{11}\,\tau_{12} \Rrightarrow \tau_{21}\,\tau_{22}} \qquad \text{PR-BETA}\ \frac{\tau_{11} \Rrightarrow \tau_{21} \qquad \tau_{12} \Rrightarrow \tau_{22}}{(\lambda\alpha.\,\tau_{11})\,\tau_{12} \Rrightarrow \{\tau_{22}/\alpha\}\,\tau_{21}}$$

Figure A.2: One-step parallel type reduction

A.3.1.4 Type Equivalence and Parallel Reduction

This section follows Pierce's soundness proof for F_ω (2002, p. 454).

DEFINITION A.30 (Parallel type reduction)**.**

Figure A.2 defines ***one-step parallel reduction*** ($\Rrightarrow$) *on types. I will also use* ($\Leftrightarrow$)*,* ($\Rrightarrow^*$)*, and* ($\Leftrightarrow^*$) *to denote the symmetric, transitive-reflexive, and transitive-symmetric-reflexive closures of one-step parallel reduction, respectively.*

Unlike Pierce's, my parallel reduction is coarser than type equivalence, because rule PR-ARR relates arrows with different qualifiers.

LEMMA A.31 (Parallel type reduction contains type equivalence)**.**

If $\tau \equiv \tau'$ *then* $\tau \Leftrightarrow^* \tau'$.

Proof. We give a derivation $\tau = \tau_0 \Leftrightarrow \tau_1 \Leftrightarrow \cdots \Leftrightarrow \tau_k = \tau'$, by induction on the derivation of $\tau \equiv \tau'$:

Case $\tau \equiv \tau$**.**

Let $k = 0$.

Case $\dfrac{\tau' \equiv \tau}{\tau \equiv \tau'}$**.**

By the induction hypothesis, we have a derivation $\tau' = \tau_0 \Leftrightarrow \tau_1 \Leftrightarrow \cdots \Leftrightarrow \tau_k = \tau$. Then $\tau = \tau_k \Leftrightarrow \tau_{k-1} \Leftrightarrow \cdots \Leftrightarrow \tau_0 = \tau'$ is also a valid derivation.

Case $\dfrac{\tau \equiv \tau'' \qquad \tau'' \equiv \tau'}{\tau \equiv \tau'}$.

By the induction hypothesis we have a derivation connecting τ to τ'', and by the induction hypothesis again, we have a derivation connecting τ'' to τ'. Then the concatenation is these two derivations is also a valid derivation.

Case $\dfrac{\tau_{11} \equiv \tau_{21} \qquad \tau_{12} \equiv \tau_{22}}{\tau_{11} \overset{\xi}{\circ} \tau_{12} \equiv \tau_{21} \overset{\xi}{\circ} \tau_{22}}$.

By the induction hypothesis twice, we have derivations:

(1) $\tau_{11} = \tau_0 \Leftrightarrow \tau_1 \Leftrightarrow \cdots \Leftrightarrow \tau_k = \tau_{21}$ and

(2) $\tau_{12} = \tau'_0 \Leftrightarrow \tau'_1 \Leftrightarrow \cdots \Leftrightarrow \tau'_k = \tau_{22}$.

Then there is a derivation

(3) $\tau_{11} \overset{\xi}{\circ} \tau_{12} = \tau_0 \overset{\xi}{\circ} \tau_{12} \Leftrightarrow \tau_1 \overset{\xi}{\circ} \tau_{12} \Leftrightarrow \cdots \Leftrightarrow \tau_k \overset{\xi}{\circ} \tau_{12} = \tau_{21} \overset{\xi}{\circ} \tau_{12} = \tau_{21} \overset{\xi}{\circ} \tau'_0 \Leftrightarrow \tau_{21} \overset{\xi}{\circ} \tau'_1 \Leftrightarrow \cdots \Leftrightarrow \tau_{21} \overset{\xi}{\circ} \tau'_k = \tau_{21} \overset{\xi}{\circ} \tau_{22}$.

Case $\dfrac{\tau \equiv \tau'}{\forall \alpha{:}\kappa.\, \tau \equiv \forall \alpha{:}\kappa.\, \tau'}$.

By the induction hypothesis twice, we have a derivation:

(1) $\tau = \tau_0 \Leftrightarrow \tau_1 \Leftrightarrow \cdots \Leftrightarrow \tau_k = \tau'$.

Then there is a derivation

(2) $\forall \alpha{:}\kappa.\, \tau = \forall \alpha{:}\kappa.\, \tau_0 \Leftrightarrow \forall \alpha{:}\kappa.\, \tau_1 \Leftrightarrow \cdots \Leftrightarrow \forall \alpha{:}\kappa.\, \tau_k = \forall \alpha{:}\kappa.\, \tau'$.

Case $\dfrac{\tau_1 \equiv \tau_2}{\lambda \alpha.\, \tau_1 \equiv \lambda \alpha.\, \tau_2}$.

As in the previous case.

Case $\dfrac{\tau_{11} \equiv \tau_{21} \qquad \tau_{12} \equiv \tau_{22}}{\tau_{11}\,\tau_{12} \equiv \tau_{21}\,\tau_{22}}$.

As in the arrow type case.

Case $(\lambda\alpha.\,\tau_1)\,\tau_2 \equiv \{\tau_2/\alpha\}\,\tau_1$.

Let $k = 1$, since $(\lambda\alpha.\,\tau_1)\,\tau_2 \Leftrightarrow \{\tau_2/\alpha\}\,\tau_1$. □

LEMMA A.32 (Parallel type reduction contains subtyping)**.**

If $\Gamma \vdash \tau <:^{\flat} \tau'$ *then* $\tau \Leftrightarrows^{*} \tau'$.

Proof. By induction on the subtyping derivation, using lemma A.31 for the rule TSUB-EQ case. □

LEMMA A.33 (Parallel substitution and reduction)**.**

If $\tau_1 \Rightrightarrows \tau_2$ *then* $\{\tau_1/\alpha\}\,\tau \Rightrightarrows \{\tau_2/\alpha\}\,\tau$.

Proof. By induction on the structure of τ. □

LEMMA A.34 (Type substitution on parallel reduction)**.**

If $\tau_1 \Rightrightarrows \tau_2$ *and* $\tau_1' \Rightrightarrows \tau_2'$ *then* $\{\tau_1/\alpha\}\,\tau_1' \Rightrightarrows \{\tau_2/\alpha\}\,\tau_2'$.

Proof. By induction on the derivation of $\tau_1' \Rightrightarrows \tau_2'$, with one non-trivial case:

Case $\dfrac{\tau_{11} \Rightrightarrows \tau_{21} \qquad \tau_{12} \Rightrightarrows \tau_{22}}{(\lambda\beta.\,\tau_{11})\,\tau_{12} \Rightrightarrows \{\tau_{22}/\beta\}\,\tau_{21}}$.

By the induction hypothesis twice,

(1) $\{\tau_1/\alpha\}\,\tau_{11} \Rightrightarrows \{\tau_2/\alpha\}\,\tau_{21}$ and

(2) $\{\tau_1/\alpha\}\,\tau_{12} \Rightrightarrows \{\tau_2/\alpha\}\,\tau_{22}$.

By rule PR-BETA,

(3) $(\lambda\beta.\,\{\tau_1/\alpha\}\,\tau_{11})\{\tau_1/\alpha\}\,\tau_{12} \Rightrightarrows \{\{\tau_2/\alpha\}\,\tau_{22}/\beta\}\{\tau_2/\alpha\}\,\tau_{21}$.

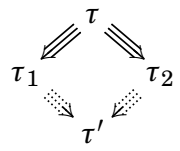
Note that $(\lambda\beta.\{\tau_1/\alpha\}\tau_{11})\{\tau_1/\alpha\}\tau_{12} = \{\tau_1/\alpha\}((\lambda\beta.\tau_{11})\tau_{12})$. Also, because β is fresh for τ_2, we know that $\{\{\tau_2/\alpha\}\tau_{22}/\beta\}\{\tau_2/\alpha\}\tau_{21} = \{\tau_2/\alpha\}\{\tau_{22}/\beta\}\tau_{21}$. Thus,

(4) $\{\tau_1/\alpha\}((\lambda\beta.\tau_{11})\tau_{12}) \Rrightarrow \{\tau_2/\alpha\}\{\tau_{22}/\beta\}\tau_{21}$

as desired. □

LEMMA A.35 (Single-step diamond property of parallel reduction)**.**

If $\tau \Rrightarrow \tau_1$ and $\tau \Rrightarrow \tau_2$ then there exists some τ' such that $\tau_1 \Rrightarrow \tau'$ and $\tau_2 \Rrightarrow \tau'$:



Proof. We start by considering cases involving rule PR-REFL, which always applies:

- If $\tau \Rrightarrow \tau_1$ by rule PR-REFL and $\tau \Rrightarrow \tau_2$ by some rule R (which may be rule PR-REFL as well), then let $\tau' = \tau_2$. Then $\tau_1 \Rrightarrow \tau'$ by rule R and $\tau_2 \Rrightarrow \tau'$ by rule PR-REFL.
- If $\tau \Rrightarrow \tau_1$ by some rule R and $\tau \Rrightarrow \tau_2$ by rule PR-REFL then by symmetry from the previous case.

We now need only consider derivations that do not involve rule PR-REFL at the root.

By induction on the structure of τ:

Case β.

This only reduces by rule PR-REFL.

Case $\lambda\beta.\tau''$.

The only two rules that allow reduction of $\lambda\beta.\tau''$ are rule PR-ABS and rule PR-REFL, and we've already considered that latter. Thus, it must be that $\tau \Rrightarrow \tau_1$ and $\tau \Rrightarrow \tau_2$ both by rule PR-ABS. Then by inversion, there must be some types τ_1' and τ_2' such that

(1) $\tau_1 = \lambda\beta.\tau_1'$,

(2) $\tau_2 = \lambda\beta.\tau_2'$,

(3) $\tau'' \Rrightarrow \tau_1'$, and

(4) $\tau'' \Rrightarrow \tau_2'$.

By the induction hypothesis, there exists some τ''' such that

(5) $\tau_1' \Rrightarrow \tau'''$ and

(6) $\tau_2' \Rrightarrow \tau'''$.

Then let $\tau' = \lambda\beta.\tau'''$, and both τ_1 and τ_2 reduce to τ''' by rule PR-ABS.

Case $\tau_1'\,\tau_2'$.

Other that rule PR-REFL, there are two rules that might apply here in any combination, PR-APP and PR-BETA.

- If $\tau \Rrightarrow \tau_1$ and $\tau \Rrightarrow \tau_2$ both by rule PR-APP, that is,

$$\frac{\tau_1' \Rrightarrow \tau_{11} \qquad \tau_2' \Rrightarrow \tau_{12}}{\tau_1'\,\tau_2' \Rrightarrow \tau_{11}\,\tau_{12}} \quad \text{and} \quad \frac{\tau_1' \Rrightarrow \tau_{21} \qquad \tau_2' \Rrightarrow \tau_{22}}{\tau_1'\,\tau_2' \Rrightarrow \tau_{21}\,\tau_{22}}$$

  where $\tau_1 = \tau_{11}\,\tau_{12}$ and $\tau_2 = \tau_{21}\,\tau_{22}$.

  By the induction hypothesis twice, there exist some types τ_1'' and τ_2'' such that

  (1) $\tau_{11} \Rrightarrow \tau_1''$,

  (2) $\tau_{21} \Rrightarrow \tau_1''$,

  (3) $\tau_{12} \Rrightarrow \tau_2''$, and

  (4) $\tau_{22} \Rrightarrow \tau_2''$.

  Then let $\tau' = \tau_1''\,\tau_2''$, and both τ_1 and τ_2 reduce to τ' by rule PR-APP.

- If $\tau \Rrightarrow \tau_1$ by rule PR-APP and $\tau \Rrightarrow \tau_2$ by rule PR-BETA, that is,

$$\frac{\tau_1'' \Rrightarrow \tau_{11} \qquad \tau_2' \Rrightarrow \tau_{12}}{(\lambda\alpha.\tau_1'')\,\tau_2' \Rrightarrow (\lambda\alpha.\tau_{11})\,\tau_{12}} \quad \text{and} \quad \frac{\tau_1'' \Rrightarrow \tau_{21} \qquad \tau_2' \Rrightarrow \tau_{22}}{(\lambda\alpha.\tau_1'')\,\tau_2' \Rrightarrow \{\tau_{22}/\alpha\}\,\tau_{21}}$$

  where

(5) $\tau_1' = \lambda\alpha.\tau_1''$,

(6) $\tau_1 = (\lambda\alpha.\tau_{11})\tau_{12}$, and

(7) $\tau_2 = \{\tau_{22}/\alpha\}\tau_{21}$.

By the induction hypothesis, twice, there exist some τ_1''' and τ_2''' such that

(8) $\tau_{11} \Rrightarrow \tau_1'''$,

(9) $\tau_{21} \Rrightarrow \tau_1'''$,

(10) $\tau_{12} \Rrightarrow \tau_2'''$, and

(11) $\tau_{22} \Rrightarrow \tau_2'''$.

Then by rule PR-BETA and lemma A.34,

(12) $(\lambda\alpha.\tau_{11})\tau_{12} \Rrightarrow \{\tau_2'''/\alpha\}\tau_1'''$ and

(13) $\{\tau_{22}/\alpha\}\tau_{21} \Rrightarrow \{\tau_2'''/\alpha\}\tau_1'''$.

- If $\tau \Rrightarrow \tau_1$ by rule PR-BETA and $\tau \Rrightarrow \tau_2$ by rule PR-APP, then by symmetry from the previous case.
- If $\tau \Rrightarrow \tau_1$ and $\tau \Rrightarrow \tau_2$ both by rule PR-BETA, that is,

$$\frac{\tau_1'' \Rrightarrow \tau_{11} \qquad \tau_2' \Rrightarrow \tau_{12}}{(\lambda\alpha.\tau_1'')\tau_2' \Rrightarrow \{\tau_{12}/\alpha\}\tau_{11}} \quad \text{and} \quad \frac{\tau_1'' \Rrightarrow \tau_{21} \qquad \tau_2' \Rrightarrow \tau_{22}}{(\lambda\alpha.\tau_1'')\tau_2' \Rrightarrow \{\tau_{22}/\alpha\}\tau_{21}}$$

where

(14) $\tau_1' = \lambda\alpha.\tau_1''$,

(15) $\tau_1 = \{\tau_{12}/\alpha\}\tau_{11}$, and

(16) $\tau_2 = \{\tau_{22}/\alpha\}\tau_{21}$.

By the induction hypothesis, twice, there exist some τ_1''' and τ_2''' such that

(17) $\tau_{11} \Rrightarrow \tau_1'''$,

(18) $\tau_{21} \Rrightarrow \tau_1'''$,

(19) $\tau_{12} \Rrightarrow \tau_2'''$, and

(20) $\tau_{22} \Rrightarrow \tau_2'''$.

Then by lemma A.34 twice,

(21) $\{\tau_{12}/\alpha\}\tau_{11} \Rrightarrow \{\tau_2'''/\alpha\}\tau_1'''$ and

(22) $\{\tau_{22}/\alpha\}\tau_{21} \Rrightarrow \{\tau_2'''/\alpha\}\tau_1'''$.

Case $\tau_1' \xrightarrow{\xi}\!\!\circ\ \tau_2'$**.**

As in the both-by-PR-APP part of the previous case, but using rule PR-ARR.

Case $\forall\beta{:}\kappa.\tau'$**.**

As in the PR-ABS case, but using rule PR-ALL.

Case χ**.**

This only reduces by rule PR-REFL. □

LEMMA A.36 (Parallel reduction confluence)**.**

If $\tau \Rrightarrow^* \tau_1$ *and* $\tau \Rrightarrow^* \tau_2$ *then there exists some* τ' *such that* $\tau_1 \Rrightarrow^* \tau'$ *and* $\tau_2 \Rrightarrow^* \tau'$.

Proof. First by induction on length of the reduction sequence for $\tau \Rrightarrow^* \tau_1$:

Case $\tau \Rrightarrow^0 \tau_1$**.**

That is, $\tau = \tau_1$. Then let $\tau' = \tau_2$, because $\tau_1 \Rrightarrow^* \tau_2$.

Case $\tau \Rrightarrow \tau_1' \Rrightarrow^k \tau_1$**.**

We would like to show that there exists some τ'' such that $\tau_1' \Rrightarrow^* \tau''$ and $\tau_2 \Rrightarrow \tau''$. By induction on length of the reduction sequence for $\tau \Rrightarrow^* \tau_2$:

Case $\tau \Rrightarrow^0 \tau_2$**.**

That is, $\tau = \tau_2$. Then let $\tau'' = \tau_1'$, because $\tau \Rrightarrow \tau_1'$ and $\tau_1' \Rrightarrow^* \tau_1'$.

Case $\tau \Rrightarrow \tau_2' \Rrightarrow^j \tau_2$**.**

Because $\tau \Rrightarrow \tau_1'$ and $\tau \Rrightarrow \tau_2'$, there exists some τ_{12}' such that $\tau_1' \Rrightarrow \tau_{12}'$ and $\tau_2' \Rrightarrow \tau_{12}'$, by lemma A.35.

Now we have that $\tau_2' \Rrightarrow \tau_{12}'$ and $\tau_2' \Rrightarrow^j \tau_2$. Since that reduction sequence is shorter than the current case, we can apply the inner induction hypothesis, by which there exists some τ'' such that $\tau_{12}' \Rrightarrow^* \tau''$ and $\tau_2 \Rrightarrow \tau''$. By transitivity, $\tau_1' \Rrightarrow \tau_{12}' \Rrightarrow^* \tau''$.

Then by the outer induction hypothesis, there exists some τ' such that $\tau_1 \Rrightarrow^* \tau'$ and $\tau'' \Rrightarrow^* \tau'$. Since $\tau \Rrightarrow \tau_1'$ and $\tau_2 \Rrightarrow \tau''$, we therefore have:

(1) $\tau \Rrightarrow \tau_1' \Rrightarrow^* \tau_1 \Rrightarrow^* \tau'$ and

(2) $\tau \Rrightarrow^* \tau_2 \Rrightarrow \tau'' \Rrightarrow^* \tau'$. □

LEMMA A.37 (Parallel reduction closure confluence)**.**

If $\tau \Leftrightarrows^* \tau'$ then there exists some type τ'' such that $\tau \Rrightarrow^* \tau''$ and $\tau' \Rrightarrow^* \tau''$.

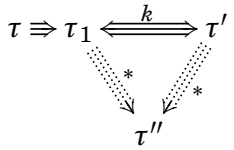
Proof. By induction on the derivation of $\tau \Leftrightarrows^* \tau'$:

Case $\tau \Leftrightarrows^0 \tau'$.

Let $\tau'' = \tau = \tau'$.

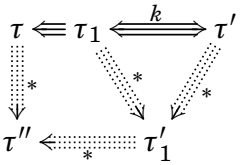
Case $\tau \Rrightarrow \tau_1 \Leftrightarrows^k \tau'$.

By the induction hypothesis, there exists some τ'' such that $\tau_1 \Rrightarrow^* \tau''$ and $\tau' \Rrightarrow^* \tau''$. Then $\tau \Rrightarrow \tau_1 \Rrightarrow^* \tau''$ as well.



Case $\tau \Lleftarrow \tau_1 \Leftrightarrows^k \tau'$.

By the induction hypothesis, there exists some τ_1'' such that $\tau_1 \Rrightarrow^* \tau_1''$ and $\tau' \Rrightarrow^* \tau_1''$. Then by lemma A.36, there exists some τ'' such that $\tau \Rrightarrow^* \tau''$ and $\tau_1'' \Rrightarrow^* \tau''$. Then $\tau' \Rrightarrow^* \tau_1'' \Rrightarrow^* \tau''$ as well.



□

COROLLARY A.38 (Subtyping confluence)**.**

If $\Gamma \vdash \tau_1 <:^{\mathfrak{v}} \tau_2$ then there exists some τ' such that $\tau_1 \Rrightarrow^* \tau'$ and $\tau_2 \Rrightarrow^* \tau'$.

Proof. By lemma A.32 and lemma A.37. □

A.3.1.5 Progress

DEFINITION A.39 (Faulty expressions)**.**

Define the ***faulty expressions with respect to store*** *s inductively as follows:*

Q_s	::=		*faulty expressions*
	|	$v\,\tau$	*where* v *is not* $\Lambda\alpha{:}\kappa.v'$
	|	$v\,v'$	*where* v *is not* $\lambda x{:}\tau.e$
	|	$\mathsf{case}\,v\,\mathsf{of}\,\mathsf{inl}\,x_1 \to e_1; \mathsf{inr}\,x_2 \to e_2$	*where* $v \notin \{\mathsf{inl}\,v_1, \mathsf{inr}\,v_2\}$
	|	$\mathsf{let}\,\langle x_1, x_2\rangle = v\,\mathsf{in}\,e$	*where* v *is not* $\langle v_1, v_2\rangle$
	|	$\mathsf{swap}\,v\,v'$	*where* v *is not* ℓ
	|	$\mathsf{delete}\,v$	*where* v *is not* ℓ
	|	$\mathsf{swap}\,\ell\,v'$	*where* $\ell \notin \mathrm{dom}\,s$
	|	$\mathsf{delete}\,\ell$	*where* $\ell \notin \mathrm{dom}\,s$
	|	$E[Q_s]$	

A ***configuration*** (s,e) ***is faulty*** *when* e *is faulty with respect to* s*.*

LEMMA A.40 (Uniform evaluation)**.**

For all configurations with closed term e*, either the configuration takes a step,* e *is a value, or the configuration is faulty.*

Proof. In each case, I will show one of:

(Q) e is faulty with respect to s,

(V) e is a value, or

(R) there is a configuration (s', e') such that $(s,e) \longmapsto (s', e')$.

By induction on e:

Case x**.**

Not closed, so it contradicts the antecedent.

Case $\lambda x{:}\tau.e_1$**.**

Then (V).

Case $e_1\,e_2$**.**

Let $E_1 = []\,e_2$. By the induction hypothesis at e_1, one of:

(Q) Then $E_1[e_1]$ is faulty as well, so (Q).

(V) Let $v_1 = e_1$ and $E_2 = v_1\,[]$. By the induction hypothesis at e_2, one of:

(Q) Then $E_2[e_2]$ is faulty as well, so (Q).

(V) Let $v_2 = e_2$. Then by cases on v_1:

Case $\lambda x{:}\tau.\,e_{11}$**.**

Then $(s,(\lambda x{:}\tau.\,e_{11})\,v_2) \longmapsto (s,\{v_2/x\}\,e_{11})$ by rule β_v, so (R).

Otherwise.

If v_1 is not an abstraction, then (Q).

(R) That is, $(s,e_2) \longmapsto (s',e_2')$. Then $(s,E_2[e_2]) \longmapsto (s',E_2[e_2'])$, so (R).

(R) That is, $(s,e_1) \longmapsto (s',e_1')$. Then $(s,E_1[e_1]) \longmapsto (s',E_1[e_1'])$, so (R).

Case $\Lambda\alpha{:}\kappa.\,v_1$**.**

Then (V).

Case $e_1\,\tau$**.**

Let $E_1 = []\,\tau$. By the induction hypothesis at e_1, one of:

(Q) Then $E_1[e_1]$ is faulty as well, so (Q).

(V) Let $v_1 = e_1$. Then by cases on v_1:

Case $\Lambda\alpha{:}\kappa.\,v_{11}$**.**

Then $(s,(\Lambda\alpha{:}\kappa.\,v_{11})\,\tau) \longmapsto (s,\{\tau/\alpha\}\,v_{11})$ by rule β_v, so (R).

Otherwise.

If v_1 is not a type abstraction, then (Q).

(R) That is, $(s,e_1) \longmapsto (s',e_1')$. Then $(s,E_1[e_1]) \longmapsto (s',E_1[e_1'])$, so (R).

Case $\mathsf{fix}\,e_1$**.**

Let $E = \mathsf{fix}[]$. By the induction hypothesis at e_1, one of:

(Q) Then $E[e_1]$ is faulty as well, so (Q).

(V) Let $v_1 = e_1$. Then (V).

(R) That is, $(s, e_1) \longmapsto (s', e_1')$. Then $(s, E[e_1]) \longmapsto (s', E_1[e_1'])$, so (R).

Case $\langle\rangle$**.**

Then (V).

Case $\mathsf{inl}\, e_1$**.**

Let $E = \mathsf{inl}\,[]$. By the induction hypothesis at e_1, one of:

(Q) Then $E[e_1]$ is faulty as well, so (Q).

(V) Let $v_1 = e_1$. Then $\mathsf{inl}\, v_1$ is a value, so (V).

(R) That is, $(s, e_1) \longmapsto (s', e_1')$. Then $(s, E[e_1]) \longmapsto (s', E_1[e_1'])$, so (R).

Case $\mathsf{inr}\, e_2$**.**

As in the previous case.

Case $\mathsf{case}\, e'\, \mathsf{of}\, \mathsf{inl}\, x_1 \to e_1; \mathsf{inr}\, x_2 \to e_2$**.**

Let $E = \mathsf{case}\,[]\, \mathsf{of}\, \mathsf{inl}\, x_1 \to e_1; \mathsf{inr}\, x_2 \to e_2$. Then by the induction hypothesis at e', one of:

(Q) Then $E_1[e_1]$ is faulty as well, so (Q).

(V) Let $v' = e'$. Then by cases on v':

Case $\mathsf{inl}\, v_1'$**.**
Then $(s, \mathsf{case}\, \mathsf{inl}\, v_1'\, \mathsf{of}\, \mathsf{inl}\, x_1 \to e_1; \mathsf{inr}\, x_2 \to e_2) \longmapsto (s, \{v_1'/x_1\}\, e_1)$ by rule CaseL, so (R).

Case $\mathsf{inr}\, v_2'$**.**
Likewise, but by rule CaseR, (R).

Otherwise.
If v' is not a sum injection, then (Q).

(R) That is, $(s, e_1) \longmapsto (s', e_1')$. Then $(s, E_1[e_1]) \longmapsto (s', E_1[e_1'])$, so (R).

Case $\langle e_1, e_2\rangle$**.**

As in the application case, with one change: If both e_1 and e_2 are values, then (V).

Case $\mathsf{let}\,\langle x_1, x_2\rangle = e'\,\mathsf{in}\,e_1$**.**

Let $E = \mathsf{let}\,\langle x_1, x_2\rangle = []\,\mathsf{in}\,e_1$. Then by the induction hypothesis at e', one of:

(Q) Then $E_1[e_1]$ is faulty as well, so (Q).

(V) Let $v' = e'$. Then by cases on v':

Case $\langle v_1, v_2\rangle$**.**

Then $(s, \mathsf{let}\,\langle x_1, x_2\rangle = \langle v_1, v_2\rangle\,\mathsf{in}\,e_1) \longmapsto (s, \{v_1/x_1\}\{v_2/x_2\}e_1)$ by rule LETPAIR, so (R).

Otherwise.

If v' is not a pair, then (Q).

(R) That is, $(s, e_1) \longmapsto (s', e_1')$. Then $(s, E_1[e_1]) \longmapsto (s', E_1[e_1'])$, so (R).

Case $\mathsf{new}\,e_1$**.**

Let $E = \mathsf{new}[]$. By the induction hypothesis at e_1, one of:

(Q) Then $E[e_1]$ is faulty as well, so (Q).

(V) Let $v_1 = e_1$. Then by rule NEW $(s, \mathsf{new}\,v_1) \longmapsto (s \uplus \{\ell \mapsto v_1\}, \ell)$, so (R).

(R) That is, $(s, e_1) \longmapsto (s', e_1')$. Then $(s, E[e_1]) \longmapsto (s', E_1[e_1'])$, so (R).

Case $\mathsf{swap}\,e_1\,e_2$**.**

As in the application case, except when both e_1 and e_2 are values. Call them v_1 and v_2. Then by cases on v_1:

Case ℓ**.**

If $\ell \in \mathrm{dom}\,s$, then let $s' \uplus \{\ell \mapsto v\} = s$, and $(s' \uplus \{\ell \mapsto v\}, \mathsf{swap}\,\ell\,v_2) \longmapsto (s' \uplus \{\ell \mapsto v_2\}, \langle \ell, v\rangle)$ by rule SWAP, so (R). Otherwise, (Q).

Otherwise.

If v_1 is not a store location, then (Q).

Case $\mathsf{delete}\,e_1$**.**

Let $E = \mathsf{delete}[]$. By the induction hypothesis at e_1, one of:

(Q) Then $E[e_1]$ is faulty as well, so (Q).

(V) Let $v_1 = e_1$. Then by cases on v_1:

Case ℓ.

If $\ell \in \mathrm{dom}\, s$, then let $s' \uplus \{\ell \mapsto v_2\} = s$, and by rule DELETE, $(s' \uplus \{\ell \mapsto v_2\}, \mathsf{delete}\ \ell) \longmapsto (s', \langle\rangle)$, so (R). Otherwise, (Q).

Otherwise.

If v_1 is not a store location, then (Q).

(R) That is, $(s, e_1) \longmapsto (s', e_1')$. Then $(s, E[e_1]) \longmapsto (s', E_1[e_1'])$, so (R).

Case ℓ.

Then (V). □

DEFINITION A.41 (Concrete types)**.**

Let $\mathscr{T}$ be the set of types and $\mathscr{K}^j$ be the set of kinds of arity j. Define the six sets of ***concrete types*** *as follows:*

$$\begin{aligned}
\mathscr{C}_{\mathrm{ARR}} &= \{\tau_1 \xrightarrow{\xi} \tau_2 \mid \tau_1, \tau_2 \in \mathscr{T}, \xi \in \mathscr{K}^0\} \\
\mathscr{C}_{\mathrm{ALL}} &= \{\forall \alpha{:}\kappa.\, \tau_1 \mid j \in \mathbb{N}, \kappa \in \mathscr{K}^j, \tau_1 \in \mathscr{T}\} \\
\mathscr{C}_{\mathrm{UNIT}} &= \{1\} \\
\mathscr{C}_{\mathrm{SUM}} &= \{\tau_1 \oplus \tau_2 \mid \tau_1, \tau_2 \in \mathscr{T}\} \\
\mathscr{C}_{\mathrm{PROD}} &= \{\tau_1 \otimes \tau_2 \mid \tau_1, \tau_2 \in \mathscr{T}\} \\
\mathscr{C}_{\mathrm{REF}} &= \{\mathsf{ref}\ \tau_1 \mid \tau_1 \in \mathscr{T}\}
\end{aligned}$$

Now define each $\mathscr{T}_i$ as the set of types that can reduce to each $\mathscr{C}_i$:

$$\mathscr{T}_i = \{\tau \mid \tau \in \mathscr{T}, \tau' \in \mathscr{C}_i, \tau \Rrightarrow^* \tau'\}$$

LEMMA A.42 (Concrete closure)**.**

If $\tau \in \mathscr{C}_i$ and $\tau \Rrightarrow^* \tau'$ then $\tau' \in \mathscr{C}_i$.

Proof. By induction on the length of the reduction sequence and cases on τ:

Case $\tau_1 \xrightarrow{\xi} \tau_2$**.**

The only rules that apply are rule PR-REFL and rule PR-ARR, neither of which changes the shape of the type.

Case $\forall \alpha{:}\kappa.\, \tau_1$**.**

The only rules that apply are rule PR-REFL and rule PR-ALL, neither of which changes the shape of the type.

Case 1**.**

The only rule that applies is rule PR-REFL, which does not change the shape of the type.

Case $(\oplus)\,\tau_1\,\tau_2$**.**

The only rules that apply are rule PR-REFL and rule PR-APP, which may change τ_1 and τ_2 but cannot change $(\oplus)$.

Case $(\otimes)\,\tau_1\,\tau_2$**.**

The only rules that apply are rule PR-REFL and rule PR-APP, which may change τ_1 and τ_2 but cannot change $(\otimes)$.

Case $\mathsf{ref}\ \tau_1$**.**

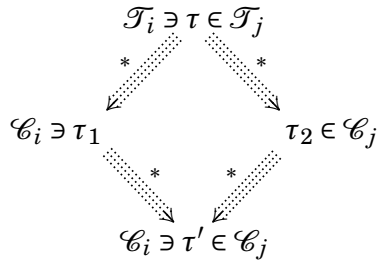
The only rules that apply are rule PR-REFL and rule PR-APP, which may change τ_1 but cannot change ref. □

COROLLARY A.43 (Partition of types)**.**

If $\tau \in \mathscr{T}_i$ *and* $\tau \in \mathscr{T}_j$ *then* $i = j$*.*

Proof. Type τ must reduce to types in both $\mathscr{C}_i$ and $\mathscr{C}_j$, which by lemma A.36 must in turn reduce to some common type τ'. By lemma A.42, τ' is in both $\mathscr{C}_i$ and $\mathscr{C}_j$, and since the six sets of concrete types are mutually disjoint, those

must be the same set.

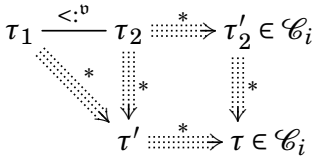


□

COROLLARY A.44 (Subtyping preserves form)**.**

If $\Gamma \vdash \tau_1 <:^{\mathfrak{v}} \tau_2$ *and* $\tau_2 \in \mathscr{T}_i$ *then* $\tau_1 \in \mathscr{T}_i$.

Proof. By the definition of $\mathscr{T}_i$, there exists some $\tau_2' \in \mathscr{C}_i$ such that $\tau_2 \Rrightarrow^* \tau_2'$. By corollary A.38, there exists some τ' such that $\tau_2 \Rrightarrow^* \tau'$ and $\tau_1 \Rrightarrow^* \tau'$. By lemma A.36, there exists some τ such that $\tau' \Rrightarrow^* \tau$ and $\tau_2' \Rrightarrow^* \tau$.



Since $\tau_2' \in \mathscr{C}_i$, by lemma A.42, $\tau \in \mathscr{C}_i$ as well. Since $\tau_1 \Rrightarrow^* \tau$, then by the definition of $\mathscr{T}_i$, $\tau_1 \in \mathscr{T}_i$. □

LEMMA A.45 (Canonical forms)**.**

The concrete type of a value dictates its form. Suppose that $\bullet; \Sigma \rhd v : \tau$.

If τ *is . . . ,*		*then* v *is . . . .*	
$\tau_1 \xrightarrow{\xi} \tau_2$	$\in \mathscr{C}_{\text{ARR}}$	$\lambda x{:}\tau'.e$	*for some* x, τ' *and* e
$\forall \alpha{:}\kappa.\tau'$	$\in \mathscr{C}_{\text{ALL}}$	$\Lambda \alpha{:}\kappa.v'$	*for some value* v'
1	$\in \mathscr{C}_{\text{UNIT}}$	$\langle\rangle$	
$\tau_1 \oplus \tau_2$	$\in \mathscr{C}_{\text{SUM}}$	$\mathsf{inl}\, v'$ *or* $\mathsf{inr}\, v'$	*for some value* v'
$\tau_1 \otimes \tau_2$	$\in \mathscr{C}_{\text{PROD}}$	$\langle v_1, v_2 \rangle$	*for some values* v_1 *and* v_2
$\mathsf{ref}\, \tau'$	$\in \mathscr{C}_{\text{REF}}$	ℓ	*for some location* ℓ

Proof. We generalize the induction hypothesis to use the sets $\mathscr{T}_i$ in place of each set $\mathscr{C}_i$. Since the $\mathscr{T}_i$ are disjoint, by corollary A.43, finding that a type is in one of the sets means we need not consider the others.

By induction on the typing derivation:

Case $\dfrac{\Gamma;\Sigma \rhd v : \tau' \qquad \Gamma \vdash \tau' <:^+ \tau \qquad \Gamma \vdash \tau : \xi}{\Gamma;\Sigma \rhd v : \tau}$.

If $\tau \in \mathscr{T}_j$, then by corollary A.44, $\tau' \in \mathscr{T}_j$ as well. Then by the induction hypothesis, v has the right form.

Case $\dfrac{\Gamma;\Sigma \rhd v : \tau \qquad \vdash \Gamma,\Gamma';\Sigma,\Sigma'}{\Gamma,\Gamma';\Sigma,\Sigma' \rhd v : \tau}$.

By the induction hypothesis.

Case $\dfrac{\ell{:}\tau' \in \Sigma \qquad \bullet \vdash \tau' : \xi \qquad \vdash \Gamma;\Sigma}{\Gamma;\Sigma \rhd \ell : \mathsf{ref}\ \tau'}$.

Then $\mathsf{ref}\ \tau' \in \mathscr{C}_{\mathrm{REF}} \subset \mathscr{T}_{\mathrm{REF}}$, and ℓ has the right form.

Case $\dfrac{\vdash (\Gamma;\Sigma), x{:}\tau_1 \rightsquigarrow \Gamma';\Sigma' \qquad \Gamma';\Sigma' \rhd e : \tau_2 \qquad \Gamma \vdash \Sigma \preceq \xi \qquad \Gamma \vdash \tau_1 : \xi_1}{\Gamma;\Sigma \rhd \lambda x{:}\tau_1.\, e : \tau_1 \xrightarrow{\xi}_{\circ} \tau_2}$.

Then $\tau_1 \xrightarrow{\xi}_{\circ} \tau_2 \in \mathscr{C}_{\mathrm{ARR}} \subset \mathscr{T}_{\mathrm{ARR}}$, and $\lambda x{:}\tau_1.\, e$ has the right form.

Case $\dfrac{\Gamma, \alpha{:}\kappa;\Sigma \rhd v' : \tau'}{\Gamma;\Sigma \rhd \Lambda\alpha{:}\kappa.\, v' : \forall\alpha{:}\kappa.\, \tau'}$.

Then $\forall\alpha{:}\kappa.\, \tau' \in \mathscr{C}_{\mathrm{ALL}} \subset \mathscr{T}_{\mathrm{ALL}}$, and $\Lambda\alpha{:}\kappa.\, v'$ has the right form.

Case $\dfrac{\vdash \Gamma;\Sigma}{\Gamma;\Sigma \rhd \langle\rangle : 1}$.

Then $1 \in \mathscr{C}_{\mathrm{UNIT}} \subset \mathscr{T}_{\mathrm{UNIT}}$, and $\langle\rangle$ has the right form.

Case $\dfrac{\Gamma;\Sigma \rhd v' : \tau_1 \qquad \Gamma \vdash \tau_2 : \xi}{\Gamma;\Sigma \rhd \mathsf{inl}\ v' : \tau_1 \oplus \tau_2}$.

Then $\tau_1 \oplus \tau_2 \in \mathscr{C}_{\mathrm{SUM}} \subset \mathscr{T}_{\mathrm{SUM}}$, and $\mathsf{inl}\ v'$ has the right form.

Case $\dfrac{\Gamma;\Sigma \rhd v' : \tau_2 \qquad \Gamma \vdash \tau_1 : \xi}{\Gamma;\Sigma \rhd \mathsf{inr}\, v' : \tau_1 \oplus \tau_2}$**.**

Then $\tau_1 \oplus \tau_2 \in \mathscr{C}_{\text{SUM}} \subset \mathscr{T}_{\text{SUM}}$, and $\mathsf{inr}\, v'$ has the right form.

Case $\dfrac{\Gamma;\Sigma_1 \rhd v_1 : \tau_1 \qquad \Gamma;\Sigma_2 \rhd v_2 : \tau_2}{\Gamma;\Sigma_1,\Sigma_2 \rhd \langle v_1, v_2\rangle : \tau_1 \otimes \tau_2}$**.**

Then $\tau_1 \otimes \tau_2 \in \mathscr{C}_{\text{PROD}} \subset \mathscr{T}_{\text{PROD}}$, and $\langle v_1, v_2\rangle$ has the right form.

Otherwise.

The remaining rules to not apply to values. □

LEMMA A.46 (Faulty expressions)**.**

If term e is faulty with respect to store s, then there is no τ such that $\rhd (s,e) : \tau$.

Proof by contradiction. Suppose that $\rhd (s,e) : \tau'$ and that e is faulty with respect to s. By inversion of rule CONF, there exist some contexts Σ_1 and Σ_2 such that

(1) $\Sigma_1 \rhd s : \Sigma_1, \Sigma_2$ and

(2) $\bullet;\Sigma_2 \rhd e : \tau'$.

It may end with some amount of subsumption and weakening, but prior to that there must be an instance of the appropriate syntax-directed rule for e, yielding

(3) $\bullet;\Sigma_{21} \rhd e : \tau$

for some Σ_{21} and τ.

Since e is faulty, let $Q_s = e$. We generalize the induction hypothesis over τ and proceed by induction on the structure of Q_s:

Case $v\,\tau_2$ where $v \not\equiv \Lambda\alpha'{:}\kappa'.v'$**.**

By inversion of rule T-TAPP, there are some α_1, κ_1, and τ_1 such that

(1) $\bullet;\Sigma_{21} \rhd v : \forall\alpha_1{:}\kappa_1.\tau_1$.

By lemma A.45, v must therefore have the form $\Lambda\alpha_1{:}\kappa_1.\,v_1$, which contradicts the side condition that $v \not\approx \Lambda\alpha'{:}\kappa'.\,v'$.

Case $v\,v'$ where $v \not\approx \lambda x''{:}\tau''.\,e''$**.**

By inversion of rule T-APP, there are some τ_1, τ_2, and ξ_1 such that
(1) $\bullet;\Sigma_{21} \rhd v : \tau_1 \xrightarrow{\xi}\circ\, \tau_2$.

By lemma A.45, v must therefore have the form $\lambda x_1{:}\tau_1.\,e_1$, which contradicts the side condition.

Case $\mathsf{case}\ v\ \mathsf{of}\ \mathsf{inl}\ x_1 \to e_1; \mathsf{inr}\ x_2 \to e_2$ where $v \notin \{\mathsf{inl}\ v'_1, \mathsf{inr}\ v'_2\}$**.**

By inversion of rule T-CASE, there are some τ_1 and τ_2 such that
(1) $\bullet;\Sigma_{21} \rhd v : \tau_1 \oplus \tau_2$.

By lemma A.45, v must therefore have either the form $\mathsf{inl}\ v_1$ or the form $\mathsf{inr}\ v_1$, both of which contradict the side condition.

Case $\mathsf{let}\ \langle x_1, x_2\rangle = v\ \mathsf{in}\ e'$ where $v \not\approx \langle v'_1, v'_2\rangle$**.**

By inversion of rule T-UNPAIR, there are some τ_1 and τ_2 such that
(1) $\bullet;\Sigma_{21} \rhd v : \tau_1 \otimes \tau_2$.

By lemma A.45, v must therefore have the form $\langle v_1, v_2\rangle$, which contradicts the side condition.

Case $\mathsf{swap}\ v\,v'$ where $v \not\approx \ell'$**.**

By inversion of rule T-SWAP, there is some τ_1 such that
(1) $\bullet;\Sigma_{21} \rhd v : \mathsf{ref}\ \tau_1$.

By lemma A.45, v must therefore have the form ℓ, which contradicts the side condition.

Case $\mathsf{delete}\ v$ where $v \not\approx \ell'$**.**

As in the previous case, but using rule T-DELETE instead of rule T-SWAP.

Case $\mathsf{swap}\ \ell\,v'$ where $\ell \notin \mathrm{dom}\, s$**.**

By inversion of rule T-SWAP, there is some τ_1 such that

(1) $\bullet; \Sigma_{21} \rhd \ell : \mathsf{ref}\ \tau_1$,

and by inversion of rule T-PTR,

(2) $\ell{:}\tau_1 \in \Sigma_{21}$.

Since $\Sigma_2 = \Sigma_{21}, \Sigma_{22}$, this means that

(3) $\ell{:}\tau_1 \in \Sigma_2$.

Recall that $\Sigma_1 \rhd s : \Sigma_1, \Sigma_2$. Without loss of generality, because contexts are identified up to permutation, let $\Sigma_2', \ell{:}\tau_1 = \Sigma_2$. In other words,

(4) $\Sigma_1 \rhd s : \Sigma_1, \Sigma_2', \ell{:}\tau_1$.

By S-CONS, this can only be the case if $s = s' \uplus \{\ell \mapsto v'\}$ for some s' and v', which contradicts the side condition that $\ell \notin \mathrm{dom}\, s$.

Case delete ℓ where $\ell \notin \mathrm{dom}\, s$**.**

As in the previous case, but using rule T-DELETE instead of rule T-SWAP.

Case $E[Q_s']$**.**

By lemma A.29, there are some typing contexts Σ_{211} and Σ_{212} and some type τ_1 such that

(1) $\bullet; \Sigma_{211} \rhd Q_s' : \tau_1$.

By weakening,

(2) $\bullet; \Sigma_{21} \rhd Q_s' : \tau_1$,

and by the induction hypothesis at Q_s' with τ_1, this cannot be so. □

LEMMA 5.7 (Progress, restated from p. 121)**.**

If $\rhd (s, e) : \tau$ *then either* e *is a value, or there exist some* s' *and* e' *such that* $(s, e) \longmapsto (s', e')$.

Proof. Please see p. 329. ◁

THEOREM 5.9 (Type soundness, restated from p. 121)**.**

If $\triangleright (\{\}, e) : \tau$ *then either* e *diverges or there exists some store* s *and value* v *such that* $(\{\}, e) \overset{*}{\longmapsto} (s, v)$ *and* $\triangleright (s, v) : \tau$.

Proof. Please see p. 330. ◁

APPENDIX B

Additional Proofs for Chapter 7

B.1 Properties of Types and Stores

In this section, I prove several properties of types and store types.

LEMMA B.1 (Type substitution on types preserves qualifiers)**.**

For any qualifier-respecting type substitution θ, $\langle \theta\tau' \rangle \sqsubseteq \langle \tau' \rangle$ *and* $\langle (\theta\mathbf{\tau}')^{\mathscr{A}} \rangle \sqsubseteq \langle (\mathbf{\tau}')^{\mathscr{A}} \rangle$.

Proof. By induction on the structure of τ' and by induction on the structure of $\mathbf{\tau}'$. □

LEMMA B.2 (Type conversion is well-behaved)**.**

1. *For any* $F_{\mathscr{C}}$ *type* $\mathbf{\tau}$, $(\mathbf{\tau}^{\mathscr{A}})^{\mathscr{C}} = \mathbf{\tau}$.
2. *For any opaque* $F^{\mathscr{A}}$ *type* ρ, $(\rho^{\mathscr{C}})^{\mathscr{A}} = \rho$.
3. *For any* $F^{\mathscr{A}}$ *type* τ, $\langle (\tau^{\mathscr{C}})^{\mathscr{A}} \rangle \sqsubseteq \langle \tau \rangle$.
4. *For any* $F^{\mathscr{A}}$ *type* τ *and opaque* $F^{\mathscr{A}}$ *type* ρ, *if* $\tau^{\mathscr{C}} = \rho^{\mathscr{C}}$ *then* $\tau = \rho$.

Proof.

1. By induction on the structure of $\mathbf{\tau}$.
2. $(\rho^{\mathscr{C}})^{\mathscr{A}} = \{\rho\}^{\mathscr{A}} = \rho$

3. By induction on the structure of τ:

   Case $\tau_1 \xrightarrow{\mathfrak{q}} \tau_2$**.**

   $\langle((\tau_1 \xrightarrow{\mathfrak{q}} \tau_2)^{\mathscr{C}})^{\mathscr{A}}\rangle = \langle(\tau_1^{\mathscr{C}} \to \tau_2^{\mathscr{C}})^{\mathscr{A}}\rangle = \langle(\tau_1^{\mathscr{C}})^{\mathscr{A}} \xrightarrow{\mathsf{U}} (\tau_2^{\mathscr{C}})^{\mathscr{A}}\rangle = \mathsf{U} \sqsubseteq \mathfrak{q}$.

   Case $\forall\, \alpha^{\mathfrak{q}}.\, \tau'$**.**

   $$\begin{aligned}
   \langle((\forall\, \alpha^{\mathfrak{q}}.\, \tau')^{\mathscr{C}})^{\mathscr{A}}\rangle &= \langle \forall\, \beta^{\mathsf{U}}.((\{\beta^{\mathsf{U}}/\alpha^{\mathfrak{q}}\}\tau')^{\mathscr{C}})^{\mathscr{A}}\rangle \\
   &= \langle(((\{\beta^{\mathsf{U}}/\alpha^{\mathfrak{q}}\}\tau')^{\mathscr{C}})^{\mathscr{A}}\rangle \\
   &\sqsubseteq \langle\{\beta^{\mathsf{U}}/\alpha^{\mathfrak{q}}\}\tau'\rangle && \text{IH} \\
   &\sqsubseteq \langle\tau'\rangle && \text{lemma B.1} \\
   &= \langle \forall\, \alpha^{\mathfrak{q}}.\, \tau'\rangle
   \end{aligned}$$

   Case $\alpha^{\mathfrak{q}}, \mathsf{ref}\ \tau', \tau_1 \otimes \tau_2$**.**

   These are opaque, so $(\tau^{\mathscr{C}})^{\mathscr{A}} = \tau$.

   Case $\{\boldsymbol{\alpha}\}$**.**

   $\langle(\{\boldsymbol{\alpha}\}^{\mathscr{C}})^{\mathscr{A}}\rangle = \langle\boldsymbol{\alpha}^{\mathscr{A}}\rangle = \langle\{\boldsymbol{\alpha}\}\rangle$.

4. Then $\rho^{\mathscr{C}} = \{\rho\}$. By inspection of the translation function, the only τ such that $\tau^{\mathscr{C}} = \{\rho\}$ is ρ. □

DEFINITION B.3 (Unlimited and affine restriction)**.**

Define the ***unlimited restriction*** *of* Γ*, written* $\Gamma|_{\mathsf{U}}$*, to be* Γ *restricted to the portion of its domain that it does not map to* affine $F^{\mathscr{A}}$ *types. Define the* ***unlimited restriction*** *of* Σ*, written* $\Sigma|_{\mathsf{U}}$*, to be* Σ *restricted to the portion of its domain that it does not map to* $F^{\mathscr{A}}$ *types, affine or unlimited.*

That is,

$$\bullet|_{\mathsf{U}} = \bullet$$

$$(\Gamma, x:\tau)|_{\mathsf{U}} = \begin{cases} \Gamma|_{\mathsf{U}}, x:\tau & \text{if } \langle\tau\rangle = \mathsf{U} \\ \Gamma|_{\mathsf{U}} & \text{if } \langle\tau\rangle = \mathsf{A} \end{cases}$$

$$(\Sigma, \ell:\boldsymbol{\tau})|_{\mathsf{U}} = \Sigma|_{\mathsf{U}}, \ell:\boldsymbol{\tau}$$

$$(\Sigma, \ell:\tau)|_{\mathsf{U}} = \Sigma|_{\mathsf{U}}$$

$$(\Sigma, \ell:[\tau]^{\ell'})|_{\mathsf{U}} = \Sigma|_{\mathsf{U}}, \ell:[\tau]^{\ell'}$$

Likewise, define the ***affine restrictions*** $d|QA$ *and* $S|QA$ *to be the remaining portions of* Γ *and* Σ*, respectively. That is,* $\Gamma = \Gamma|_U, \Gamma|_A$ *and* $\Sigma = \Sigma|_U, \Sigma|_A$ *(up to exchange).*

If $\Sigma_1|_U = \Sigma_2|_U$*, we say that* $\Sigma_1 \sim_U \Sigma_2$*, and likewise for typing contexts; clearly* $\sim_U$ *is an equivalence relation.*

LEMMA B.4 (Context splitting properties)**.**

Commutativity *If* $\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$ *then* $\Gamma \rightsquigarrow \Gamma_2 \boxplus \Gamma_1$*. Likewise, if* $\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2$ *then* $\Sigma \rightsquigarrow \Sigma_2 \boxplus \Sigma_1$*.*

Associativity *There is some* Γ_{23} *such that* $\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_{23}$ *and* $\Gamma_{23} \rightsquigarrow \Gamma_2 \boxplus \Gamma_3$ *if and only if there is some* Γ_{12} *such that* $\Gamma \rightsquigarrow \Gamma_{12} \boxplus \Gamma_3$ *and* $\Gamma_{12} \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$*. Likewise for store types.*

Absorption *If* $\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2|_U$ *then* $\Gamma_1 = \Gamma$ *and* $\Gamma_2 \sim_U \Gamma$*. Likewise for store types. As a trivial corollary, for any* Γ*,* $\Gamma \rightsquigarrow \Gamma \boxplus \Gamma|_U$*, and for any* Σ*,* $\Sigma \rightsquigarrow \Sigma \boxplus \Sigma|_U$*.*

Equivalence *For any* Γ_1 *and* Γ_2*, if there exists some* Γ *such that* $\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$*, then* $\Gamma_1 \sim_U \Gamma_2$*. Likewise for store types.*

Disjunction *For any* Γ_1 *and* Γ_2*, if there exists some* Γ *such that* $\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$*, then* $\mathrm{dom}(\Gamma_1|_A) \cap \mathrm{dom}(\Gamma_2|_A) = \varnothing$*. Likewise for store types.*

Permutation *If* $\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$ *and* $\Gamma' \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$*, then* Γ *is a permutation of* Γ'*. Likewise for store types.*

Recombination *If* $\Gamma_1 \sim_U \Gamma_2$ *and* $\mathrm{dom}\,\Gamma_1 \cap \mathrm{dom}\,\Gamma_2 = \varnothing$*, then there exists some* Γ *such that* $\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$*. Likewise for store types.*

Proof. Each case by a trivial structural induction. □

DEFINITION B.5 (Context notation)**.**

The previous lemma justifies a notational convention: For any typing contexts Γ_1 *and* Γ_2*, if there exists some* Γ *such that* $\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$*, then I may write* $\Gamma_1 \boxplus \Gamma_2$ *for* Γ*; if there exists no such* Γ*, then* $\Gamma_1 \boxplus \Gamma_2$ *is undefined. Likewise for store*

types. This notation extends from the binary case to any number of typing contexts or store types.

OBSERVATION B.6 (Context recombination)**.**

Suppose that some store type Γ *splits into several parts* $\Gamma_1 \boxplus \cdots \boxplus \Gamma_k$*. Then we know that* $\Gamma_i \sim_\cup \Gamma_j$ *for all* i *and* j*, by induction on the number of splits and lemma B.4. Likewise, we know that* $\operatorname{dom}\Gamma_i \cap \operatorname{dom}\Gamma_j = \varnothing$ *for all* $i \neq j$*, again by induction on the number of splits and lemma B.4. Then* $\Gamma_i \boxplus \Gamma_j$ *is defined for all* $i \neq j$*, and likewise for larger combinations of subcontexts. Thus, when I split a context into several parts, I am free to recombine the parts in any order or combination.*

This observation holds for store types as well.

LEMMA B.7 (Protection is free)**.**

If a store has a type, then protecting or unprotecting any part of its type preserves the typing. In particular, for any Σ_1*,* Σ_2*, and* ℓ*,*

$$\mathrm{M}; \Sigma_1 \rhd s : \Sigma_2, \Sigma_3 \qquad \Longleftrightarrow \qquad \mathrm{M}; \Sigma_1 \rhd s : \Sigma_2, [\Sigma_3]^\ell .$$

Proof. By induction on Σ_3, noting that rules RS-LocA and RS-LocAProt have the same premises. □

LEMMA B.8 (Contexts close typed terms)**.**

The free variables, type variables, and locations in a well-typed term are contained in the contexts used to type it.

1. *If* $\mathrm{M}; \Sigma; \mathbf{\Gamma} \rhd_{\mathscr{C}} \mathbf{e} : \boldsymbol{\tau}$ *then* $\mathrm{FV}(\mathbf{e}) \subseteq \operatorname{dom}\mathbf{\Gamma}$ *and* $\mathrm{FL}(\mathbf{e}) \subseteq \operatorname{dom}\Sigma$.

2. *If* $\mathrm{M}; \Sigma; \Gamma \rhd_{\mathscr{A}} e : \tau$ *then* $\mathrm{FV}(e) \subseteq \operatorname{dom}\Gamma$ *and* $\mathrm{FL}(e) \subseteq \operatorname{dom}\Sigma$.

Proof. By induction on the type rules and the definition of free locations and variables. □

B.2 Evaluation Contexts and Substitution

In this section, I prove several lemmas about terms in holes and about substitution. In lemma B.11, I show that if a well-typed term is decomposed into an evaluation context and a subterm in the hole, then the subterm types, and the evaluation context types with a suitable replacement term in the hole as well. Unlike the usual replacement theorem, lemma B.11 allows changing the typing context for the replacement term. I also prove a standard substitution lemma.

We begin, however, with an observation about how one may often ignore subsumption rule (rule RAT-SUBSUME), which is not syntax directed, when dealing with type derivations.

OBSERVATION B.9 (Subsumption and proof by inversion)**.**

Observe, first, that multiple adjacent applications of rule RAT-SUBSUME *may always be condensed into one, by the transitivity of* (<:)*. By induction, any instance of multiple adjacent subsumptions may be rewritten to have only one subsumption. Furthermore, any derivation in* $F^{\mathscr{A}}$ *that does* not *end with a subsumption may have a subsumption added at the root, by reflexivity of the subtype relation. Thus, without loss of generality, we may consider any type derivation in* $F^{\mathscr{A}}$ *to end with rule* RAT-SUBSUME*, with a* different *rule preceding it in the derivation.*

Now we consider inverting type judgments of the form $\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} e : \tau$*. The subsumption rule may always appear at the root, and in general only one or two other rules will match the syntax of e. Denote the applicable syntax-specific rule for e as rule* R*. Because we do not consider proofs with multiple adjacent subsumptions, the premise to rule* RAT-SUBSUME *must be the conclusion of a different rule. But because e is the same, only rule* R *applies!:*

$$\dfrac{\dfrac{A1 \cdots A_k}{\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} e : \tau_<}\,\mathrm{R} \qquad \tau' <: \tau}{\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} e : \tau}\,\text{RAT-SUBSUME}$$

Thus, when inverting a type judgment for the $F^{\mathscr{A}}$ *subcalculus, we may safely consider inverting the syntax-specific judgment for e at an arbitrary type* $\tau_< <: \tau$*.*

If our goal is to reconstruct a new type judgment giving τ, by subsumption it is sufficient to reconstruct a type judgment giving $\tau_<$.

NOTATION B.10 (The type of an evaluation context)**.**

The lemma relies on the notion of giving a type to an evaluation context independent of the expression in the hole. I use the following notations:

$$\begin{aligned}
\mathrm{M};\Sigma;\mathbf{\Gamma} \rhd_{\mathscr{C}} \mathbf{E} : [\boldsymbol{\tau}']\boldsymbol{\tau} &\triangleq (\forall \mathbf{e}', \Sigma')\ \mathrm{M};\Sigma';\bullet \rhd_{\mathscr{C}} \mathbf{e}' : \boldsymbol{\tau}' \Longrightarrow \mathrm{M};\Sigma \boxplus \Sigma';\mathbf{\Gamma} \rhd_{\mathscr{C}} \mathbf{E}[\mathbf{e}'] : \boldsymbol{\tau} \\
\mathrm{M};\Sigma;\mathbf{\Gamma} \rhd_{\mathscr{C}} \mathbf{E} : [\tau']\boldsymbol{\tau} &\triangleq (\forall e', \Sigma')\ \mathrm{M};\Sigma';\bullet \rhd_{\mathscr{A}} e' : \tau' \Longrightarrow \mathrm{M};\Sigma \boxplus \Sigma';\mathbf{\Gamma} \rhd_{\mathscr{C}} \mathbf{E}[e'] : \boldsymbol{\tau} \\
\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} E : [\tau']\tau &\triangleq (\forall e', \Sigma')\ \mathrm{M};\Sigma';\bullet \rhd_{\mathscr{A}} e' : \tau' \Longrightarrow \mathrm{M};\Sigma \boxplus \Sigma';\Gamma \rhd_{\mathscr{A}} E[e'] : \tau \\
\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} E : [\boldsymbol{\tau}']\tau &\triangleq (\forall \mathbf{e}', \Sigma')\ \mathrm{M};\Sigma';\bullet \rhd_{\mathscr{C}} \mathbf{e}' : \boldsymbol{\tau}' \Longrightarrow \mathrm{M};\Sigma \boxplus \Sigma';\Gamma \rhd_{\mathscr{A}} E[\mathbf{e}'] : \tau
\end{aligned}$$

LEMMA B.11 (Terms in holes are typeable and replaceable)**.**

1. *If* $\mathrm{M};\Sigma;\mathbf{\Gamma} \rhd_{\mathscr{C}} \mathbf{E}[\mathbf{e}'] : \boldsymbol{\tau}$*, then there exist some* $\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2$ *and* $\boldsymbol{\tau}'$ *such that* $\mathrm{M};\Sigma_1;\mathbf{\Gamma} \rhd_{\mathscr{C}} \mathbf{e}' : \boldsymbol{\tau}'$ *and* $\mathrm{M};\Sigma_2;\mathbf{\Gamma} \rhd_{\mathscr{C}} \mathbf{E} : [\boldsymbol{\tau}']\boldsymbol{\tau}$.

2. *If* $\mathrm{M};\Sigma;\mathbf{\Gamma} \rhd_{\mathscr{C}} \mathbf{E}[e'] : \boldsymbol{\tau}$*, then there exist some* $\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2$ *and* τ' *such that* $\mathrm{M};\Sigma_1;\bullet \rhd_{\mathscr{A}} e' : \tau'$ *and* $\mathrm{M};\Sigma_2;\mathbf{\Gamma} \rhd_{\mathscr{C}} \mathbf{E} : [\tau']\boldsymbol{\tau}$.

3. *If* $\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} E[e'] : \tau$*, then there exist some* $\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2$*,* $\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$*, and* τ' *such that* $\mathrm{M};\Sigma_1;\Gamma_1 \rhd_{\mathscr{A}} e' : \tau'$ *and* $\mathrm{M};\Sigma_2;\Gamma_2 \rhd_{\mathscr{A}} E : [\tau']\tau$.

4. *If* $\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} E[\mathbf{e}'] : \tau$*, then there exist some* $\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2$ *and* $\boldsymbol{\tau}'$ *such that* $\mathrm{M};\Sigma_1;\bullet \rhd_{\mathscr{C}} \mathbf{e}' : \boldsymbol{\tau}'$ *and* $\mathrm{M};\Sigma_2;\Gamma \rhd_{\mathscr{A}} E : [\boldsymbol{\tau}']\tau$.

In particular, if $E[e']$ is closed, then so is e' (and likewise for the other three cases).

Proof. We take the statement of the theorem as an induction hypothesis in four parts and proceed by mutual induction on the structures of $\mathbf{E}$ and E.

1. Consider first $\mathbf{E}$. Suppose $\mathbf{e}''$ and Σ' such that $\mathrm{M};\Sigma';\bullet \rhd_{\mathscr{C}} \mathbf{e}'' : \boldsymbol{\tau}'$. It suffices to show that $\mathrm{M};\Sigma_2 \boxplus \Sigma';\bullet \rhd_{\mathscr{C}} \mathbf{E}[\mathbf{e}''] : \boldsymbol{\tau}$. Then by cases:

Case [].

Then $\mathbf{E}[\mathbf{e}'] = \mathbf{e}'$.

Let $\boldsymbol{\tau}' = \boldsymbol{\tau}$, $\Sigma_1 = \Sigma$ and $\Sigma_2 = \Sigma|_{\mathsf{U}}$.

Note that $\mathbf{E}[\mathbf{e}''] = \mathbf{e}''$. Then by weakening, $\mathrm{M}; \Sigma_2 \boxplus \Sigma'; \boldsymbol{\Gamma} \rhd_{\mathscr{C}} \mathbf{e}'' : \boldsymbol{\tau}'$. Thus, $\mathrm{M}; \Sigma_2; \boldsymbol{\Gamma} \rhd_{\mathscr{C}} \mathbf{E} : [\boldsymbol{\tau}']\boldsymbol{\tau}$.

Case $\mathbf{E}'\,\mathbf{e}_2$.

This only types if

(1) $\mathrm{M}; \Sigma_1; \boldsymbol{\Gamma} \rhd_{\mathscr{C}} \mathbf{E}'[\mathbf{e}'] : \boldsymbol{\tau}_1 \to \boldsymbol{\tau}$ and

(2) $\mathrm{M}; \Sigma_2; \boldsymbol{\Gamma} \rhd_{\mathscr{C}} \mathbf{e}_2 : \boldsymbol{\tau}_1$ where

(3) $\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2$.

By the induction hypothesis, there exist some $\boldsymbol{\tau}'$, Σ_{11}, and Σ_{12} such that

(4) $\Sigma_1 \rightsquigarrow \Sigma_{11} \boxplus \Sigma_{12}$,

(5) $\mathrm{M}; \Sigma_{11}; \boldsymbol{\Gamma} \rhd_{\mathscr{C}} \mathbf{e}' : \boldsymbol{\tau}'$ and

(6) $\mathrm{M}; \Sigma_{12}; \boldsymbol{\Gamma} \rhd_{\mathscr{C}} \mathbf{E}' : [\boldsymbol{\tau}']\boldsymbol{\tau}_1 \to \boldsymbol{\tau}$.

Then

(7) $\mathrm{M}; \Sigma_{12} \boxplus \Sigma'; \boldsymbol{\Gamma} \rhd_{\mathscr{C}} \mathbf{E}'[\mathbf{e}''] : \boldsymbol{\tau}_1 \to \boldsymbol{\tau}$.

By rule RCT-APP, $\mathrm{M}; \Sigma_{12} \boxplus \Sigma_2 \boxplus \Sigma'; \boldsymbol{\Gamma} \rhd_{\mathscr{C}} \mathbf{E}'[\mathbf{e}'']\mathbf{e}_2 : \boldsymbol{\tau}$, noting that $\Sigma \rightsquigarrow \Sigma_{11} \boxplus (\Sigma_{12} \boxplus \Sigma_2)$.

Case $(\underset{\mathbf{g}f}{\Leftarrow} \tau')E'$.

This only types if

(1) $(\tau')^{\mathscr{C}} = \boldsymbol{\tau}$ and

(2) $\mathrm{M}; \Sigma; \bullet \rhd_{\mathscr{A}} E'[\mathbf{e}'] : \tau'$.

By part 4 of the induction hypothesis, there exist some $\boldsymbol{\tau}''$ and $\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2$ such that

(3) $\mathrm{M}; \Sigma_1; \bullet \rhd_{\mathscr{C}} \mathbf{e}' : \boldsymbol{\tau}''$ and

(4) $\mathrm{M}; \Sigma_2; \bullet \rhd_{\mathscr{A}} E' : [\boldsymbol{\tau}'']\tau'$.

Then,

(5) $\mathrm{M}; \Sigma_2 \boxplus \Sigma'; \bullet \rhd_{\mathscr{A}} E'[\mathbf{e}''] : \tau'$.

By rule RCT-BOUNDARY, $\mathrm{M}; \Sigma_2 \boxplus \Sigma'; \mathbf{\Gamma} \rhd_{\mathscr{C}} (\underset{\mathbf{g}f}{\Leftarrow} \tau')(E'[\mathbf{e}'']) : \boldsymbol{\tau}$.

The remaining cases are all similar.

2. The second part proceeds *mutatis mutandis*, with two notable changes:

   - The $E = []$ case is vacuous.
   - The boundary case appeals to part 3 of the induction hypothesis.

3. For the third part, suppose an e'' such that $\mathrm{M}; \Sigma'; \bullet \rhd_{\mathscr{A}} e'' : \tau'$. It suffices to show that $\mathrm{M}; \Sigma_2 \boxplus \Sigma'; \bullet \rhd_{\mathscr{A}} E[e''] : \tau$. Then by cases on E:

   Case **[]**.

   Then $E[e'] = e'$. Let $\tau' = \tau$, $\Sigma_1 = \Sigma$, $\Sigma_2 = \Sigma|_{\cup}$, $\Gamma_1 = \Gamma$, and $\Gamma_2 = \Gamma|_{\cup}$.

   Note that $E[e''] = e''$.

   Then by weakening $\mathrm{M}; \Sigma_2 \boxplus \Sigma'; \Gamma_2 \rhd_{\mathscr{A}} e'' : \tau'$.

   Case $E_1\, \tau_2$**.**

   Consider the type derivation for $E_1[e']\tau_2$. According to observation B.9, without loss of generality, there exists some $\tau_< <: \tau$ with rule RAT-TAPP concluding that $E_1[e']\tau_2$ has that type, followed by a subsumption. This can be the case only if

   (1) $\mathrm{M}; \Sigma; \Gamma \rhd_{\mathscr{A}} E_1[e'] : \forall\, \alpha^{\mathfrak{q}}.\, \tau'_<$ where

   (2) $\tau' = \{\tau_2/\alpha^{\mathfrak{q}}\}\tau'_<$ and

   (3) $\langle \tau'_< \rangle \sqsubseteq \mathfrak{q}$.

   By induction, there exist some τ', $\Sigma_1 \boxplus \Sigma_2 \leftrightsquigarrow \Sigma$, and $\Gamma_1 \boxplus \Gamma_2 \leftrightsquigarrow \Gamma$ such that

   (4) $\mathrm{M}; \Sigma_1; \Gamma_1 \rhd_{\mathscr{A}} e' : \tau'$ and

   (5) $\mathrm{M}; \Sigma_2; \Gamma_2 \rhd_{\mathscr{A}} E' : [\tau']\forall\, \alpha^{\mathfrak{q}}.\, \tau''$

   Then,

   (6) $\mathrm{M}; \Sigma_2 \boxplus \Sigma'; \Gamma_2 \rhd_{\mathscr{A}} E'[e''] : \forall\, \alpha^{\mathfrak{q}}.\, \tau''$.

By rule RAT-TApp and rule RAT-Subsume, then, $\mathrm{M}; \Sigma_2 \boxplus \Sigma'; \Gamma_2 \rhd_{\mathscr{A}} E'[e''] \tau_2 : \tau$.

Case $\mathsf{let}\, \langle y_1, y_2 \rangle = E'\, \mathsf{in}\, e_2$**.**

This only types if

(1) $\mathrm{M}; \Sigma_1; \Gamma_1 \rhd_{\mathscr{A}} E'[e'] : \tau_1 \otimes \tau_2$ and

(2) $\mathrm{M}; \Sigma_2; \Gamma_2, y_1 : \tau_1, y_2 : \tau_2 \rhd_{\mathscr{A}} e_2 : \tau_<$

for some τ_1, τ_2, $\Sigma_1 \boxplus \Sigma_2 \leftrightsquigarrow \Sigma$ and $\Gamma_1 \boxplus \Gamma_2 \leftrightsquigarrow \Gamma$.

By induction, there exist some τ', $\Sigma_{11} \boxplus \Sigma_{12} \leftrightsquigarrow \Sigma_1$, and $\Gamma_{11} \boxplus \Gamma_{12} \leftrightsquigarrow \Gamma_1$ such that

(3) $\mathrm{M}; \Sigma_{11}; \Gamma_{11} \rhd_{\mathscr{A}} e' : \tau'$ and

(4) $\mathrm{M}; \Sigma_{12}; \Gamma_{12} \rhd_{\mathscr{A}} E' : [\tau'] \tau_1 \otimes \tau_2$

Then

(5) $\mathrm{M}; \Sigma_{12} \boxplus \Sigma'; \Gamma_{12} \rhd_{\mathscr{A}} E'[e''] : \tau_1 \otimes \tau_2$.

By rule RAT-LetPair,

(6) $\mathrm{M}; \Sigma_{11} \boxplus \Sigma_2 \boxplus \Sigma'; \Gamma_{11} \boxplus \Gamma_2 \rhd_{\mathscr{A}} \mathsf{let}\, \langle y_1, y_2 \rangle = E'[e'']\, \mathsf{in}\, e_2 : \tau_<$,

noting that $\Sigma \rightsquigarrow \Sigma_{12} \boxplus (\Sigma_{11} \boxplus \Sigma_2)$.

Case $(\tau_< \underset{f\,\mathbf{g}}{\Leftarrow}) \mathbf{E}'$**.**

This only types if

(1) $\mathrm{M}; \Sigma; \bullet \rhd_{\mathscr{C}} \mathbf{E}'[e'] : \tau^{\mathscr{C}}_<$.

By part 4 of the induction hypothesis, there exist some τ' and $\Sigma_1 \boxplus \Sigma_2 \leftrightsquigarrow \Sigma$ such that

(2) $\mathrm{M}; \Sigma_1; \bullet \rhd_{\mathscr{A}} e' : \tau'$ and

(3) $\mathrm{M}; \Sigma_2; \bullet \rhd_{\mathscr{C}} \mathbf{E}' : [\tau'] \tau^{\mathscr{C}}_<$

Then

(4) $\mathrm{M}; \Sigma_2 \boxplus \Sigma'; \bullet \rhd_{\mathscr{C}} \mathbf{E}'[e''] : \tau^{\mathscr{C}}_<$.

By rule RAT-Boundary, $\mathrm{M}; \Sigma_2 \boxplus \Sigma'; \Gamma_2 \rhd_{\mathscr{A}} (\tau_< \underset{f\,\mathbf{g}}{\Leftarrow}) \mathbf{E}'[e''] : \tau_<$.

The remaining cases are all similar.

4. The proof of the fourth part follows the proof of the third, again *mutatis mutandis*, where again the hole case is vacuous and the boundary case appeals to part 1. □

The next several lemmas concern substitution types on typing contexts, types on expressions, and values on expressions.

LEMMA B.12 (Type substitution on typing contexts preserves qualifiers)**.**

If θ is a qualifier-respecting type substitution then $\langle\theta\Gamma\rangle \sqsubseteq \langle\Gamma\rangle$.

Proof. By induction on Γ with lemma B.1. □

LEMMA B.13 (Type substitution preserves context splitting)**.**

For any qualifier-respecting type substitution θ, if $\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$ then $\theta\Gamma \rightsquigarrow (\theta\Gamma_1, \Gamma_2') \boxplus (\theta\Gamma_2, \Gamma_1')$ (up to exchange), for some Γ_1' and Γ_2'.[1]

Proof. By induction on the derivation of $\Gamma_0 \rightsquigarrow \Gamma_{01} \boxplus \Gamma_{02}$:

Case $\dfrac{\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \langle\tau\rangle = \mathsf{A}}{\Gamma, x:\tau \rightsquigarrow \Gamma_1, x:\tau \boxplus \Gamma_2}$**.**

By cases on $\langle\theta\tau\rangle$:

Case U**.**

Then

$$\dfrac{\dfrac{\text{IH}}{\theta\Gamma \rightsquigarrow \theta\Gamma_1, \theta\Gamma_2'' \boxplus \theta\Gamma_2, \theta\Gamma_1''} \qquad \dfrac{\text{case}}{\langle\theta\tau\rangle = \mathsf{U}}}{\theta\Gamma, x:\theta\tau \rightsquigarrow \theta\Gamma_1, x:\theta\tau, \Gamma_2'' \boxplus \theta\Gamma_2, x:\theta\tau, \Gamma_1''.}$$

So $\Gamma_1' = \Gamma_1'', x:\theta\tau$ and $\Gamma_2' = \Gamma_2''$.

Case A**.**

Then

$$\dfrac{\dfrac{\text{IH}}{\theta\Gamma \rightsquigarrow \theta\Gamma_1, \theta\Gamma_2' \boxplus \theta\Gamma_2, \theta\Gamma_1'} \qquad \dfrac{\text{case}}{\langle\theta\tau\rangle = \mathsf{A}}}{\theta\Gamma, x:\theta\tau \rightsquigarrow \theta\Gamma_1, x:\theta\tau, \Gamma_2' \boxplus \theta\Gamma_2, \Gamma_1'.}$$

[1] Each Γ_i' (for $i \in \{1,2\}$) is the largest subcontext of Γ_i such that $\Gamma_i'|_{\mathsf{A}} = \Gamma_i'$ and $(\theta\Gamma_i')|_{\mathsf{U}} = \theta\Gamma_i'$, but this fact is not necessary, so I will not prove it.

Case $\dfrac{\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \langle\tau\rangle = \mathsf{A}}{\Gamma, x:\tau \rightsquigarrow \Gamma_1 \boxplus \Gamma_2, x:\tau}$.

By symmetry

Case $\dfrac{\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \langle\tau\rangle = \mathsf{U}}{\Gamma, x:\tau \rightsquigarrow \Gamma_1, x:\tau \boxplus \Gamma_2, x:\tau}$.

Then

$$\dfrac{\dfrac{\text{IH}}{\theta\Gamma \rightsquigarrow \theta\Gamma_1, \theta\Gamma_2' \boxplus \theta\Gamma_2, \theta\Gamma_1'} \qquad \dfrac{\text{lemma B.1}}{\langle\theta\tau\rangle = \mathsf{U}}}{\theta\Gamma, x:\theta\tau \rightsquigarrow \theta\Gamma_1, x:\theta\tau, \Gamma_2' \boxplus \theta\Gamma_2, x:\theta\tau, \Gamma_1'.}$$

Case $\bullet \rightsquigarrow \bullet \boxplus \bullet$.

Then $\Gamma_1' = \Gamma_2' = \bullet$. □

LEMMA 7.4 (Type substitution on expressions preserves types, restated from p. 182)**.**

For any qualifier-respecting type substitution θ and any Σ such that $\mathrm{FTV}(\Sigma) = \emptyset$*:*

1. *If* $\mathrm{M};\Sigma;\boldsymbol{\Gamma} \rhd_{\mathscr{C}} \mathbf{e} : \boldsymbol{\tau}$ *then* $\mathrm{M};\Sigma;\theta\boldsymbol{\Gamma} \rhd_{\mathscr{C}} \theta\mathbf{e} : \theta\boldsymbol{\tau}$.

2. *If* $\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} e : \tau$ *then* $\mathrm{M};\Sigma;\theta\Gamma \rhd_{\mathscr{A}} \theta e : \theta\tau$.

Proof. Note first that $\mathrm{FV}(\mathbf{e}) = \mathrm{FV}(\theta\mathbf{e})$ and $\mathrm{FL}(\mathbf{e}) = \mathrm{FL}(\theta\mathbf{e})$; likewise $\mathrm{FV}(e) = \mathrm{FV}(\theta e)$ and $\mathrm{FL}(e) = \mathrm{FL}(\theta e)$,

1. By induction on the structure of $\mathbf{e}$:

   Case $\Lambda\boldsymbol{\beta}.\mathbf{v}'$**.**

   By rule RCT-TABS, $\boldsymbol{\tau} = \forall\boldsymbol{\beta}.\boldsymbol{\tau}'$, so it must be the case that

   $$\dfrac{\dfrac{\mathcal{A}}{\mathrm{M};\Sigma;\boldsymbol{\Gamma} \rhd_{\mathscr{C}} \mathbf{v}' : \boldsymbol{\tau}'}}{\mathrm{M};\Sigma;\boldsymbol{\Gamma} \rhd_{\mathscr{C}} \Lambda\boldsymbol{\beta}.\mathbf{v}' : \forall\boldsymbol{\beta}.\boldsymbol{\tau}'}.$$

Then,

$$\dfrac{\dfrac{\mathcal{A}, \text{IH}}{\mathrm{M};\Sigma;\theta\boldsymbol{\Gamma} \rhd_{\mathscr{C}} \theta\mathbf{v}' : \theta\boldsymbol{\tau}'}}{\mathrm{M};\Sigma;\theta\boldsymbol{\Gamma} \rhd_{\mathscr{C}} \theta(\Lambda\boldsymbol{\beta}.\,\mathbf{v}') : \theta(\forall\,\boldsymbol{\beta}.\,\boldsymbol{\tau}')}.$$

Case $\lambda\mathbf{x}{:}\boldsymbol{\tau}''.\,\mathbf{e}'$.

By rule RCT-ABS, $\boldsymbol{\tau} = \boldsymbol{\tau}'' \to \boldsymbol{\tau}'$. So it must be the case that

$$\dfrac{\dfrac{\mathcal{A}}{\mathrm{M};\Sigma;\boldsymbol{\Gamma},\mathbf{x} : \boldsymbol{\tau}'' \rhd_{\mathscr{C}} \mathbf{e}' : \boldsymbol{\tau}'} \qquad \dfrac{\mathcal{B}}{\langle\Sigma|_{\mathrm{FL}(\lambda\mathbf{x}:\boldsymbol{\tau}''.\,\mathbf{e}')}\rangle = \mathsf{U}}}{\mathrm{M};\Sigma;\boldsymbol{\Gamma} \rhd_{\mathscr{C}} \lambda\mathbf{x}{:}\boldsymbol{\tau}''.\,\mathbf{e}' : \boldsymbol{\tau}'' \to \boldsymbol{\tau}'}.$$

Note that $\langle\Sigma|_{\mathrm{FL}(\theta\mathbf{e}')}\rangle = \mathsf{U}$.

Then,

$$\dfrac{\dfrac{\mathcal{A}, \text{IH}}{\mathrm{M};\Sigma;\theta(\boldsymbol{\Gamma},\mathbf{x} : \boldsymbol{\tau}'') \rhd_{\mathscr{C}} \theta\mathbf{e}' : \theta\boldsymbol{\tau}'} \qquad \mathcal{B}}{\mathrm{M};\Sigma;\theta\boldsymbol{\Gamma} \rhd_{\mathscr{C}} \theta(\lambda\mathbf{x}{:}\boldsymbol{\tau}''.\,\mathbf{e}') : \theta(\boldsymbol{\tau}'' \to \boldsymbol{\tau}')}.$$

Case x.

By rule RCT-VAR, it must be the case that

$$\dfrac{\mathbf{x} : \boldsymbol{\tau} \in \boldsymbol{\Gamma}}{\mathrm{M};\Sigma;\boldsymbol{\Gamma} \rhd_{\mathscr{C}} \mathbf{x} : \boldsymbol{\tau}}.$$

Since $\theta\mathbf{x} = \mathbf{x}$,

$$\dfrac{\mathbf{x} : \theta\boldsymbol{\tau} \in \theta\boldsymbol{\Gamma}}{\mathrm{M};\Sigma;\theta\boldsymbol{\Gamma} \rhd_{\mathscr{C}} \theta\mathbf{x} : \theta\boldsymbol{\tau}}.$$

Case f.

By rule RCT-MOD, it must be the case that

$$\dfrac{(\mathbf{f} : \boldsymbol{\tau} = \mathbf{v}) \in \mathrm{M} \qquad \mathrm{FTV}(\boldsymbol{\tau}) = \emptyset}{\mathrm{M};\Sigma;\boldsymbol{\Gamma} \rhd_{\mathscr{C}} \mathbf{f} : \boldsymbol{\tau}}.$$

Thus $\theta\boldsymbol{\tau} = \boldsymbol{\tau}$, and since $\theta\mathbf{f} = \mathbf{f}$,

$$\dfrac{(\mathbf{f} : \theta\boldsymbol{\tau} = \mathbf{v}) \in \mathrm{M} \qquad \mathrm{FTV}(\theta\boldsymbol{\tau}) = \emptyset}{\mathrm{M};\Sigma;\theta\boldsymbol{\Gamma} \rhd_{\mathscr{C}} \theta\mathbf{f} : \theta\boldsymbol{\tau}}.$$

Case $f^{\mathbf{g}}$.

By rule RCT-MODA, it must be the case that

$$\frac{(f : \tau' = v) \in \mathrm{M} \qquad \mathrm{FTV}(\tau') = \emptyset}{\mathrm{M}; \Sigma; \mathbf{\Gamma} \rhd_{\mathscr{C}} f^{\mathbf{g}} : (\tau')^{\mathscr{C}}},$$

where $(\tau')^{\mathscr{C}} = \boldsymbol{\tau}$.

Thus $\theta\tau' = \tau'$, and since $\theta f^{\mathbf{g}} = f^{\mathbf{g}}$,

$$\frac{(f : \theta\tau = v) \in \mathrm{M} \qquad \mathrm{FTV}(\theta\tau) = \emptyset}{\mathrm{M}; \Sigma; \theta\mathbf{\Gamma} \rhd_{\mathscr{C}} \theta f^{\mathbf{g}} : \theta((\tau')^{\mathscr{C}})}.$$

Case $\mathbf{e}_1\,\boldsymbol{\tau}_2$.

By rule RCT-TAPP, it must be the case that

$$\frac{\dfrac{\mathcal{A}}{\mathrm{M}; \Sigma; \mathbf{\Gamma} \rhd_{\mathscr{C}} \mathbf{e}_1 : \forall\boldsymbol{\beta}.\,\boldsymbol{\tau}'}}{\mathrm{M}; \Sigma; \mathbf{\Gamma} \rhd_{\mathscr{C}} \mathbf{e}_1\,\boldsymbol{\tau}_2 : \{\boldsymbol{\tau}_2/\boldsymbol{\beta}\}\boldsymbol{\tau}'},$$

where $\boldsymbol{\tau} = \{\boldsymbol{\tau}_2/\boldsymbol{\beta}\}\boldsymbol{\tau}'$.

By Barendregt's convention, we may assume that $\boldsymbol{\alpha} \neq \boldsymbol{\beta}$, and thus $\theta(\forall\boldsymbol{\beta}.\,\boldsymbol{\tau}') = \forall\boldsymbol{\beta}.\theta\boldsymbol{\tau}'$. Noting that $\{\theta\boldsymbol{\tau}_2/\boldsymbol{\beta}\} \circ \theta = \theta \circ \{\boldsymbol{\tau}_2/\boldsymbol{\beta}\}$,

$$\frac{\dfrac{\mathcal{A}, \mathrm{IH}}{\mathrm{M}; \Sigma; \theta\mathbf{\Gamma} \rhd_{\mathscr{C}} \theta\mathbf{e}_1 : \forall\boldsymbol{\beta}.\theta\boldsymbol{\tau}'}}{\mathrm{M}; \Sigma; \theta\mathbf{\Gamma} \rhd_{\mathscr{C}} \theta(\mathbf{e}_1\,\boldsymbol{\tau}_2) : \theta\{\boldsymbol{\tau}_2/\boldsymbol{\beta}\}\boldsymbol{\tau}'}.$$

Case $\mathbf{e}_1\,\mathbf{e}_2$.

By rule RCT-APP, it must be the case that

$$\frac{\dfrac{\mathcal{A}}{\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2} \qquad \dfrac{\mathcal{B}}{\mathrm{M}; \Sigma_1; \mathbf{\Gamma} \rhd_{\mathscr{C}} \mathbf{e}_1 : \boldsymbol{\tau}_2 \to \boldsymbol{\tau}} \qquad \dfrac{\mathcal{C}}{\mathrm{M}; \Sigma_2; \mathbf{\Gamma} \rhd_{\mathscr{C}} \mathbf{e}_2 : \boldsymbol{\tau}_2}}{\mathrm{M}; \Sigma; \mathbf{\Gamma} \rhd_{\mathscr{C}} \mathbf{e}_1\,\mathbf{e}_2 : \boldsymbol{\tau}}.$$

Then, by IH,

(1) $\mathrm{M}; \Sigma_1; \theta\mathbf{\Gamma} \rhd_{\mathscr{C}} \theta\mathbf{e}_1 : \theta(\boldsymbol{\tau}_2 \to \boldsymbol{\tau})$ and

(2) $\mathrm{M};\Sigma_2;\theta\mathbf{\Gamma} \rhd_{\mathscr{C}} \theta\mathbf{e}_2 : \theta\boldsymbol{\tau}_2$,

and thus $\mathrm{M};\Sigma;\theta\mathbf{\Gamma} \rhd_{\mathscr{C}} \theta(\mathbf{e}_1\,\mathbf{e}_2) : \theta\boldsymbol{\tau}$.

Case $(\underset{\mathbf{g}f}{\Leftarrow} \tau')e'$**.**

By rule RCT-BOUNDARY, it must be the case that

$$\frac{\dfrac{\mathcal{A}}{\mathrm{M};\Sigma;\bullet \rhd_{\mathscr{A}} e' : \tau'} \qquad \dfrac{\mathcal{B}}{\mathrm{FTV}(\tau') = \emptyset}}{\mathrm{M};\Sigma;\mathbf{\Gamma} \rhd_{\mathscr{C}} (\underset{\mathbf{g}f}{\Leftarrow} \tau')e' : (\tau')^{\mathscr{C}}},$$

where $(\tau')^{\mathscr{C}} = \boldsymbol{\tau}$.

Note that $\theta\tau' = \tau'$. Then,

$$\frac{\dfrac{\mathcal{A}, \text{IH (part 2)}}{\mathrm{M};\Sigma;\bullet \rhd_{\mathscr{A}} \theta e' : \theta\tau'} \qquad \dfrac{\mathcal{B}}{\mathrm{FTV}(\theta\tau') = \emptyset}}{\mathrm{M};\Sigma;\theta\mathbf{\Gamma} \rhd_{\mathscr{C}} \theta(\underset{\mathbf{g}f}{\Leftarrow} \tau')e' : \theta((\tau')^{\mathscr{C}})}.$$

Case $(\underset{\mathbf{g}f}{\Leftarrow} \tau')^{\ell}\, v'$**.**

There are three ways to type such an expression:

- If by rule RCT-BLESSED, it must be the case that

$$\frac{\dfrac{\mathcal{A}}{\mathrm{M};\Sigma_1,\Sigma_2;\bullet \rhd_{\mathscr{A}} v' : \tau'} \qquad \dfrac{\mathcal{B}}{\langle\tau'\rangle = \mathsf{A}} \qquad \dfrac{\mathcal{C}}{\mathrm{FTV}(\tau') = \emptyset}}{\mathrm{M};[\Sigma_1]^{\ell},\ell:\mathbb{B},[\Sigma_2]^{\ell};\mathbf{\Gamma} \rhd_{\mathscr{C}} (\underset{\mathbf{g}f}{\Leftarrow} \tau')^{\ell}\, v' : (\tau')^{\mathscr{C}}},$$

  where $(\tau')^{\mathscr{C}} = \boldsymbol{\tau}$ and $[\Sigma_1]^{\ell},\ell:\mathbb{B},[\Sigma_2]^{\ell} = \Sigma$.
  Note that $\theta\tau' = \tau'$. Then,

$$\frac{\dfrac{\mathcal{A}, \text{IH (part 2)}}{\mathrm{M};\Sigma_1,\Sigma_2;\bullet \rhd_{\mathscr{A}} \theta v' : \theta\tau'} \qquad \dfrac{\mathcal{B}}{\langle\theta\tau'\rangle = \mathsf{A}} \qquad \dfrac{\mathcal{C}}{\mathrm{FTV}(\theta\tau') = \emptyset}}{\mathrm{M};[\Sigma_1]^{\ell},\ell:\mathbb{B},[\Sigma_2]^{\ell};\theta\mathbf{\Gamma} \rhd_{\mathscr{C}} \theta(\underset{\mathbf{g}f}{\Leftarrow} \tau')^{\ell}\, v' : \theta(\tau')^{\mathscr{C}}}.$$

- If by rule RCT-DEFUNCT, it must be the case that

$$\frac{\dfrac{\mathcal{A}}{\langle\tau'\rangle = \mathsf{A}} \qquad \dfrac{\mathcal{B}}{\mathrm{FTV}(\tau') = \emptyset}}{\mathrm{M};[\Sigma_1]^\ell, \ell : \mathbb{D}, [\Sigma_2]^\ell; \mathbf{\Gamma} \rhd_{\mathscr{C}} (\underset{\mathbf{g}f}{\Leftarrow} \tau')^\ell \, v' : (\tau')^{\mathscr{C}}},$$

  where $(\tau')^{\mathscr{C}} = \boldsymbol{\tau}$ and $[\Sigma_1]^\ell, \ell : \mathbb{B}, [\Sigma_2]^\ell = \Sigma$.
  Note that $\theta\tau' = \tau'$. Then,

$$\frac{\dfrac{\mathcal{A}}{\langle\theta\tau'\rangle = \mathsf{A}} \qquad \dfrac{\mathcal{B}}{\mathrm{FTV}(\theta\tau') = \emptyset}}{\mathrm{M};[\Sigma_1]^\ell, \ell : \mathbb{D}, [\Sigma_2]^\ell; \theta\mathbf{\Gamma} \rhd_{\mathscr{C}} \theta(\underset{\mathbf{g}f}{\Leftarrow} \tau')^\ell \, v' : \theta(\tau')^{\mathscr{C}}}.$$

- The rule RCT-SEALED case is similar to the RCT-BLESSED case, except that Σ is not protected.

That completes the first part.

2. By induction on the structure of e. Most cases are the same as the first part, but I show several that differ non-trivially:

   Case $\lambda x{:}\tau''.\, e'$**.**

   By rule RAT-ABS, it must be the case that

$$\frac{\dfrac{\mathcal{A}}{\mathrm{M};\Sigma;\Gamma, x : \tau'' \rhd_{\mathscr{A}} e' : \tau'} \qquad \dfrac{\mathcal{B}}{\langle\Sigma|_{\mathrm{FL}(\lambda x:\tau''.\, e')}\rangle \sqcup \langle\Gamma|_{\mathrm{FV}(\lambda x:\tau''.\, e')}\rangle = \mathfrak{q}}}{\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} \lambda x{:}\tau''.\, e' : \tau'' \xrightarrow{\mathfrak{q}}\!\!\circ\, \tau'}.$$

   where $\tau = \tau'' \xrightarrow{\mathfrak{q}}\!\!\circ\, \tau'$.

   Note that $\mathrm{FV}(\lambda x{:}\tau''.\, e') = \mathrm{FV}(\theta(\lambda x{:}\tau''.\, e'))$, so

   (1) $\langle\Gamma|_{\mathrm{FV}(\theta(\lambda x:\tau''.\, e'))}\rangle = \mathfrak{q}$.

   Then by lemma B.12,

   (2) $\langle\theta\Gamma|_{\mathrm{FV}(\theta(\lambda x:\tau''.\, e'))}\rangle \sqsubseteq \mathfrak{q}$,

   so

(3) $\langle \Sigma|_{\mathrm{FL}(\theta(\lambda x{:}\tau''.e'))} \rangle \sqcup \langle \theta\Gamma|_{\mathrm{FV}(\theta(\lambda x{:}\tau''.e'))} \rangle = \mathfrak{q}'$

where $\mathfrak{q}' \sqsubseteq \mathfrak{q}$.

Then,

$$\dfrac{\dfrac{\dfrac{\mathcal{A}, \mathrm{IH}}{\mathrm{M};\Sigma;\theta(\Gamma, x:\tau'') \rhd_{\mathscr{A}} \theta e' : \theta\tau'} \quad (3)}{\mathrm{M};\Sigma;\theta\Gamma \rhd_{\mathscr{A}} \theta(\lambda x{:}\tau''.e') : \theta(\tau'' \xrightarrow{\mathfrak{q}'}\!\circ\, \tau')} \qquad \dfrac{\cdots\ \mathfrak{q}' \sqsubseteq \mathfrak{q}\ \cdots}{\tau'' \xrightarrow{\mathfrak{q}'}\!\circ\, \tau' <: \tau'' \xrightarrow{\mathfrak{q}}\!\circ\, \tau'}}{\mathrm{M};\Sigma;\theta\Gamma \rhd_{\mathscr{A}} \theta(\lambda x{:}\tau''.e') : \theta(\tau'' \xrightarrow{\mathfrak{q}}\!\circ\, \tau')}.$$

Case $e_1\, e_2$.

By rule RAT-App, it must be the case that

$$\dfrac{\dfrac{\mathcal{A}}{\mathrm{M};\Sigma_1;\Gamma_1 \rhd_{\mathscr{A}} e_1 : \tau_2 \xrightarrow{\mathfrak{q}'}\!\circ\, \tau} \qquad \dfrac{\mathcal{B}}{\mathrm{M};\Sigma_2;\Gamma_2 \rhd_{\mathscr{A}} e_2 : \tau_2}}{\mathrm{M};\Sigma_1 \boxplus \Sigma_2;\Gamma_1 \boxplus \Gamma_2 \rhd_{\mathscr{A}} e_1\, e_2 : \tau},$$

using the notation of definition B.5.

Then

(1) $\mathrm{M};\Sigma_1;\theta\Gamma_1 \rhd_{\mathscr{A}} \theta e_1 : \theta(\tau_2 \xrightarrow{\mathfrak{q}'}\!\circ\, \tau)$ by $\mathcal{A}$, IH

(2) $\mathrm{M};\Sigma_2;\theta\Gamma_2 \rhd_{\mathscr{A}} \theta e_2 : \theta\tau_2$ by $\mathcal{B}$, IH

(3) $\theta\Gamma \rightsquigarrow (\theta\Gamma_1, \Gamma_2') \boxplus (\theta\Gamma_2, \Gamma_1')$ by lemma B.13

(4) $\mathrm{M};\Sigma_1;\theta\Gamma_1, \Gamma_2' \rhd_{\mathscr{A}} \theta e_1 : \theta(\tau_2 \xrightarrow{\mathfrak{q}'}\!\circ\, \tau)$ by (1), weak.

(5) $\mathrm{M};\Sigma_2;\theta\Gamma_2, \Gamma_1' \rhd_{\mathscr{A}} \theta e_2 : \theta\tau_2$ by (2), weak.

(6) $\mathrm{M};\Sigma;\theta\Gamma \rhd_{\mathscr{A}} \theta(e_1\, e_2) : \theta\tau$ by (3–5).

Case ℓ.

By rule RAT-Loc, it must be the case that

$$\dfrac{\ell : \tau' \in \Sigma}{\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} \ell : \mathsf{ref}\ \tau'},$$

where $\tau = \mathsf{ref}\ \tau'$.

Note that $\mathrm{FTV}(\Sigma) = \emptyset$, so $\mathrm{FTV}(\tau') = \emptyset$, so $\theta\tau' = \tau'$. Then,

$$\frac{\ell : \theta\tau' \in \Sigma}{\mathrm{M};\Sigma;\theta\Gamma \rhd_{\mathscr{A}} \theta\ell : \theta(\mathsf{ref}\ \tau')}.$$

Case $(\tau \underset{f\,\mathbf{g}}{\Leftarrow})\mathbf{e}'$**.**

By rule RAT-BOUNDARY, it must be the case that

$$\frac{\dfrac{\mathcal{A}}{\mathrm{M};\Sigma;\bullet \rhd_{\mathscr{C}} \mathbf{e}' : \tau^{\mathscr{C}}} \qquad \dfrac{\mathcal{B}}{\mathrm{FTV}(\tau) = \emptyset}}{\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} (\tau \underset{f\,\mathbf{g}}{\Leftarrow})\mathbf{e}' : \tau}.$$

Then,

$$\frac{\dfrac{\mathcal{A}, \text{IH (part 1)}}{\mathrm{M};\Sigma;\bullet \rhd_{\mathscr{C}} \theta\mathbf{e}' : \theta(\tau^{\mathscr{C}})} \qquad \dfrac{\mathcal{B}, \theta\tau = \tau}{\mathrm{FTV}(\theta\tau) = \emptyset}}{\mathrm{M};\Sigma;\theta\Gamma \rhd_{\mathscr{A}} \theta(\tau \underset{f\,\mathbf{g}}{\Leftarrow})\mathbf{e}' : \theta\tau}.$$

The remaining cases are the same as in the first part, with context splitting as in this (the second) part. □

LEMMA 7.6 (No hidden locations, restated from p. 183)**.**

The type of a value reveals whether locations might appear in that value:

1. *If* $\mathrm{M};\Sigma;\bullet \rhd_{\mathscr{C}} \mathbf{v} : \boldsymbol{\tau}$ *then* $\Sigma \rhd_{\mathscr{C}} \mathbf{v}$ worthy.
2. *If* $\mathrm{M};\Sigma;\bullet \rhd_{\mathscr{A}} v : \tau$ *then* $\langle \Sigma|_{\mathrm{FL}(v)} \rangle \sqsubseteq \langle \tau \rangle$*. That is, if* τ *is unlimited then* $\Sigma;\bullet \rhd_{\mathscr{A}} v$ worthy.

Proof. By mutual induction on $\mathbf{v}$ and v.

1. By cases on $\mathbf{v}$:

   Case $\Lambda\boldsymbol{\alpha}.\mathbf{v}'$**.**

   By inversion of rule RCT-TABS and the induction hypothesis at $\mathbf{v}'$, since $\mathrm{FL}(\mathbf{v}') = \mathrm{FL}(\Lambda\boldsymbol{\alpha}.\mathbf{v}')$.

   Case $\lambda\mathbf{x}{:}\boldsymbol{\tau}'.\mathbf{e}'$**.**

   By inversion of rule RCT-ABS.

Case $(\underset{\mathbf{g}f}{\Leftarrow} \tau')^{\ell}\, v'$**.**

There three possible rules for typing this term: rule RCT-BLESSED, rule RCT-DEFUNCT, and rule RCT-SEALED.

The first two require that $\Sigma = [\Sigma_1]^{\ell}, \ell : \boldsymbol{\tau}, [\Sigma_2]^{\ell}$ for particular Σ_1, Σ_2, and $\boldsymbol{\tau}$. By inspection of the definition of $[\Sigma]^{\ell}$, it is clear that there are no bare $\mathrm{F}^{\mathscr{A}}$ types in the range of Σ. Thus, $\langle \Sigma|_{\mathrm{FL}(v)} \rangle = \mathsf{U}$.

For rule RCT-SEALED, by inversion, it must be the case that

(1) $\langle \tau' \rangle = \mathsf{U}$ and

(2) $\mathrm{M}; \Sigma; \bullet \rhd_{\mathscr{A}} v' : \tau'$.

By the induction hypothesis (part 2),

(3) $\langle \Sigma|_{\mathrm{FL}(v')} \rangle \sqsubseteq \langle \tau' \rangle = \mathsf{U}$.

Since $\mathrm{FL}((\underset{\mathbf{g}f}{\Leftarrow} \tau')^{\ell}\, v') = \mathrm{FL}(v') \cup \{\ell\}$, and since $\Sigma(\ell) = \boldsymbol{\tau}'$, we see that $\langle \Sigma|_{\mathrm{FL}(\mathbf{v})} \rangle = \mathsf{U}$.

2. By cases on v:

Case $\Lambda \alpha^{\mathfrak{q}}.\, v'$**.**

By rule RAT-TABS, it must be the case that

(1) $\mathrm{M}; \Sigma; \bullet \rhd_{\mathscr{A}} \Lambda \alpha^{\mathfrak{q}}.\, v' : \forall\, \alpha^{\mathfrak{q}}.\, \tau'$ where

(2) $\forall\, \alpha^{\mathfrak{q}}.\, \tau' = \tau'$, and thus

(3) $\langle \tau \rangle = \langle \forall\, \alpha^{\mathfrak{q}}.\, \tau' \rangle = \langle \tau' \rangle$.

By inversion, it must be the case that

(4) $\mathrm{M}; \Sigma; \bullet \rhd_{\mathscr{A}} v' : \tau'$.

By the induction hypothesis,

(5) $\langle \Sigma|_{\mathrm{FL}(v')} \rangle \sqsubseteq \langle \tau' \rangle$,

and since $\mathrm{FL}(v') = \mathrm{FL}(v)$, we have that $\langle \Sigma|_{\mathrm{FL}(v)} \rangle \sqsubseteq \langle \tau \rangle$.

Case $\lambda x{:}\tau'.\, e'$**.**

Let $\mathfrak{q} = \langle \tau \rangle$. Then by inversion of rule RAT-ABS,

(1) $\mathfrak{q} = \mathfrak{q}_1 \sqcup \mathfrak{q}_2$ where

(2) $\langle \bullet|_{\mathrm{FV}(v)} \rangle = \mathfrak{q}_1$ and

(3) $\langle \Sigma|_{\mathrm{FL}(v)} \rangle = \mathfrak{q}_2$.

Since $\mathfrak{q}_1 = \mathsf{U} = \bot$, we know that $\mathfrak{q}_2 = \mathfrak{q} = \langle \tau \rangle$.

Case $\langle v_1, v_2 \rangle$**.**

This has a pair type of the form $\tau_1 \otimes \tau_2$, and therefore

$$\begin{aligned}
\langle \Sigma|_{\mathrm{FL}(v)} \rangle &= \langle \Sigma|_{\mathrm{FL}(v_1) \cup \mathrm{FL}(v_2)} \rangle && \text{def. of } \mathrm{FL}(\langle v_1, v_2 \rangle) \\
&= \left| \Sigma|_{\mathrm{FL}(v_1)} \cup \Sigma|_{\mathrm{FL}(v_2)} \right| && \text{set theory} \\
&= \langle \Sigma|_{\mathrm{FL}(v_1)} \rangle \sqcup \langle \Sigma|_{\mathrm{FL}(v_2)} \rangle && \text{monotonicity of } |\cdot| \\
&\sqsubseteq \langle \tau_1 \rangle \sqcup \langle \tau_2 \rangle && \text{IH twice; monotonicity of } \sqcup \\
&= \langle \tau \rangle && \text{def. of } \langle \tau_1 \otimes \tau_2 \rangle.
\end{aligned}$$

Case ℓ**.**

By inversion of rule RAT-Loc, this has type $\mathsf{ref}\ \tau'$ if and only if $\ell : \tau' \in \Sigma$. Then

$$\begin{aligned}
\langle \tau \rangle &= \langle \mathsf{ref}\ \tau' \rangle \\
&= \mathsf{A} \\
&= \langle \bullet, \ell : \tau' \rangle \\
&= \langle \Sigma|_{\{\ell\}} \rangle \\
&= \langle \Sigma|_{\mathrm{FL}(\ell)} \rangle.
\end{aligned}$$

Case $(\tau \underset{f\,\mathbf{g}}{\Leftarrow})^{\bullet}\, \mathbf{v}'$**.**

By inversion of rule RAT-Wrapped, we know that

(1) $\mathsf{M}; \Sigma; \bullet \rhd_{\mathscr{C}} \mathbf{v}' : \tau^{\mathscr{C}}$.

Then by part 1 of the induction hypothesis,

(2) $\langle \Sigma|_{\mathrm{FL}(\mathbf{v}')} \rangle = \mathsf{U}$.

Since $\mathrm{FL}(v) = \mathrm{FL}(\mathbf{v}')$, we have that $\langle \Sigma|_{\mathrm{FL}(\mathbf{v})} \rangle = \mathsf{U} \sqsubseteq \mathfrak{q}$ for all $\mathfrak{q}$. □

Lemma B.14 (Substitution and worthiness)**.**

1. *If* $\Sigma_1 \rhd_{\mathscr{C}} \mathbf{e}$ worthy *and* $\Sigma_2 \rhd_{\mathscr{C}} \mathbf{v}$ worthy*, where* $\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2$*, then* $\Sigma \rhd_{\mathscr{C}} \{\mathbf{v}/\mathbf{x}\}\mathbf{e}$ worthy.

2. *If* $\Sigma_1; \Gamma, x : \tau \rhd_{\mathscr{A}} e$ worthy *and* $\Sigma_2; \bullet \rhd_{\mathscr{A}} v$ worthy, *where* $\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2$, *then* $\Sigma; \Gamma \rhd_{\mathscr{A}} \{v/x\}e$ worthy.

Proof.

1. Suppose that $\Sigma_1 \rhd_{\mathscr{C}} \mathbf{e}$ worthy and $\Sigma_2 \rhd_{\mathscr{C}} \mathbf{v}$ worthy. Then by definition 7.5, $\langle \Sigma_1|_{\mathrm{FL}(\mathbf{e})} \rangle = \mathsf{U}$ and $\langle \Sigma_2|_{\mathrm{FL}(\mathbf{v})} \rangle = \mathsf{U}$. This means that all the locations of $\mathbf{e}$ have $\mathrm{F}_{\mathscr{C}}$ types or protected $\mathrm{F}^{\mathscr{A}}$ types in Σ_1, and likewise for $\mathbf{v}$ and Σ_2. Since $\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2$, by lemma B.4, $\Sigma_1 \sim_{\mathsf{U}} \Sigma_2$, so $\Sigma_1|_{\mathrm{FL}(\mathbf{e})} = \Sigma_2|_{\mathrm{FL}(\mathbf{v})}$. Note that $\mathrm{FL}(\{\mathbf{v}/\mathbf{x}\}\mathbf{e}) = \mathrm{FL}(\mathbf{e}) \cup \mathrm{FL}(\mathbf{v})$. Therefore, $\langle \Sigma|_{\mathrm{FL}(\{\mathbf{v}/\mathbf{x}\}\mathbf{e})} \rangle = \langle \Sigma|_{\mathrm{FL}(\mathbf{e}) \cup \mathrm{FL}(\mathbf{v})} \rangle = \langle \Sigma_1|_{\mathrm{FL}(\mathbf{e})} \rangle \sqcup \langle \Sigma_2|_{\mathrm{FL}(\mathbf{v})} \rangle = \mathsf{U}$. Thus, $\Sigma \rhd_{\mathscr{C}} \{\mathbf{v}/\mathbf{x}\}\mathbf{e}$ worthy.

2. Suppose that $\Sigma_1; \Gamma, x : \tau_x \rhd_{\mathscr{A}} e$ worthy and $\Sigma_2; \bullet \rhd_{\mathscr{A}} v$ worthy. Then by the same reasoning as in the previous part, $\langle \Sigma|_{\mathrm{FL}(\{v/x\}e)} \rangle = \mathsf{U}$. It remains to be shown that $\langle \Gamma|_{\mathrm{FV}(\{v/x\}e)} \rangle = \mathsf{U}$.

   By definition 7.5, $\langle \Gamma_1, x : \tau_x|_{\mathrm{FV}(e)} \rangle = \mathsf{U}$. Since v types in the empty typing context, we know that $\mathrm{FV}(v) = \emptyset$. Then by induction on e, $\mathrm{FV}(\{v/x\}e) = \mathrm{FV}(e) \setminus \{x\}$. Thus, $\Gamma_1|_{\mathrm{FV}(\{v/x\}e)}$ is merely a restriction of $\Gamma_1, x : \tau_x|_{\mathrm{FV}(e)}$, so its qualifier must be U as well. Thus, $\Sigma; \Gamma_1 \rhd_{\mathscr{A}} \{v/x\}e$ worthy. □

LEMMA 7.7 (Substitution, restated from p. 183)**.**

1. *If* $\mathrm{M}; \Sigma_1; \boldsymbol{\Gamma}, \mathbf{x} : \boldsymbol{\tau}_{\mathbf{x}} \rhd_{\mathscr{C}} \mathbf{e} : \boldsymbol{\tau}$ *and* $\mathrm{M}; \Sigma_2; \bullet \rhd_{\mathscr{C}} \mathbf{v} : \boldsymbol{\tau}_{\mathbf{x}}$ *where* $\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2$, *then* $\mathrm{M}; \Sigma; \boldsymbol{\Gamma} \rhd_{\mathscr{C}} \{\mathbf{v}/\mathbf{x}\}\mathbf{e} : \boldsymbol{\tau}$.

2. *If* $\mathrm{M}; \Sigma_1; \Gamma, x : \tau_x \rhd_{\mathscr{A}} e : \tau$ *and* $\mathrm{M}; \Sigma_2; \bullet \rhd_{\mathscr{A}} v : \tau_x$ *where* $\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2$, *then* $\mathrm{M}; \Sigma; \Gamma \rhd_{\mathscr{A}} \{v/x\}e : \tau$.

Proof. By induction on the structure of the type derivation for $\mathbf{e}$ or e. We consider each proof tree by the expression in its conclusion (where possible).

1. If $\mathbf{x} \notin \mathrm{FV}(\mathbf{e})$, then $\{\mathbf{v}/\mathbf{x}\}\mathbf{e} = \mathbf{e}$, so the conclusion holds by weakening.

   Otherwise, by cases in e, considering multiple type rules where necessary. Let value substitution $\theta = \{\mathbf{v}/\mathbf{x}\}$.

Case $\Lambda\boldsymbol{\alpha}.\mathbf{v}'$.

By rule RCT-TABS, it must be the case that

(1) $\mathrm{M};\Sigma_1;\boldsymbol{\Gamma},\mathbf{x}:\boldsymbol{\tau}_{\mathbf{x}} \rhd_{\mathscr{C}} \mathbf{v}':\boldsymbol{\tau}'$, where

(2) $\boldsymbol{\tau} = \forall\,\boldsymbol{\alpha}.\boldsymbol{\tau}'$.

By the induction hypothesis,

(3) $\mathrm{M};\Sigma;\boldsymbol{\Gamma} \rhd_{\mathscr{C}} \theta\mathbf{v}':\boldsymbol{\tau}'$.

By evasive relettering, $\Lambda\boldsymbol{\alpha}.\theta\,\mathbf{v}' = \theta\,(\Lambda\boldsymbol{\alpha}.\mathbf{v}')$.

Then by rule RCT-TABS,

(4) $\mathrm{M};\Sigma;\boldsymbol{\Gamma} \rhd_{\mathscr{C}} \theta(\Lambda\boldsymbol{\alpha}.\mathbf{v}'):\forall\,\boldsymbol{\alpha}.\boldsymbol{\tau}'$.

Case $\lambda\mathbf{y}{:}\boldsymbol{\tau}_{\mathbf{y}}.\mathbf{e}'$.

Without loss of generality, assume that $\mathbf{x} \neq \mathbf{y}$. By rule RCT-ABS, it must be the case that

(1) $\mathrm{M};\Sigma_1;\boldsymbol{\Gamma},\mathbf{x}:\boldsymbol{\tau}_{\mathbf{x}},\mathbf{y}:\boldsymbol{\tau}_{\mathbf{y}} \rhd_{\mathscr{C}} \mathbf{e}':\boldsymbol{\tau}'$ where

(2) $\boldsymbol{\tau} = \boldsymbol{\tau}_{\mathbf{y}} \to \boldsymbol{\tau}'$ and

(3) $\Sigma_1 \rhd_{\mathscr{C}} \lambda\mathbf{y}{:}\boldsymbol{\tau}_{\mathbf{y}}.\mathbf{e}'$ worthy.

By exchange and the induction hypothesis,

(4) $\mathrm{M};\Sigma;\boldsymbol{\Gamma},\mathbf{y}:\boldsymbol{\tau}_{\mathbf{y}} \rhd_{\mathscr{C}} \theta\mathbf{e}':\boldsymbol{\tau}'$.

By lemma B.14,

(5) $\Sigma \rhd_{\mathscr{C}} \theta(\lambda\mathbf{y}{:}\boldsymbol{\tau}_{\mathbf{y}}.\mathbf{e}')$ worthy.

Thus, by rule RCT-ABS, $\mathrm{M};\Sigma;\boldsymbol{\Gamma} \rhd_{\mathscr{C}} \theta(\lambda\mathbf{y}{:}\boldsymbol{\tau}_{\mathbf{y}}.\mathbf{e}'):\boldsymbol{\tau}_y \to \boldsymbol{\tau}'$.

Case y.

Either $\mathbf{x} = \mathbf{y}$ or $\mathbf{x} \neq \mathbf{y}$:

Case $\mathbf{x} = \mathbf{y}$.

Then $\boldsymbol{\tau} = \boldsymbol{\tau}_{\mathbf{x}}$, by inversion of rule RCT-VAR.

Since $\theta\,\mathbf{x} = \mathbf{v}$,

(1) $\mathrm{M};\Sigma_2;\bullet \rhd_{\mathscr{C}} \theta\mathbf{x}:\boldsymbol{\tau}$.

Then by weakening.

Case $\mathbf{x} \neq \mathbf{y}$.

Then $\theta\,\mathbf{y} = \mathbf{y}$, so by inversion of rule RCT-VAR, $\mathbf{y} : \boldsymbol{\tau} \in \boldsymbol{\Gamma}$. Then by rule RCT-VAR. Other cases where $\mathbf{x} \notin \mathrm{FV}(\mathbf{e})$ will be similar.

Case $\mathbf{e}'_1\ \boldsymbol{\tau}_2$.

By inversion of rule RCT-TAPP, we have that

(1) $\mathrm{M}; \Sigma_1; \boldsymbol{\Gamma}, \mathbf{x} : \boldsymbol{\tau}_{\mathbf{x}} \rhd_{\mathscr{C}} \mathbf{e}'_1 : \forall\, \boldsymbol{\alpha}.\, \boldsymbol{\tau}'$ where

(2) $\{\boldsymbol{\tau}_2/\boldsymbol{\alpha}\}\boldsymbol{\tau}' = \boldsymbol{\tau}$.

By the induction hypothesis,

(3) $\mathrm{M}; \Sigma; \boldsymbol{\Gamma} \rhd_{\mathscr{C}} \theta\mathbf{e}'_1 : \forall\, \boldsymbol{\alpha}.\, \boldsymbol{\tau}'$.

Then by rule RCT-TAPP,

(4) $\mathrm{M}; \Sigma; \boldsymbol{\Gamma} \rhd_{\mathscr{C}} \theta(\mathbf{e}'_1\ \boldsymbol{\tau}_2) : \{\boldsymbol{\tau}_2/\boldsymbol{\alpha}\}\boldsymbol{\tau}'$.

Case $e_1\, e_2$.

By inversion of rule RCT-APP,

(1) $\mathrm{M}; \Sigma_{11}; \boldsymbol{\Gamma}, \mathbf{x} : \boldsymbol{\tau}_{\mathbf{x}} \rhd_{\mathscr{C}} \mathbf{e}_1 : \boldsymbol{\tau}' \to \boldsymbol{\tau}$ and

(2) $\mathrm{M}; \Sigma_{12}; \boldsymbol{\Gamma}, \mathbf{x} : \boldsymbol{\tau}_{\mathbf{x}} \rhd_{\mathscr{C}} \mathbf{e}_2 : \boldsymbol{\tau}'$, where

(3) $\Sigma_1 \rightsquigarrow \Sigma_{11} \boxplus \Sigma_{12}$,

for some $\boldsymbol{\tau}'$.

By lemma 7.6, $\mathbf{v}$ is worthy, thus $\Sigma_2|_{\mathsf{U}}$ is sufficient for typing $\mathbf{v}$.

By the induction hypothesis (twice),

(4) $\mathrm{M}; \Sigma_{11} \boxplus \Sigma_2|_{\mathsf{U}}; \boldsymbol{\Gamma} \rhd_{\mathscr{C}} \theta\mathbf{e}_1 : \boldsymbol{\tau}' \to \boldsymbol{\tau}$ and

(5) $\mathrm{M}; \Sigma_{12} \boxplus \Sigma_2|_{\mathsf{U}}; \boldsymbol{\Gamma} \rhd_{\mathscr{C}} \theta\mathbf{e}_2 : \boldsymbol{\tau}'$.

By lemma B.4,

(6) $\Sigma_{11} \rightsquigarrow \Sigma_{11} \boxplus \Sigma_2|_{\mathsf{U}}$ and

(7) $\Sigma_{12} \rightsquigarrow \Sigma_{12} \boxplus \Sigma_2|_{\mathsf{U}}$,

and thus

(8) $\mathrm{M}; \Sigma_{11}; \boldsymbol{\Gamma} \rhd_{\mathscr{C}} \theta\mathbf{e}_1 : \boldsymbol{\tau}' \to \boldsymbol{\tau}$ and

(9) $\mathrm{M}; \Sigma_{12}; \boldsymbol{\Gamma} \rhd_{\mathscr{C}} \theta\mathbf{e}_2 : \boldsymbol{\tau}'$.

Then by rule RCT-APP and weakening,

(10) $\mathrm{M}; \Sigma; \mathbf{\Gamma} \rhd_{\mathscr{C}} \theta(\mathbf{e}_1\, \mathbf{e}_2) : \boldsymbol{\tau}$.

Case g.

By weakening, as before when $\mathbf{x} \notin \mathrm{FV}(\mathbf{e})$.

Case $f^{\mathbf{g}}$.

$\mathbf{x} \notin \mathrm{FV}(\mathbf{e})$.

Case $(\underset{\mathbf{g}f}{\Leftarrow} \tau')e'$.

By inversion of rule RCT-BOUNDARY, it must be the case that

(1) $\mathrm{M}; \Sigma_1; \bullet \rhd_{\mathscr{A}} e' : \tau'$.

Thus $\mathrm{FV}(e') = \emptyset$, so by weakening as before when $\mathbf{x} \notin \mathrm{FV}(\mathbf{e})$.

Case $(\underset{\mathbf{g}f}{\Leftarrow} \tau')^{\ell}\, v'$.

There are three rules that may be at the root of our type derivation. In two, rules RCT-BLESSED and RCT-SEALED, this is as in the previous case where $\mathbf{x} \notin \mathrm{FV}(\mathbf{e})$, noting that since τ' remains wrappable. We must consider the case of rule RCT-DEFUNCT.

Then we know that,

(1) $(\tau')^{\mathscr{C}} = \boldsymbol{\tau}$,

(2) $\Sigma_1 = [\Sigma_{11}]^{\ell}, \ell : \mathbb{D}, [\Sigma_{12}]^{\ell}$,

(3) $\mathrm{FTV}(\tau') = \emptyset$,

(4) $\tau' \in W$ and

(5) $\langle \tau' \rangle = \mathsf{A}$.

This is sufficient to prove that

(6) $\mathrm{M}; \Sigma_1; \mathbf{\Gamma} \rhd_{\mathscr{C}} (\underset{\mathbf{g}f}{\Leftarrow} \tau')^{\ell}\, \theta\, v' : \boldsymbol{\tau}$

for any $\mathbf{\Gamma}$ and v'. Then by weakening.

2. If $x \notin \mathrm{FV}(e)$, then $\{v/x\}e = e$, so the conclusion holds by weakening. Otherwise we consider cases on the root of the type derivation.

   The structural cases for e are insufficient due to rules with overlapping conclusions, so in the case of subsumption, we identify the rule at the

root of the derivation; when unambiguous among the remaining cases, we identify the subject term at the root. (Let substitution $\theta = \{v/x\}$.)

Case RAT-SUBSUME**.**

Then by inversion, we know that there exists some $\tau_<$ such that

(1) $\mathrm{M}; \Sigma_1; \Gamma, x : \tau_x \rhd_{\mathscr{A}} e : \tau_<$ and

(2) $\tau_< <: \tau$.

By the induction hypothesis,

(3) $\mathrm{M}; \Sigma; \Gamma \rhd_{\mathscr{A}} \theta e : \tau_<$,

and then by rule RAT-SUBSUME.

Case $\Lambda\alpha^{\mathfrak{q}}.v'$**.**

As for $\mathrm{F}_{\mathscr{C}}$.

Case $\lambda y{:}\tau_y.e'$**.**

Without loss of generality, assume that $x \neq y$.

By inversion of rule RAT-ABS, we know that

(1) $\mathrm{M}; \Sigma_1; \Gamma, x : \tau_x, y : \tau_y \rhd_{\mathscr{A}} e' : \tau'$ and

(2) $\langle \Gamma, x : \tau_x |_{\mathrm{FV}(\lambda y:\tau_y.e')} \rangle \sqcup \langle \Sigma_1 |_{\mathrm{FL}(\lambda y:\tau_y.e')} \rangle = \mathfrak{q}$ where

(3) $\tau = \tau_y \xrightarrow{\mathfrak{q}} \tau'$,

for some τ' and $\mathfrak{q}$.

By exchange and the induction hypothesis,

(4) $\mathrm{M}; \Sigma_1; \Gamma, y : \tau_y \rhd_{\mathscr{A}} \theta e' : \tau'$.

By rules RAT-ABS and RAT-SUBSUME, it will suffice to show that $\langle \Gamma |_{\mathrm{FV}(\lambda y:\tau_y.\theta e')} \rangle \sqcup \langle \Sigma |_{\mathrm{FL}(\lambda y:\tau_y.\theta e')} \rangle \sqsubseteq \mathfrak{q}$. If $\mathfrak{q} = \mathsf{A}$, then this holds trivially, so we only need to show the case where $\mathfrak{q} = \mathsf{U}$. Assuming that $\mathfrak{q} = \mathsf{U}$, it suffices to show that $\Sigma; \Gamma \rhd_{\mathscr{A}} \theta(\lambda y{:}\tau_y.e')$ worthy.

If $\mathfrak{q} = \mathsf{U}$ then by (2),

(5) $\langle \Gamma, x : \tau_x |_{\mathrm{FV}(\lambda y:\tau_y.e')} \rangle = \mathsf{U}$ and

(6) $\langle \Sigma_1 |_{\mathrm{FL}(\lambda y:\tau_y.e')} \rangle = \mathsf{U}$.

That is,

(7) $\Sigma_1;\Gamma,x:\tau_x \rhd_{\mathscr{A}} \lambda y{:}\tau_y.\, e'$ worthy.

Because we already considered the case where $x \notin \mathrm{FV}(e)$ (trivial, by weakening), we consider here only the case where $x \in \mathrm{FV}(e)$. This means that $x \in \mathrm{dom}(\Gamma, x:\tau_x|_{\mathrm{FV}(\lambda y:\tau_y.\, e')})$. Then from (5) and by the definition of qualifiers of typing contexts, this means that $\langle\tau_x\rangle = \mathsf{U}$. Since $\mathrm{M};\Sigma_2;\bullet \rhd_{\mathscr{A}} v:\tau_x$, by lemma 7.6,

(8) $\Sigma_2;\bullet \rhd_{\mathscr{A}} v$ worthy.

Then by lemma B.14, (7), and (8),

(9) $\Sigma;\Gamma \rhd_{\mathscr{A}} \theta(\lambda y{:}\tau_y.\, e')$ worthy,

as desired.

Case y**.**

If $x \neq y$, then as before when $x \notin \mathrm{FV}(e)$.

Otherwise, $\tau = \tau_x$ by rule RAT-VAR.

Since $\theta\, y = v$,

(1) $\mathrm{M};\Sigma_2;\bullet \rhd_{\mathscr{A}} \theta y:\tau$.

Then by weakening.

Case $e_1'\,\tau_2$**.**

As for $\mathrm{F}_{\mathscr{C}}$.

Case $e_1\,e_2$**.**

By inversion of rule RAT-APP,

(1) $\mathrm{M};\Sigma_{11};\Gamma_1 \rhd_{\mathscr{A}} e_1 : \tau' \xrightarrow{\mathsf{q}} \tau$,

(2) $\mathrm{M};\Sigma_{12};\Gamma_2 \rhd_{\mathscr{A}} e_2 : \tau'$,

(3) $\Sigma_1 \rightsquigarrow \Sigma_{11} \boxplus \Sigma_{12}$, and

(4) $\Gamma, x:\tau_x \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$

for some τ'.

By the definition of context splitting, there are three ways to reach (4):

Case $\dfrac{\Gamma \rightsquigarrow \Gamma_1' \boxplus \Gamma_2 \qquad \langle \tau_x \rangle = \mathsf{A}}{\Gamma, x : \tau_x \rightsquigarrow \Gamma_1', x : \tau_x \boxplus \Gamma_2}$.

In particular, $x \notin \operatorname{dom} \Gamma_2$, so it must not be free in e_2; thus $\theta e_2 = e_2$.

We apply the induction hypothesis only to e_1, yielding

(1) $\mathrm{M}; \Sigma_{11} \boxplus \Sigma_2; \Gamma_1' \rhd_{\mathscr{A}} \theta e_1 : \tau' \xrightarrow{\mathsf{q}} \tau$.

By rule RAT-APP,

(2) $\mathrm{M}; \Sigma; \Gamma \rhd_{\mathscr{A}} \theta(e_1\, e_2) : \tau$.

Case $\dfrac{\Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2' \qquad \langle \tau_x \rangle = \mathsf{A}}{\Gamma, x : \tau_x \rightsquigarrow \Gamma_1 \boxplus \Gamma_2', x : \tau_x}$.

By symmetry.

Case $\dfrac{\Gamma \rightsquigarrow \Gamma_1' \boxplus \Gamma_2' \qquad \langle \tau_x \rangle = \mathsf{U}}{\Gamma, x : \tau_x \rightsquigarrow \Gamma_1', x : \tau_x \boxplus \Gamma_2', x : \tau_x}$.

As in the RCT-APP for $\mathrm{F}_{\mathscr{C}}$.

Case $f, \mathbf{g}^f, \ell$.

$x \notin \mathrm{FV}(e)$.

Case $\langle v_1, v_2 \rangle$.

As in the RAT-APP case above.

Case $\mathsf{let}\, \langle y_1, y_2 \rangle = e_1\, \mathsf{in}\, e_2$.

As in the RAT-APP case above.

Case $(\tau \underset{f\,\mathbf{g}}{\Leftarrow}) \mathbf{e}'$.

By inversion of rule RAT-BOUNDARY, it must be the case that

(1) $\mathrm{M}; \Sigma_1; \bullet \rhd_{\mathscr{C}} \mathbf{e}' : \tau^{\mathscr{C}}$.

Thus $\mathrm{FV}(e) = \emptyset$.

Case $(\tau \underset{f\,\mathbf{g}}{\Leftarrow})^{\bullet} \mathbf{v}'$.

As in the previous case, with a second premise that $\tau^{\mathscr{C}} \in \mathbf{W}$. □

B.3 Preservation

Observe that changing the type of a location from $\mathbb{B}$ to $\mathbb{D}$ in a store context Σ does not break the typing of an expression using Σ. Furthermore, changing the value in a location in the store from $\mathbb{B}$ to $\mathbb{D}$ does not change the typing of the store, *except* that it updates the type associated with that location in the store context. To be precise:

LEMMA B.15 (Going defunct)**.**

1. *If* $\mathrm{M}; \Sigma_1, \ell : \mathbb{B}; \bullet \rhd_{\mathscr{C}} \mathbf{e} : \boldsymbol{\tau}$ *then* $\mathrm{M}; \Sigma_1, \ell : \mathbb{D}; \bullet \rhd_{\mathscr{C}} \mathbf{e} : \boldsymbol{\tau}$
2. *If* $\mathrm{M}; \Sigma_1, \ell : \mathbb{B}; \bullet \rhd_{\mathscr{A}} e : \tau$ *then* $\mathrm{M}; \Sigma_1, \ell : \mathbb{D}; \bullet \rhd_{\mathscr{A}} e : \tau$
3. *If* $\mathrm{M}; \Sigma_1, [\Sigma_2]^{\ell}, \ell : \mathbb{D}; \bullet \rhd_{\mathscr{C}} \mathbf{e} : \boldsymbol{\tau}$ *then* $\mathrm{M}; \Sigma_1, \Sigma_2|_{\mathsf{U}}, \ell : \mathbb{D}; \bullet \rhd_{\mathscr{C}} \mathbf{e} : \boldsymbol{\tau}$
4. *If* $\mathrm{M}; \Sigma_1, [\Sigma_2]^{\ell}, \ell : \mathbb{D}; \bullet \rhd_{\mathscr{A}} e : \tau$ *then* $\mathrm{M}; \Sigma_1, \Sigma_2|_{\mathsf{U}}, \ell : \mathbb{D}; \bullet \rhd_{\mathscr{A}} e : \tau$
5. *If* $\mathrm{M}; \Sigma_1, [\Sigma']^{\ell}, \ell : \mathbb{B} \rhd s \uplus \{\ell \mapsto \textsc{blssd}\} : \Sigma_2, [\Sigma']^{\ell}, \ell : \mathbb{B}$
   then $\mathrm{M}; \Sigma_1, \Sigma'|_{\mathsf{U}}, \ell : \mathbb{D} \rhd s \uplus \{\ell \mapsto \textsc{dfnct}\} : \Sigma_2, \Sigma', \ell : \mathbb{D}$.

Proof.

1. Observe that there are only two rules that mention store context bindings of the form $\ell : \boldsymbol{\tau}'$ (note that $\boldsymbol{\tau}'$ is an $\mathrm{F}_{\mathscr{C}}$ types):

   RCT-BLESSED Then the subterm types in the new store context by rule RCT-DEFUNCT.

   RCT-DEFUNCT Vacuous, as it requires that $\ell : \mathbb{D}$, which contradicts the assumption.

   Thus, we can construct a new derivation.

2. Likewise.

3. By induction on the length of Σ_2. The only rule that makes use of a protected binding like $\ell' : [\tau]^{\ell}$ is rule RCT-BLESSED. But since $\ell : \mathbb{D}$, that rule never applies. Thus, such a binding for ℓ' is irrelevant to the typing. The remaining bindings are present in $\Sigma_2|_{\mathsf{U}}$.

4. Likewise.

5. By inversion of rule RS-LocC,

   (1) $\Sigma_1, [\Sigma']^\ell, \ell : \mathbb{B} \rightsquigarrow \Sigma_1' \boxplus \Sigma_2'$,

   (2) $\mathrm{M}; \Sigma_1' \rhd s : \Sigma_2, [\Sigma']^\ell$, and

   (3) $\mathrm{M}; \Sigma_2'; \bullet \rhd_{\mathscr{C}} \textsc{blssd} : \mathbb{B}$

   for some Σ_1' and Σ_2'.

   Note that (3) requires no context to type, so any affine assumptions in $\Sigma_1, [\Sigma']^\ell, \ell : \mathbb{B}$ may be distributed to Σ_1'. (If not all of them are, we may assume that they are by weakening.) Hence,

   (4) $\mathrm{M}; \Sigma_1, [\Sigma']^\ell, \ell : \mathbb{B} \rhd s : \Sigma_2, [\Sigma']^\ell$.

   Now by induction on the length of s: Consider that the derivation of (4) types each value v_i in the range of s using $\Sigma_i', [\Sigma']^\ell, \ell : \mathbb{B}$ where Σ_i' is some portion split from Σ_1. By parts 1 and 2 of this lemma, each v_i may be given the same type using $\Sigma_i', [\Sigma']^\ell, \ell : \mathbb{D}$. Then by parts 3 and 4, each value v_i may be given the same type using $\Sigma_i', \Sigma'|_{\mathsf{U}}, \ell : \mathbb{D}$. Thus,

   (5) $\mathrm{M}; \Sigma_1, \Sigma'|_{\mathsf{U}}, \ell : \mathbb{D} \rhd s : \Sigma_2, [\Sigma']^\ell$.

   Then by lemma B.7,

   (6) $\mathrm{M}; \Sigma_1, \Sigma'|_{\mathsf{U}}, \ell : \mathbb{D} \rhd s : \Sigma_2, \Sigma'$.

   Note that

   (7) $\Sigma_1, \Sigma'|_{\mathsf{U}}, \ell : \mathbb{D} \rightsquigarrow \Sigma_1, \Sigma'|_{\mathsf{U}}, \ell : \mathbb{D} \boxplus \Sigma_1|_{\mathsf{U}}, \Sigma'|_{\mathsf{U}}, \ell : \mathbb{D}$ and

   (8) $\mathrm{M}; \Sigma_1|_{\mathsf{U}}, \Sigma'|_{\mathsf{U}}, \ell : \mathbb{D}; \bullet \rhd_{\mathscr{C}} \textsc{dfnct} : \mathbb{D}$.

   Then by rule RS-LocC and (6–8),

   (9) $\mathrm{M}; \Sigma_1, \Sigma'|_{\mathsf{U}}, \ell : \mathbb{D} \rhd s \uplus \{\ell \mapsto \textsc{dfnct}\} : \Sigma_2, \Sigma', \ell : \mathbb{D}$. □

LEMMA B.16 (Canonical Forms)**.**

1. *For $F_{\mathscr{C}}$:*

   a) *If* $\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{C}} \mathbf{v} : \forall\,\boldsymbol{\alpha}.\boldsymbol{\tau}$ *then* $\mathbf{v}$ *is either:*

   - $\Lambda\boldsymbol{\alpha}.\mathbf{v}'$ *for some* $\mathbf{v}'$*, or*
   - $(\underset{\mathbf{g}f}{\Leftarrow} \forall\,\beta^{\mathfrak{q}}.\tau')^{\ell}\, v'$ *where* $\forall\,\boldsymbol{\alpha}.\boldsymbol{\tau} = (\forall\,\beta^{\mathfrak{q}}.\tau')^{\mathscr{C}}$ *for some* ℓ*,* $\mathbf{g}$*,* f*,* $\beta^{\mathfrak{q}}$*,* τ'*, and* v'*.*

   b) *If* $\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{C}} \mathbf{v} : \boldsymbol{\tau}_1 \to \boldsymbol{\tau}_2$ *then* $\mathbf{v}$ *is either:*

   - $\lambda\mathbf{x}{:}\boldsymbol{\tau}_1.\mathbf{e}$ *for some* $\mathbf{e}$*, or*
   - $(\underset{\mathbf{g}f}{\Leftarrow} \tau_1 \xrightarrow{\mathfrak{q}}\!\circ\, \tau_2)^{\ell}\, v'$ *where* $\boldsymbol{\tau}_1 \to \boldsymbol{\tau}_2 = (\tau_1 \xrightarrow{\mathfrak{q}}\!\circ\, \tau_2)^{\mathscr{C}}$ *for some* ℓ*,* $\mathbf{g}$*,* f*,* τ_1*,* $\mathfrak{q}$*,* τ_2*, and* v'*.*

   c) *If* $\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{C}} \mathbf{v} : \{\rho\}$ *then* $\mathbf{v} = (\underset{\mathbf{g}f}{\Leftarrow} \rho)^{\ell}\, v'$ *for some* ℓ*,* $\mathbf{g}$*,* f*,* ρ*, and* v'*.*

2. *For $F^{\mathscr{A}}$:*

   a) *If* $\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} v : \forall\,\alpha^{\mathfrak{q}}.\tau$ *then* v *is either:*

   - $\Lambda\alpha^{\mathfrak{q}}.v'$ *for some* v'*, or*
   - $(\forall\,\alpha^{\mathfrak{q}}.\tau \underset{f\mathbf{g}}{\Leftarrow})^{\bullet}\, \mathbf{v}'$ *for some* f*,* $\mathbf{g}$*, and* $\mathbf{v}'$*.*

   b) *If* $\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} v : \tau_1 \xrightarrow{\mathfrak{q}}\!\circ\, \tau_2$ *then* $\mathbf{v}$ *is either:*

   - $\lambda x{:}\tau_1.e$ *for some* e*, or*
   - $(\tau_1 \xrightarrow{\mathfrak{q}}\!\circ\, \tau_2 \underset{f\mathbf{g}}{\Leftarrow})^{\bullet}\, \mathbf{v}'$ *for some* $\mathbf{g}$*,* f*, and* $\mathbf{v}'$*.*

   c) *If* $\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} v : \tau_1 \otimes \tau_2$ *then* $v = \langle v_1, v_2\rangle$ *for some* v_1 *and* v_2*.*

   d) *If* $\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} v : \mathsf{ref}\ \tau$ *then* $v = \ell$ *for some* ℓ*.*

   e) *If* $\mathrm{M};\Sigma;\Gamma \rhd_{\mathscr{A}} v : \{\boldsymbol{\alpha}\}$ *then* $v = (\{\boldsymbol{\alpha}\} \underset{f\mathbf{g}}{\Leftarrow})^{\bullet}\, \mathbf{v}'$ *for some* ℓ*,* $\mathbf{g}$*,* f*,* $\boldsymbol{\alpha}$*, and* $\mathbf{v}'$*.*

Proof. We exhaustively consider the values and their possible types, showing that no possibilities contradict the lemma:

1. By cases on $\mathbf{v}$:

   Case $\Lambda\boldsymbol{\alpha}.\mathbf{v}'$.

   This types only by rule RCT-TABS, which gives it a type of the form $\forall\,\boldsymbol{\alpha}.\boldsymbol{\tau}$. Therefore, $\Lambda\boldsymbol{\alpha}.\mathbf{v}'$ is a possibility for part (a).

Case $\lambda\mathbf{x}{:}\boldsymbol{\tau}.\mathbf{e}$**.**

This types only by rule RCT-ABS, which gives it a type of the form $\boldsymbol{\tau} \to \boldsymbol{\tau}'$. Therefore, $\lambda\mathbf{x}{:}\boldsymbol{\tau}.\mathbf{e}$ is a possibility for part (b).

Case $(\underset{\mathbf{g}f}{\Leftarrow}\tau)^{\ell}\, v'$**.**

This types only by rules RCT-SEALED, RCT-BLESSED, and RCT-DEFUNCT, all of which require that $\tau \in W$. Each of these gives $\mathbf{v}$ type $\tau^{\mathscr{C}}$. By cases on τ:

Case ρ**.**

Then $\tau^{\mathscr{C}} = \{\rho\}$, so this is a possibility for part (c).

Case $\tau_1 \xrightarrow{\mathfrak{q}} \tau_2$**.**

Then $\tau^{\mathscr{C}} = (\tau_1 \xrightarrow{\mathfrak{q}} \tau_2)^{\mathscr{C}}$, so this is a possibility for part (b).

Case $\forall\, \alpha^{\mathfrak{q}}.\tau'$**.**

Then $\tau^{\mathscr{C}} = (\forall\, \alpha^{\mathfrak{q}}.\tau')^{\mathscr{C}}$, so this is a possibility for part (a).

Case $\{\boldsymbol{\alpha}\}$**.**

Vacuous: contradicts the premise that $\tau \in W$.

2. In $\mathrm{F}^{\mathscr{A}}$, besides the rules mentioned for each syntactic form, each may type by rule RAT-SUBSUME with the syntax-specific rule proving the premise to the subsumption. We merely note that subtyping relates only types that are the same but for potentially different qualifiers $\mathfrak{q}$ on each function type, which we do not distinguish in this lemma.

   By cases on v:

   Case $\Lambda\alpha^{\mathfrak{q}}.v'$**.**

   This types only by rule RAT-TABS, which gives it a type of the form $\forall\, \alpha^{\mathfrak{q}}.\tau$; thus part (a).

   Case $\lambda x{:}\tau.e$**.**

   This types only by rule RAT-ABS, which gives it a type of the form $\tau_1 \xrightarrow{\mathfrak{q}} \tau_2$; thus part (b).

   Case $\langle v_1, v_2\rangle$**.**

   This types only by rule RAT-PAIR, which gives it a type of the form $\tau_1 \otimes \tau_2$; thus part (c).

Case ℓ.

This types only by rule RAT-LOC, which gives it a type of the form ref τ; thus part (d).

Case $(\tau \underset{f\,\mathbf{g}}{\Leftarrow})^{\bullet}\,\mathbf{v}'$.

This types only by rule RAT-WRAPPED, which requires that $\tau^{\mathscr{C}} \in \mathbf{W}$. By cases on τ:

Case ρ.

Then $\tau^{\mathscr{C}} = \{\rho\}$, which contradicts that $\tau^{\mathscr{C}} \in \mathbf{W}$; thus, vacuous.

Case $\tau_1 \xrightarrow{\mathfrak{q}} \tau_2$.

This is a possibility for part (b).

Case $\forall\, \alpha^{\mathfrak{q}}.\, \tau'$.

This is a possibility for part (a).

Case $\{\boldsymbol{\alpha}\}$.

This is a possibility for part (e). □

LEMMA 7.8 (Preservation, restated from p. 184)**.**

If $\mathrm{M} \rhd C_1 : \boldsymbol{\tau}$ *and* $C_1 \longmapsto_{\mathrm{M}} C_2$ *then* $\mathrm{M} \rhd C_2 : \boldsymbol{\tau}$.

Proof. Let $(s_1, \mathbf{e}_{01}) = C_1$ and $(s_2, \mathbf{e}_{02}) = C_2$. Since $\mathrm{M} \rhd (s_1, \mathbf{e}_{01}) : \boldsymbol{\tau}$, then by inversion of rule RCONF, we know that

(i) $(\forall \mathrm{m} \in \mathrm{M})\, \mathrm{M} \vdash \mathrm{m}$ okay,

(ii) $\mathrm{M}; \Sigma_1 \rhd s_1 : \Sigma$,

(iii) $\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2$ and

(iv) $\mathrm{M}; \Sigma_2; \bullet \rhd_{\mathscr{C}} \mathbf{e}_{01} : \boldsymbol{\tau}$.

Now by cases on the derivation of $(s_1, \mathbf{e}_{01}) \longmapsto_{\mathrm{M}} (s_2, \mathbf{e}_{02})$:

Case $\dfrac{(s_1, \mathbf{e}_1) \longmapsto_{\mathrm{M}} (s_2, \mathbf{e}_2)}{(s_1, \mathbf{E}[\mathbf{e}_1]) \longmapsto_{\mathrm{M}} (s_2, \mathbf{E}[\mathbf{e}_2])}$**.**

Then by lemma B.11, there exist some $\Sigma_{21} \boxplus \Sigma_{22} \rightsquigarrow \Sigma_2$ and $\boldsymbol{\tau}'$ such that

(v) $\mathrm{M}; \Sigma_{21}; \bullet \rhd_{\mathscr{C}} \mathbf{e}_1 : \boldsymbol{\tau}'$ and

(vi) $\mathrm{M}; \Sigma_{22}; \bullet \rhd_{\mathscr{C}} \mathbf{E} : [\boldsymbol{\tau}']\boldsymbol{\tau}$.

We need to find some $\Sigma' \rightsquigarrow \Sigma'_1 \boxplus \Sigma'_{21} \boxplus \Sigma'_{22}$ such that

(ii′) $\mathrm{M}; \Sigma'_1 \rhd s_2 : \Sigma'$,

(v′) $\mathrm{M}; \Sigma'_{21}; \bullet \rhd_{\mathscr{C}} \mathbf{e}_2 : \boldsymbol{\tau}'$, and

(vi′) $\mathrm{M}; \Sigma'_{22}; \bullet \rhd_{\mathscr{C}} \mathbf{E} : [\boldsymbol{\tau}']\boldsymbol{\tau}$.

Then by instantiating (vi′), we have $\mathrm{M}; \Sigma'_{21} \boxplus \Sigma'_{22}; \bullet \rhd_{\mathscr{C}} \mathbf{E}[\mathbf{e}_2] : \boldsymbol{\tau}$, and the rest follows by rule RCONF. (If $s_1 = s_2$, we let $\Sigma' = \Sigma$ and $\Sigma'_1 = \Sigma_1$ and $\Sigma'_{21} = \Sigma_{21}$ and $\Sigma'_{22} = \Sigma_{22}$; then it suffices to show that $\mathrm{M}; \Sigma_{21}; \bullet \rhd_{\mathscr{C}} \mathbf{e}'_2 : \boldsymbol{\tau}'$.)

By cases on the derivation of $(s_1, \mathbf{e}_1) \xrightarrow[\mathrm{M}]{} (s_2, \mathbf{e}_2)$:

Case $(\Lambda\boldsymbol{\alpha}.\mathbf{v})\,\boldsymbol{\tau}_2 \xrightarrow[\mathrm{M}]{} \{\boldsymbol{\tau}_2/\boldsymbol{\alpha}\}\mathbf{v}$**.**

By inversion of rule RCT-TAPP, we know that

(1) $\mathrm{M}; \Sigma_{21}; \bullet \rhd_{\mathscr{C}} \Lambda\boldsymbol{\alpha}.\mathbf{v} : \forall\,\boldsymbol{\alpha}.\boldsymbol{\tau}_1$ where

(2) $\boldsymbol{\tau}' = \{\boldsymbol{\tau}_2/\boldsymbol{\alpha}\}\boldsymbol{\tau}_1$.

Then, by inversion of rule RCT-TABS, we know that

(3) $\mathrm{M}; \Sigma_{21}; \bullet \rhd_{\mathscr{C}} \mathbf{v} : \boldsymbol{\tau}_1$.

By lemma 7.4, noting that $\{\boldsymbol{\tau}_2/\boldsymbol{\alpha}\}$ respects qualifiers, we then conclude that

(4) $\mathrm{M}; \Sigma_{21}; \bullet \rhd_{\mathscr{C}} \{\boldsymbol{\tau}_2/\boldsymbol{\alpha}\}\mathbf{v} : \{\boldsymbol{\tau}_2/\boldsymbol{\alpha}\}\boldsymbol{\tau}_2$.

Case $(\lambda\mathbf{x}{:}\boldsymbol{\tau}_\mathbf{x}.\mathbf{e})\mathbf{v} \xrightarrow[\mathrm{M}]{} \{\mathbf{v}/\mathbf{x}\}\mathbf{e}$**.**

By inversion of rule RCT-APP, we know that there exist some $\Sigma_{211} \boxplus \Sigma_{212} \leftrightsquigarrow \Sigma_{21}$ such that

(1) $\mathrm{M}; \Sigma_{211}; \bullet \rhd_{\mathscr{C}} \lambda\mathbf{x}{:}\boldsymbol{\tau}_\mathbf{x}.\mathbf{e} : \boldsymbol{\tau}_\mathbf{x} \to \boldsymbol{\tau}'$ and

(2) $\mathrm{M}; \Sigma_{212}; \bullet \rhd_{\mathscr{C}} \mathbf{v} : \boldsymbol{\tau}_\mathbf{x}$.

Then, by inversion of rule RCT-ABS on the former, we know that

(3) $\mathrm{M}; \Sigma_{211}; \bullet, \mathbf{x} : \boldsymbol{\tau}_\mathbf{x} \rhd_{\mathscr{C}} \mathbf{e} : \boldsymbol{\tau}'$.

By lemma 7.7, we have that $\mathrm{M}; \Sigma_{21}; \bullet \rhd_{\mathscr{C}} \{\mathbf{v}/\mathbf{x}\}\mathbf{e} : \boldsymbol{\tau}'$.

Case $\dfrac{(\mathbf{g} : \boldsymbol{\tau}'' = \mathbf{v}) \in \mathrm{M}}{\mathbf{g} \underset{\mathrm{M}}{\longrightarrow} \mathbf{v}}$.

By inversion of rule RCT-MOD, $\boldsymbol{\tau}'' = \boldsymbol{\tau}'$.

Then

(1) $\mathrm{M} \vdash \mathbf{g} : \boldsymbol{\tau}' = \mathbf{v}$ okay — by (i),

(2) $\mathrm{M}; \bullet \vdash_{\mathscr{C}} \mathbf{v} : \boldsymbol{\tau}'$ — by inv. of MODULEC,

(3) $\mathrm{M}; \bullet; \bullet \rhd_{\mathscr{C}} \mathbf{v} : \boldsymbol{\tau}'$ — by lemma 7.1, and

(4) $\mathrm{M}; \Sigma_{21}; \bullet \rhd_{\mathscr{C}} \mathbf{v} : \boldsymbol{\tau}'$ — by weakening.

Case $\dfrac{(f : \tau'' = v) \in \mathrm{M}}{f^{\mathbf{g}} \underset{\mathrm{M}}{\longrightarrow} (\underset{\mathbf{g}f}{\Leftarrow} \tau'')f}$.

By inversion of rule RCT-MODA, $(\tau'')^{\mathscr{C}} = \boldsymbol{\tau}'$.

Then

(1) $\mathrm{M}; \Sigma_{21}; \bullet \rhd_{\mathscr{A}} f : \tau''$ — by RAT-MOD,

(2) $\mathrm{M}; \Sigma_{21}; \bullet \rhd_{\mathscr{C}} (\underset{\mathbf{g}f}{\Leftarrow} \tau'')f : \boldsymbol{\tau}'$ — by RCT-BOUNDARY.

Case $\dfrac{v \neq (\{\boldsymbol{\alpha}\} \underset{f'\mathbf{g}'}{\Leftarrow})^{\bullet} \mathbf{v}'}{(s_1, (\underset{\mathbf{g}f}{\Leftarrow} \tau'')v) \underset{\mathrm{M}}{\longrightarrow} (s_1 \uplus \{\ell \mapsto \mathrm{BLSSD}\}, (\underset{\mathbf{g}f}{\Leftarrow} \tau'')^{\ell} v)}$.

By inversion of rule RCT-BOUNDARY,

(1) $\mathrm{M}; \Sigma_{21}; \bullet \rhd_{\mathscr{A}} v : \tau''$ where

(2) $(\tau'')^{\mathscr{C}} = \boldsymbol{\tau}'$.

Because $v \neq (\{\boldsymbol{\alpha}\} \underset{f'\mathbf{g}'}{\Leftarrow})^{\bullet} \mathbf{v}'$, by the contrapositive of lemma B.16, we know that τ'' is not of the form $\{\boldsymbol{\alpha}\}$; thus

(3) $\tau'' \in W$.

Then by cases on $\langle \tau'' \rangle$:

Case $\cup$**.**

Then by rule RCT-SEALED and weakening, $\mathrm{M}; \Sigma_{21}, \ell : \mathbb{B}; \bullet \rhd_{\mathscr{C}} (\underset{\mathbf{g}f}{\Leftarrow} \tau'')^{\ell} v : \boldsymbol{\tau}'$. Let $\Sigma'_{21} = \Sigma_{21}, \ell : \mathbb{B}$, $\Sigma'_{22} = \Sigma_{22}, \ell : \mathbb{B}$, and so on. Clearly $\mathrm{M}; \Sigma_1, \ell : \mathbb{B} \rhd s_1 \uplus \{\ell \mapsto \mathrm{BLSSD}\} : \Sigma_1, \ell : \mathbb{B}$ by rule RS-LOCC and weakening.

Case A**.**

Then by rule RCT-BLESSED,

(4) $\mathrm{M};[\Sigma_{21}]^\ell, \ell : \mathbb{B}; \bullet \rhd_{\mathscr{C}} (\underset{\mathbf{g}f}{\Leftarrow} \tau'')^\ell \, v : \boldsymbol{\tau}'$.

That satisfies (v′).

Then let

(5) $\Sigma'_1 = \Sigma_1, [\Sigma_{21}|_{\mathsf{A}}]^\ell, \ell : \mathbb{B}$,

(6) $\Sigma'_{21} = [\Sigma_{21}]^\ell, \ell : \mathbb{B}$,

(7) $\Sigma'_{22} = \Sigma_{22}, [\Sigma_{21}|_{\mathsf{A}}]^\ell, \ell : \mathbb{B}$, and

(8) $\Sigma' = \Sigma|_{\mathsf{U}}, \Sigma_1|_{\mathsf{A}}, \Sigma_{22}|_{\mathsf{A}}, [\Sigma_{21}|_{\mathsf{A}}]^\ell, \ell : \mathbb{B}$.

It should be apparent that $\Sigma' = \Sigma'_1 \boxplus \Sigma'_{21} \boxplus \Sigma'_{22}$.

From (vi) and by weakening,

(9) $\mathrm{M}; \Sigma'_{22}; \bullet \rhd_{\mathscr{C}} \mathbf{E} : [\boldsymbol{\tau}']\boldsymbol{\tau}$.

That satisfies (vi′).

It remains to show (ii′): $\mathrm{M}; \Sigma'_1 \rhd s_2 : \Sigma'$. Then,

(10) $\mathrm{M}; \Sigma_1 \rhd s_1 : \Sigma$ by (ii), lemma B.4

(11) $\mathrm{M}; \Sigma_1 \rhd s_1 \uplus \{\ell \mapsto \text{BLSSD}\} : \Sigma, \ell : \mathbb{B}$ by RS-LOCC

(12) $\mathrm{M}; \Sigma_1 \rhd s_2 : \Sigma, \ell : \mathbb{B}$ by subst.

(13) $\mathrm{M}; \Sigma_1 \rhd s_2 : \Sigma|_{\mathsf{U}}, \Sigma_1|_{\mathsf{A}}, \Sigma_{21}|_{\mathsf{A}}, \Sigma_{22}|_{\mathsf{A}}, \ell : \mathbb{B}$ by lemma B.4

(14) $\mathrm{M}; \Sigma_1 \rhd s_2 : \Sigma|_{\mathsf{U}}, \Sigma_1|_{\mathsf{A}}, \Sigma_{21}|_{\mathsf{A}}, [\Sigma_{22}|_{\mathsf{A}}]^\ell, \ell : \mathbb{B}$ by lemma B.7

(15) $\mathrm{M}; \Sigma_1 \rhd s_2 : \Sigma'$ by (8)

(16) $\mathrm{M}; \Sigma'_1 \rhd s_2 : \Sigma'$ by weakening.

Case $(s, (\underset{\mathbf{g}f}{\Leftarrow} \tau'')((\{\boldsymbol{\alpha}\} \underset{f'\mathbf{g}'}{\Leftarrow})^\bullet \mathbf{v})) \xrightarrow[\mathrm{M}]{} (s, \mathbf{v})$**.**

By inversion of rules RCT-BOUNDARY and RAT-WRAPPED, we know that

(1) $\boldsymbol{\tau}' = \boldsymbol{\alpha}$ and

(2) $\mathrm{M}; \Sigma_{21}; \bullet \rhd_{\mathscr{C}} \mathbf{v} : \boldsymbol{\alpha}$.

Case $(s_1,((\underset{\mathbf{g}f}{\Leftarrow} \forall\, \alpha^{\mathsf{q}}.\, \tau_1)^{\ell}\, v)\, \boldsymbol{\tau}_2) \xrightarrow[\mathrm{M}]{} \mathit{check}(s_1,\ell,\langle \tau_1\rangle,(\underset{\mathbf{g}f}{\Leftarrow} \{\boldsymbol{\tau}_2^{\mathscr{A}}/\alpha^{\mathsf{q}}\}\tau_1)(v\, \boldsymbol{\tau}_2^{\mathscr{A}}),\mathbf{g})$**.**

Consider the three cases of *check*:

Case $\langle \forall\, \alpha^{\mathsf{q}}.\, \tau_1\rangle = \mathsf{U}$**.**

Then

(1) $(s_1,\mathbf{e}_1) \xrightarrow[\mathrm{M}]{} (s_1,(\underset{\mathbf{g}f}{\Leftarrow} \{\boldsymbol{\tau}_2^{\mathscr{A}}/\alpha^{\mathsf{q}}\}\tau_1)(v\, \boldsymbol{\tau}_2^{\mathscr{A}}))$.

By inversion of rules RCT-TApp and RCT-Sealed,

(2) $\mathrm{M};\Sigma_{21};\bullet \rhd_{\mathscr{A}} v : \forall\, \alpha^{\mathsf{q}}.\, \tau_1$ and

(3) $\boldsymbol{\tau}' = (\{\boldsymbol{\tau}_2^{\mathscr{A}}/\alpha^{\mathsf{q}}\}\tau_2)^{\mathscr{C}}$.

Then

(4) $\mathrm{M};\Sigma_{21};\bullet \rhd_{\mathscr{A}} v\, \boldsymbol{\tau}_2^{\mathscr{A}} : \{\boldsymbol{\tau}_2^{\mathscr{A}}/\alpha^{\mathsf{q}}\}\tau_1$ by RAT-TApp

(5) $\mathrm{M};\Sigma_{21};\bullet \rhd_{\mathscr{C}} (\underset{\mathbf{g}f}{\Leftarrow} \{\boldsymbol{\tau}_2^{\mathscr{A}}/\alpha^{\mathsf{q}}\}\tau_1)(v\, \boldsymbol{\tau}_2^{\mathscr{A}}) : (\{\boldsymbol{\tau}_2^{\mathscr{A}}/\alpha^{\mathsf{q}}\}\tau_1)^{\mathscr{C}}$ by RCT-Boundary.

Case $\langle \forall\, \alpha^{\mathsf{q}}.\, \tau_1\rangle = \mathsf{A}$ **and** $s_1 = s_1' \uplus \{\ell \mapsto \textsc{blssd}\}$**.**

Then

(1) $(s_1,\mathbf{e}_1) \xrightarrow[\mathrm{M}]{} (s_1' \uplus \{\ell \mapsto \textsc{dfnct}\},(\underset{\mathbf{g}f}{\Leftarrow} \{\boldsymbol{\tau}_2^{\mathscr{A}}/\alpha^{\mathsf{q}}\}\tau_1)(v\, \boldsymbol{\tau}_2^{\mathscr{A}}))$.

By inversion of rules RCT-TApp and RCT-Blessed,

(2) $\mathrm{M};\Sigma_{21}';\bullet \rhd_{\mathscr{A}} v : \forall\, \alpha^{\mathsf{q}}.\, \tau_1$ where

(3) $\Sigma_{21} = [\Sigma_{21}']^{\ell}, \ell : \mathbb{B}$ and

(4) $\boldsymbol{\tau}' = (\{\boldsymbol{\tau}_2^{\mathscr{A}}/\alpha^{\mathsf{q}}\}\tau_2)^{\mathscr{C}}$.

Then,

(5) $\mathrm{M};\Sigma_{21}',\ell : \mathbb{D};\bullet \rhd_{\mathscr{A}} v : \forall\, \alpha^{\mathsf{q}}.\, \tau_1$ by weakening,

(6) $\mathrm{M};\Sigma_{21}',\ell : \mathbb{D};\bullet \rhd_{\mathscr{A}} v\, \boldsymbol{\tau}_2^{\mathscr{A}} : \{\boldsymbol{\tau}_2^{\mathscr{A}}/\alpha^{\mathsf{q}}\}\tau_1$ by RAT-TApp

(7) $\mathrm{M};\Sigma_{21}',\ell : \mathbb{D};\bullet \rhd_{\mathscr{C}} (\underset{\mathbf{g}f}{\Leftarrow} \{\boldsymbol{\tau}_2^{\mathscr{A}}/\alpha^{\mathsf{q}}\}\tau_1)(v\, \boldsymbol{\tau}_2^{\mathscr{A}}) : (\{\boldsymbol{\tau}_2^{\mathscr{A}}/\alpha^{\mathsf{q}}\}\tau_1)^{\mathscr{C}}$ by RCT-Boundary.

This satisfies (ii′).

Now let

(8) $\Sigma_1' = \Sigma_1|_{\mathsf{A}}, \Sigma_{21}'|_{\mathsf{U}}, \ell : \mathbb{D}$,

(9) $\Sigma_{22}' = \Sigma_{22}|_{\mathsf{A}}, \Sigma_{21}'|_{\mathsf{U}}, \ell : \mathbb{D}$, and

(10) $\Sigma' = \Sigma_1|_{\mathsf{A}}, \Sigma_{22}|_{\mathsf{A}}, \Sigma'_{21}, \ell : \mathbb{D}$.

It should be apparent that $\Sigma' = \Sigma'_1 \boxplus \Sigma'_{21} \boxplus \Sigma'_{22}$.

Suppose some $\mathbf{e}_\circ$ and $\Sigma_\circ$ such that $\mathrm{M}; \Sigma_\circ; \bullet \rhd_{\mathscr{C}} \mathbf{e}_\circ : \boldsymbol{\tau}'$. Then,

(11) $\mathrm{M}; \Sigma_\circ \boxplus \Sigma_{22}; \bullet \rhd_{\mathscr{C}} \mathbf{E}[\mathbf{e}_\circ] : \boldsymbol{\tau}$ by (vi),

(12) $\mathrm{M}; \Sigma_\circ \boxplus \Sigma_{22}|_{\mathsf{A}}, \Sigma_{21}|_{\mathsf{U}}; \bullet \rhd_{\mathscr{C}} \mathbf{E}[\mathbf{e}_\circ] : \boldsymbol{\tau}$ by $\Sigma_{22} \sim_{\mathsf{U}} \Sigma_{21}$,

(13) $\mathrm{M}; \Sigma_\circ \boxplus \Sigma_{22}|_{\mathsf{A}}, ([\Sigma'_{21}]^\ell, \ell : \mathbb{B})|_{\mathsf{U}}; \bullet \rhd_{\mathscr{C}} \mathbf{E}[\mathbf{e}_\circ] : \boldsymbol{\tau}$ by (3),

(14) $\mathrm{M}; \Sigma_\circ \boxplus \Sigma_{22}|_{\mathsf{A}}, [\Sigma'_{21}]^\ell, \ell : \mathbb{B}; \bullet \rhd_{\mathscr{C}} \mathbf{E}[\mathbf{e}_\circ] : \boldsymbol{\tau}$ by def. $\Sigma|_{\mathsf{U}}$,

(15) $\mathrm{M}; \Sigma_\circ \boxplus \Sigma_{22}|_{\mathsf{A}}, \Sigma'_{21}|_{\mathsf{U}}, \ell : \mathbb{D}; \bullet \rhd_{\mathscr{C}} \mathbf{E}[\mathbf{e}_\circ] : \boldsymbol{\tau}$ by lemma B.15,

(16) $\mathrm{M}; \Sigma_\circ \boxplus \Sigma'_{22}; \bullet \rhd_{\mathscr{C}} \mathbf{E}[\mathbf{e}_\circ] : \boldsymbol{\tau}$ by (9).

This satisfies (vi′).

It remains to show that $\mathrm{M}; \Sigma'_1 \rhd s_2 : \Sigma'$:

(17) $\mathrm{M}; \Sigma_1 \rhd s'_1 \uplus \{\ell \mapsto \textsc{blssd}\} : \Sigma$ by (ii), subst,

(18) $\mathrm{M}; \Sigma_1|_{\mathsf{A}}, [\Sigma'_{21}]^\ell, \ell : \mathbb{B} \rhd s'_1 \uplus \{\ell \mapsto \textsc{blssd}\} : \Sigma_1|_{\mathsf{A}}, \Sigma_{22}|_{\mathsf{A}}, [\Sigma'_{21}]^\ell, \ell : \mathbb{B}$ by (3)

(19) $\mathrm{M}; \Sigma_1|_{\mathsf{A}}, \Sigma'_{21}|_{\mathsf{U}}, \ell : \mathbb{D} \rhd s'_1 \uplus \{\ell \mapsto \textsc{dfnct}\} : \Sigma_1|_{\mathsf{A}}, \Sigma_{22}|_{\mathsf{A}}, \Sigma'_{21}, \ell : \mathbb{D}$ by lemma B.15

(20) $\mathrm{M}; \Sigma'_1 \rhd s_2 : \Sigma'$ by (8, 10), subst.

Otherwise.

Then $(s_1, \mathbf{e}_1) \xrightarrow[\mathrm{M}]{} \mathsf{blame}\,\mathbf{g}$, and by rule RBLAME, $\mathsf{blame}\,\mathbf{g}$ has whatever type is needed.

Case $(s_1, ((\Leftarrow_{\mathbf{g}f} \tau_1 \xrightarrow{\mathsf{q}}_\circ \tau_2)^\ell\, v_1)\mathbf{v}_2) \xrightarrow[\mathrm{M}]{} \mathit{check}(s_1, \ell, \mathsf{q}, (\Leftarrow_{\mathbf{g}f} \tau_2)(v_1((\tau_1 \Leftarrow_{f\mathbf{g}})\mathbf{v}_2)), \mathbf{g})$**.**

Consider the three cases of *check*:

Case $\mathsf{q} = \mathsf{U}$**.**

Then

(1) $(s_1, \mathbf{e}_1) \xrightarrow[\mathrm{M}]{} (s_1, (\Leftarrow_{\mathbf{g}f} \tau_2)(v_1((\tau_1 \Leftarrow_{f\mathbf{g}})\mathbf{v}_2)))$

By inversion of rules RCT-APP and RCT-SEALED, there exist some $\Sigma_{211} \boxplus \Sigma_{212} \leftrightsquigarrow \Sigma_{21}$ such that

(2) $\mathrm{M}; \Sigma_{211}; \bullet \triangleright_{\mathscr{A}} v_1 : \tau_1 \xrightarrow{\mathsf{q}}\!\circ\, \tau_2$,

(3) $\mathrm{M}; \Sigma_{212}; \bullet \triangleright_{\mathscr{C}} \mathbf{v}_2 : \tau_1^{\mathscr{C}}$, and

(4) $\boldsymbol{\tau}' = \tau_2^{\mathscr{C}}$.

Then

(5) $\mathrm{M}; \Sigma_{212}; \bullet \triangleright_{\mathscr{A}} (\tau_1 \underset{f\,\mathbf{g}}{\Leftarrow}) \mathbf{v}_2 : \tau_1$ by (3–4), RAT-BOUNDARY

(6) $\mathrm{M}; \Sigma_{21}; \bullet \triangleright_{\mathscr{A}} v_1((\tau_1 \underset{f\,\mathbf{g}}{\Leftarrow}) \mathbf{v}_2) : \tau_2$ by (2, 5), RAT-APP.

Case $\mathsf{q} = \mathsf{A}$ **and** $s_1 = s_1' \uplus \{\ell \mapsto \text{BLSSD}\}$**.**

Then

(1) $(s_1, \mathbf{e}_1) \xrightarrow[\mathrm{M}]{} (s_1' \uplus \{\ell \mapsto \text{DFNCT}\}, (\underset{\mathbf{g}f}{\Leftarrow} \tau_2)(v_1((\tau_1 \underset{f\,\mathbf{g}}{\Leftarrow}) \mathbf{v}_2)))$

By inversion of rule RCT-APP, there exist some $\Sigma_{211} \boxplus \Sigma_{212} \leftrightsquigarrow \Sigma_{21}$ such that

(2) $\mathrm{M}; \Sigma_{211}; \bullet \triangleright_{\mathscr{C}} (\underset{\mathbf{g}f}{\Leftarrow} \tau_1 \xrightarrow{\mathsf{q}}\!\circ\, \tau_2) v_1 : \tau_1^{\mathscr{C}} \to \tau_2^{\mathscr{C}}$,

(3) $\mathrm{M}; \Sigma_{212}; \bullet \triangleright_{\mathscr{C}} \mathbf{v}_2 : \tau_1^{\mathscr{C}}$, and

(4) $\boldsymbol{\tau}' = \tau_2^{\mathscr{C}}$.

Then by inversion of rule RCT-BLESSED, there exists some Σ_{211}' such that

(5) $\mathrm{M}; \Sigma_{211}'; \bullet \triangleright_{\mathscr{A}} v_1 : \tau_1 \xrightarrow{\mathsf{q}}\!\circ\, \tau_2$ where

(6) $\Sigma_{211} = [\Sigma_{211}']^{\ell}, \ell : \mathbb{B}$.

Then for $\mathbf{v}_2$,

(7) $\mathrm{M}; \Sigma_{212}|_{\mathsf{A}}, \Sigma_{211}|_{\mathsf{U}}; \bullet \triangleright_{\mathscr{C}} \mathbf{v}_2 : \tau_1^{\mathscr{C}}$ by $\Sigma_{212} \sim_{\mathsf{U}} \Sigma_{211}$,

(8) $\mathrm{M}; \Sigma_{212}|_{\mathsf{A}}, [\Sigma_{211}']^{\ell}, \ell : \mathbb{B}; \bullet \triangleright_{\mathscr{C}} \mathbf{v}_2 : \tau_1^{\mathscr{C}}$ by (6),

(9) $\mathrm{M}; \Sigma_{212}|_{\mathsf{A}}, \Sigma_{211}'|_{\mathsf{U}}, \ell : \mathbb{D}; \bullet \triangleright_{\mathscr{C}} \mathbf{v}_2 : \tau_1^{\mathscr{C}}$ by lemma B.15.

(10) $\mathrm{M}; \Sigma_{212}|_{\mathsf{A}}, \Sigma_{211}'|_{\mathsf{U}}, \ell : \mathbb{D}; \bullet \triangleright_{\mathscr{A}} (\tau_1 \underset{f\,\mathbf{g}}{\Leftarrow}) \mathbf{v}_2 : \tau_1$ by RAT-BOUNDARY.

Then,

(11) $\mathrm{M}; \Sigma_{211}', \ell : \mathbb{D}; \bullet \triangleright_{\mathscr{A}} v_1 : \tau_1 \xrightarrow{\mathsf{q}}\!\circ\, \tau_2$ by weakening,

(12) $\mathrm{M}; \Sigma_{211}', \Sigma_{212}|_{\mathsf{A}}, \ell : \mathbb{D}; \bullet \triangleright_{\mathscr{A}} v_1((\tau_1 \underset{f\,\mathbf{g}}{\Leftarrow}) \mathbf{v}_2) : \tau_2$ by (10), RAT-APP,

(13) $\mathrm{M}; \Sigma'_{211}, \Sigma_{212}|_{\mathsf{A}}, \ell : \mathbb{D}; \bullet \rhd_{\mathscr{C}} (\underset{\mathbf{g}f}{\Leftarrow} \tau_2)(v_1((\tau_1 \underset{f\,\mathbf{g}}{\Leftarrow})\mathbf{v}_2)) : \tau_2^{\mathscr{C}}$ by RCT-BOUNDARY.

This satisfies (ii′).

Now let

(14) $\Sigma'_1 = \Sigma_1|_{\mathsf{A}}, \Sigma'_{211}|_{\mathsf{U}}, \ell : \mathbb{D}$,

(15) $\Sigma'_{21} = \Sigma'_{211}, \Sigma_{212}|_{\mathsf{A}}, \ell : \mathbb{D}$,

(16) $\Sigma'_{22} = \Sigma_{22}|_{\mathsf{A}}, \Sigma'_{211}|_{\mathsf{U}}, \ell : \mathbb{D}$, and

(17) $\Sigma' = \Sigma_1|_{\mathsf{A}}, \Sigma_{22}|_{\mathsf{A}}, \Sigma'_{211}, \Sigma_{212}|_{\mathsf{A}}, \ell : \mathbb{D}$.

It should be apparent that $\Sigma' = \Sigma'_1 \boxplus \Sigma'_{21} \boxplus \Sigma'_{22}$.

Suppose some $\mathbf{e}_\circ$ and $\Sigma_\circ$ such that $\mathrm{M}; \Sigma_\circ; \bullet \rhd_{\mathscr{C}} \mathbf{e}_\circ : \boldsymbol{\tau}'$. Then,

(18) $\mathrm{M}; \Sigma_\circ \boxplus \Sigma_{22}; \bullet \rhd_{\mathscr{C}} \mathbf{E}[\mathbf{e}_\circ] : \boldsymbol{\tau}$ by (vi),

(19) $\mathrm{M}; \Sigma_\circ \boxplus \Sigma_{22}|_{\mathsf{A}}, \Sigma_{21}|_{\mathsf{U}}; \bullet \rhd_{\mathscr{C}} \mathbf{E}[\mathbf{e}_\circ] : \boldsymbol{\tau}$ by $\Sigma_{22} \sim_{\mathsf{U}} \Sigma_{21}$,

(20) $\mathrm{M}; \Sigma_\circ \boxplus \Sigma_{22}|_{\mathsf{A}}, ([\Sigma'_{211}]^\ell, \Sigma_{212}|_{\mathsf{A}}, \ell : \mathbb{B})|_{\mathsf{U}}; \bullet \rhd_{\mathscr{C}} \mathbf{E}[\mathbf{e}_\circ] : \boldsymbol{\tau}$ by (6),

(21) $\mathrm{M}; \Sigma_\circ \boxplus \Sigma_{22}|_{\mathsf{A}}, [\Sigma'_{211}]^\ell, \ell : \mathbb{B}; \bullet \rhd_{\mathscr{C}} \mathbf{E}[\mathbf{e}_\circ] : \boldsymbol{\tau}$ by def. $\Sigma|_{\mathsf{U}}$,

(22) $\mathrm{M}; \Sigma_\circ \boxplus \Sigma_{22}|_{\mathsf{A}}, \Sigma'_{211}|_{\mathsf{U}}, \ell : \mathbb{D}; \bullet \rhd_{\mathscr{C}} \mathbf{E}[\mathbf{e}_\circ] : \boldsymbol{\tau}$ by lemma B.15,

(23) $\mathrm{M}; \Sigma_\circ \boxplus \Sigma'_{22}; \bullet \rhd_{\mathscr{C}} \mathbf{E}[\mathbf{e}_\circ] : \boldsymbol{\tau}$ by (16).

This satisfies (vi′).

It remains to show that $\mathrm{M}; \Sigma'_1 \rhd s_2 : \Sigma'$:

(24) $\mathrm{M}; \Sigma_1 \rhd s'_1 \uplus \{\ell \mapsto \mathrm{BLSSD}\} : \Sigma$ by (ii), subst,

(25) $\mathrm{M}; \Sigma_1|_{\mathsf{A}}, [\Sigma'_{211}]^\ell, \ell : \mathbb{B} \rhd s'_1 \uplus \{\ell \mapsto \mathrm{BLSSD}\} :$
$\Sigma_1|_{\mathsf{A}}, \Sigma_{22}|_{\mathsf{A}}, [\Sigma'_{211}]^\ell, \Sigma_{212}|_{\mathsf{A}}, \ell : \mathbb{B}$ by (6)

(26) $\mathrm{M}; \Sigma_1|_{\mathsf{A}}, \Sigma'_{211}|_{\mathsf{U}}, \ell : \mathbb{D} \rhd s'_1 \uplus \{\ell \mapsto \mathrm{DFNCT}\} :$
$\Sigma_1|_{\mathsf{A}}, \Sigma_{22}|_{\mathsf{A}}, \Sigma'_{211}, \Sigma_{212}|_{\mathsf{A}}, \ell : \mathbb{D}$ by lemma B.15

(27) $\mathrm{M}; \Sigma'_1 \rhd s_2 : \Sigma'$ by (14, 17), subst.

Otherwise.

Then $(s_1, \mathbf{e}_1) \xrightarrow[\mathrm{M}]{} \mathsf{blame}\,\mathbf{g}$, and by rule RBLAME, $\mathsf{blame}\,\mathbf{g}$ has whatever type is needed.

That completes the CXT-C case of $\xrightarrow[M]{}$. Now for the CXT-A case:

Case $\dfrac{(s_1, e_1) \xrightarrow[M]{} (s_2, e_2)}{(s_1, \mathbf{E}[e_1]) \xrightarrow[M]{} (s_2, \mathbf{E}[e_2])}$.

Then by lemma B.11, there exist some $\Sigma_{21} \boxplus \Sigma_{22} \leftrightsquigarrow \Sigma_2$ and τ' such that

(v) $M; \Sigma_{21}; \bullet \rhd_{\mathscr{A}} e_1 : \tau'$ and

(vi) $M; \Sigma_{22}; \bullet \rhd_{\mathscr{A}} E : [\tau']\tau$.

We need to find some $\Sigma' \rightsquigarrow \Sigma'_1 \boxplus \Sigma'_{21} \boxplus \Sigma'_{22}$ such that

(ii′) $M; \Sigma'_1 \rhd s_2 : \Sigma'$,

(v′) $M; \Sigma'_{21}; \bullet \rhd_{\mathscr{A}} e_2 : \tau'$, and

(vi′) $M; \Sigma'_{22}; \bullet \rhd_{\mathscr{A}} E : [\tau']\tau$.

Then by instantiating (vi′), we have $M; \Sigma'_{21} \boxplus \Sigma'_{22}; \bullet \rhd_{\mathscr{A}} E[e_2] : \tau$, and the rest follows by rule RCONF. (If $s_1 = s_2$, we let $\Sigma' = \Sigma$ and $\Sigma'_1 = \Sigma_1$ and $\Sigma'_{21} = \Sigma_{21}$ and $\Sigma'_{22} = \Sigma_{22}$; then it suffices to show that $M; \Sigma_{21}; \bullet \rhd_{\mathscr{A}} e_2 : \tau'$.)

By cases on the derivation of $(s_1, e_1) \xrightarrow[M]{} (s_2, e_2)$:

Case $(s_1, (\Lambda \alpha^{\mathfrak{q}}. v)\, \tau_2) \xrightarrow[M]{} (s_1, \{\alpha^{\mathfrak{q}}/\alpha^{\mathfrak{q}}\}v)$**).**

By inversion of rule RAT-TAPP, we know that

(1) $M; \Sigma_{21}; \bullet \rhd_{\mathscr{A}} \Lambda \alpha^{\mathfrak{q}}. v : \forall\, \alpha^{\mathfrak{q}}. \tau_1$ and

(2) $\langle \tau_2 \rangle \sqsubseteq \mathfrak{q}$ where

(3) $\tau' = \{\tau_2/\alpha^{\mathfrak{q}}\}\tau_1$.

Then, by inversion of rule RAT-TABS, we know that

(4) $M; \Sigma_{21}; \bullet \rhd_{\mathscr{A}} v : \tau_1$.

By (2), substitution $\{\tau_2/\alpha^{\mathfrak{q}}\}$ respects qualifiers, so by lemma 7.4, we then conclude that

(5) $M; \Sigma_{21}; \bullet \rhd_{\mathscr{A}} \{\tau_2/\alpha^{\mathfrak{q}}\}v : \{\tau_2/\alpha^{\mathfrak{q}}\}\tau_2$.

Case $(s_1, (\lambda x{:}\tau_x.\, e)\, v) \xrightarrow[\mathrm{M}]{} (s_1, \{v/x\}e)$**.**

As for $\mathrm{F}_{\mathscr{C}}$.

Case $\dfrac{(g : \tau'' = v) \in \mathrm{M}}{(s_1, g) \xrightarrow[\mathrm{M}]{} (s_1, v)}$**.**

As for $\mathrm{F}_{\mathscr{C}}$.

Case $\dfrac{(\mathbf{g} : \boldsymbol{\tau}'' = \mathbf{v}) \in \mathrm{M}}{(s, \mathbf{g}^f) \xrightarrow[\mathrm{M}]{} (s, ((\boldsymbol{\tau}'')^{\mathscr{A}} \underset{f\,\mathbf{g}}{\Leftarrow}) \mathbf{g})}$**.**

By inversion of rule RAT-ModC, $(\boldsymbol{\tau}'')^{\mathscr{A}} = \tau'$.

Then

(1) $\mathrm{M}; \Sigma_{21}; \bullet \rhd_{\mathscr{C}} \mathbf{f} : \boldsymbol{\tau}''$ by RCT-Mod,

(2) $\mathrm{M}; \Sigma_{21}; \bullet \rhd_{\mathscr{A}} (\tau'' \underset{f\,\mathbf{g}}{\Leftarrow}) \mathbf{g} : \tau'$ by RCT-Boundary.

Case $\dfrac{(\mathbf{g} :> \tau'' = \mathbf{g}') \in \mathrm{M}}{(s, \mathbf{g}^f) \xrightarrow[\mathrm{M}]{} (s, (\tau'' \underset{f\,\mathbf{g}}{\Leftarrow}) \mathbf{g}')}$**.**

By inversion of rule RAT-ModI, $\tau'' = \tau'$.

By (i),

(1) $\mathrm{M} \vdash \mathbf{g} :> \tau' = \mathbf{g}'$ okay,

and by inversion of rule Interface

(2) $(\mathbf{g}' : (\tau')^{\mathscr{C}} = \mathbf{v}) \in \mathrm{M}$.

Then,

(3) $\mathrm{M}; \Sigma_{21}; \bullet \rhd_{\mathscr{C}} \mathbf{g}' : (\tau')^{\mathscr{C}}$ by RCT-Mod

(4) $\mathrm{M}; \Sigma_{21}; \bullet \rhd_{\mathscr{A}} (\tau' \underset{f\,\mathbf{g}}{\Leftarrow}) \mathbf{g}' : \tau'$ by RAT-Boundary.

Case $(s_1, \mathsf{let}\, \langle x, y\rangle = \langle v_1, v_2\rangle \,\mathsf{in}\, e) \xrightarrow[\mathrm{M}]{} (s_1, \{v_1/x\}\{v_2/y\}e)$**.**

By inversion of rule RAT-LetPair,

(1) $\Sigma_{21} \rightsquigarrow \Sigma_{211} \boxplus \Sigma_{212}$,

(2) $\mathrm{M}; \Sigma_{211}; \bullet \rhd_{\mathscr{A}} \langle v_1, v_2\rangle : \tau_1 \otimes \tau_2$, and

(3) $\mathrm{M}; \Sigma_{212}; \bullet, x : \tau_1, y : \tau_2 \rhd_{\mathscr{A}} e : \tau'$.

Then by inversion of rule RAT-Pair,

(4) $\Sigma_{211} \rightsquigarrow \Sigma_{2111} \boxplus \Sigma_{2112}$,

(5) $\mathrm{M}; \Sigma_{2111}; \bullet \rhd_{\mathscr{A}} v_1 : \tau_1$, and

(6) $\mathrm{M}; \Sigma_{2112}; \bullet \rhd_{\mathscr{A}} v_2 : \tau_2$.

By lemma B.4, $\Sigma_{212} \boxplus \Sigma_{2112}$ is defined. Then,

(7) $\mathrm{M}; \Sigma_{212} \boxplus \Sigma_{2112}; \bullet \rhd_{\mathscr{A}} \{v_2/y\}e : \tau'$ by lemma 7.7,

(8) $\mathrm{M}; \Sigma_{21}; \bullet \rhd_{\mathscr{A}} \{v_1/x\}\{v_2/y\}e : \tau'$ by lemma 7.7.

Case $(s_1, \mathsf{new}\ v) \xrightarrow[\mathrm{M}]{} (s_1 \uplus \{\ell \mapsto v\}, \ell)$**.**

By inversion of rule RAT-NEW,

(1) $\mathrm{M}; \Sigma_{21}; \bullet \rhd_{\mathscr{A}} v : \tau''$ where

(2) $\tau' = \mathsf{ref}\ \tau''$.

Then let

(3) $\Sigma'_1 = \Sigma_1 \boxplus \Sigma_{21}$,

(4) $\Sigma'_{21} = \bullet, \ell : \tau''$,

(5) $\Sigma'_{22} = \Sigma_{22}$, and

(6) $\Sigma' = \Sigma, \ell : \tau''$.

Note that $s_2 = s_1 \uplus \{\ell \mapsto v\}$. Then,

(7) $\mathrm{M}; \Sigma_1 \rhd s_1 : \Sigma$ by (ii),

(8) $\mathrm{M}; \Sigma_1 \boxplus \Sigma_{21} \rhd s_1 \uplus \{\ell \mapsto v\} : \Sigma, \ell : \tau''$ by (1), RS-LOCA,

(9) $\mathrm{M}; \Sigma'_1 \rhd s_2 : \Sigma'$ by subst,

satisfying (ii′).

By rule RAT-LOC,

(10) $\mathrm{M}; \bullet, \ell : \tau''; \bullet \rhd_{\mathscr{A}} \ell : \mathsf{ref}\ \tau''$,

satisfying (v′).

Finally, by (vi) and $\Sigma'_{22} = \Sigma_{22}$,

(11) $\mathrm{M}; \Sigma'_{22}; \bullet \rhd_{\mathscr{A}} E : [\tau']\tau$,

for (vi′).

Case $(s_1' \uplus \{\ell \mapsto v_1\}, \mathsf{swap}\,\langle \ell, v_2\rangle) \xrightarrow[\mathrm{M}]{} (s_1' \uplus \{\ell \mapsto v_2\}, \langle \ell, v_1\rangle)$**.**

Note that $s_1 = s_1' \uplus \{\ell \mapsto v_1\}$ and $s_2 = s_1' \uplus \{\ell \mapsto v_2\}$.

By inversion of rule RAT-SWAP,

(1) $\mathrm{M}; \Sigma_{21}; \bullet \rhd_{\mathscr{A}} \langle \ell, v_2\rangle : \mathsf{ref}\ \tau_1 \otimes \tau_2$, where

(2) $\tau' = \mathsf{ref}\ \tau_2 \otimes \tau_1$.

By inversion of rules RAT-PAIR and RAT-LOC,

(3) $\Sigma_{21} = \Sigma_{212}, \ell : \tau_1$,

(4) $\mathrm{M}; \bullet, \ell : \tau_1; \bullet \rhd_{\mathscr{A}} \ell : \mathsf{ref}\ \tau_1, and$

(5) $\mathrm{M}; \Sigma_{212}; \bullet \rhd_{\mathscr{A}} v_2 : \tau_2$.

From (ii), we know that

(6) $\mathrm{M}; \Sigma_1 \rhd s_1' \uplus \{\ell \mapsto v_1\} : \Sigma_1 \boxplus \Sigma_{212} \boxplus \Sigma_{22}, \ell : \tau_1$.

Then by inversion of rule RS-LOCA,

(7) $\Sigma_1 \rightsquigarrow \Sigma_{11} \boxplus \Sigma_{12}$,

(8) $\mathrm{M}; \Sigma_{11} \rhd s_1' : \Sigma_{11} \boxplus \Sigma_{12} \boxplus \Sigma_{212} \boxplus \Sigma_{22}$, and

(9) $\mathrm{M}; \Sigma_{12}; \bullet \rhd_{\mathscr{A}} v_1 : \tau_1$.

Let

(10) $\Sigma_1' = \Sigma_{11} \boxplus \Sigma_{212}$,

(11) $\Sigma_{21}' = \Sigma_{12}, \ell : \tau_2$,

(12) $\Sigma_{22}' = \Sigma_{22}$, and

(13) $\Sigma' = \Sigma_{11} \boxplus \Sigma_{12} \boxplus \Sigma_{212} \boxplus \Sigma_{22}, \ell : \tau_2$.

Then

(14) $\mathrm{M}; \Sigma_{11} \boxplus \Sigma_{212} \rhd s_1' \uplus \{\ell \mapsto v_2\} : \Sigma_{11} \boxplus \Sigma_{12} \boxplus \Sigma_{212} \boxplus \Sigma_{22}, \ell : \tau_2$.
by (5, 8), RS-LOCA,

(15) $\mathrm{M}; \Sigma_1' \rhd s_2 : \Sigma'$ by subst,

for (ii′).

For (v′),

(16) $\mathrm{M}; \bullet, \ell : \tau_2; \bullet \rhd_{\mathscr{A}} \ell : \mathsf{ref}\ \tau_2$ by RAT-LOC,

(17) $\mathrm{M};\Sigma_{12},\ell:\tau_2;\bullet \rhd_{\mathscr{A}} \langle \ell, v_1\rangle : \mathsf{ref}\ \tau_2 \otimes \tau_1$ by RAT-PAIR.

Finally, for (vi′), note that $\Sigma'_{22} = \Sigma_{22}$.

Case $\dfrac{\mathbf{v} \neq (\underset{\mathbf{g}'f'}{\Leftarrow} \rho)^{\ell} v'}{(s,(\tau' \underset{f\mathbf{g}}{\Leftarrow})\mathbf{v}) \xrightarrow[\mathrm{M}]{} (s,(\tau' \underset{f\mathbf{g}}{\Leftarrow})^{\bullet}\mathbf{v})}$.

By inversion of rule RAT-BOUNDARY,

(1) $\mathrm{M};\Sigma_{21};\bullet \rhd_{\mathscr{C}} \mathbf{v} : (\tau')^{\mathscr{C}}$ and

(2) $\tau' = \tau$.

Because $\mathbf{v} \neq (\underset{\mathbf{g}'f'}{\Leftarrow} \rho)^{\ell} v'$, by the contrapositive of lemma B.16, we know that $(\tau')^{\mathscr{C}}$ is not of the form $\{\rho\}$; thus

(3) $(\tau')^{\mathscr{C}} \in \mathbf{W}$.

This is the necessary additional premise to show that

(4) $\mathrm{M};\Sigma_{21};\bullet \rhd_{\mathscr{A}} (\tau' \underset{f\mathbf{g}}{\Leftarrow})^{\bullet}\mathbf{v} : \tau'$

by rule RAT-WRAPPED.

Case $(s,(\tau' \underset{f\mathbf{g}}{\Leftarrow})((\underset{\mathbf{g}'f'}{\Leftarrow} \rho)^{\ell} v)) \xrightarrow[\mathrm{M}]{} \mathit{check}(s,\ell,\langle\rho\rangle,v,\mathbf{g}')$**.**

By inversion of rule RAT-BOUNDARY, $\mathrm{M};\Sigma_{21};\bullet \rhd_{\mathscr{C}} (\underset{\mathbf{g}'f'}{\Leftarrow} \rho)^{\ell} v : \tau^{\mathscr{C}}$ and $\tau' = \tau$.

Consider the three cases of *check*:

Case $\langle\rho\rangle = \mathsf{U}$**.**

Then

(1) $(s_1,e_1) \xrightarrow[\mathrm{M}]{} (s_1,v)$.

Then by inversion of rule RCT-SEALED,

(2) $\mathrm{M};\Sigma_{21};\bullet \rhd_{\mathscr{A}} v : \rho$ where

(3) $\tau^{\mathscr{C}} = \rho^{\mathscr{C}}$.

By lemma B.2, $\tau = \rho$. Thus, by (2) and substitution,

(4) $\mathrm{M};\Sigma_{21};\bullet \rhd_{\mathscr{A}} v : \tau$.

Case $\langle\rho\rangle = \mathsf{A}$ **and** $s_1 = s'_1 \uplus \{\ell \mapsto \mathrm{BLSSD}\}$**.**

Then

(1) $(s_1,e_1) \xrightarrow[\mathrm{M}]{} (s'_1 \uplus \{\ell \mapsto \mathrm{DFNCT}\}, v)$.

Then by inversion of rule RCT-BLESSED,

(2) $\mathrm{M}; \Sigma'_{21}; \bullet \rhd_{\mathscr{A}} v : \rho$ where

(3) $\Sigma_{21} = [\Sigma'_{21}]^{\ell}, \ell : \mathbb{B}$ and

(4) $\tau^{\mathscr{C}} = \rho^{\mathscr{C}}$.

By lemma B.2, $\tau = \rho$. Then,

(5) $\mathrm{M}; \Sigma'_{21}; \bullet \rhd_{\mathscr{A}} v : \tau$ by (2),

(6) $\mathrm{M}; \Sigma'_{21}, \ell : \mathbb{D}; \bullet \rhd_{\mathscr{A}} v : \tau$ by weakening.

This satisfies (ii′).

The remainder of this case follows the C-B$_\tau$ case above.

Otherwise.

Then $(s_1, e_1) \xrightarrow[\mathrm{M}]{} \mathsf{blame}\,\mathbf{g}$, and by rule RBLAME, $\mathsf{blame}\,\mathbf{g}$ has whatever type is needed.

Case $(s, ((\forall\, \alpha^{\mathfrak{q}}.\, \tau_1 \underset{f\,\mathbf{g}}{\Leftarrow})^{\bullet}\, \mathbf{v})\, \tau_2) \xrightarrow[\mathrm{M}]{} (s, (\{\tau_2/\alpha^{\mathfrak{q}}\}\tau_1 \underset{f\,\mathbf{g}}{\Leftarrow})(\mathbf{v}\, \tau_2^{\mathscr{C}}))$**.**

By inversion of rule RAT-TAPP,

(1) $\mathrm{M}; \Sigma_{21}; \bullet \rhd_{\mathscr{A}} (\forall\, \alpha^{\mathfrak{q}}.\, \tau_1 \underset{f\,\mathbf{g}}{\Leftarrow})^{\bullet}\, \mathbf{v} : \forall\, \alpha^{\mathfrak{q}}.\, \tau_1$ where

(2) $\tau = \{\tau_2/\alpha^{\mathfrak{q}}\}\tau_1$.

Then by inversion of rule RAT-WRAPPED,

(3) $\mathrm{M}; \Sigma_{21}; \bullet \rhd_{\mathscr{C}} \mathbf{v} : \forall\, \boldsymbol{\beta}.\, (\{\{\boldsymbol{\beta}\}/\alpha^{\mathfrak{q}}\}\tau_1)^{\mathscr{C}}$

Then,

(4) $\mathrm{M}; \Sigma_{21}; \bullet \rhd_{\mathscr{C}} \mathbf{v}\, \tau_2^{\mathscr{C}} : (\{\tau_2/\alpha^{\mathfrak{q}}\}\tau_1)^{\mathscr{C}}$ by RCT-TAPP,

(5) $\mathrm{M}; \Sigma_{21}; \bullet \rhd_{\mathscr{A}} (\{\tau_2/\alpha^{\mathfrak{q}}\}\tau_1 \underset{f\,\mathbf{g}}{\Leftarrow})(\mathbf{v}\, \tau_2^{\mathscr{C}}) : \{\tau_2/\alpha^{\mathfrak{q}}\}\tau_1$ by RAT-BOUNDARY.

Case $(s, ((\tau_1 \xrightarrow{\mathfrak{q}} \tau_2 \underset{f\,\mathbf{g}}{\Leftarrow})^{\bullet}\, \mathbf{v}_1)\, v_2) \xrightarrow[\mathrm{M}]{} (s, (\tau_2 \underset{f\,\mathbf{g}}{\Leftarrow})(\mathbf{v}_1\, ((\underset{\mathbf{g}\,f}{\Leftarrow})\, \tau_1)\, v_2)))$**.**

By inversion of rule RAT-APP,

(1) $\Sigma_{21} \rightsquigarrow \Sigma_{211} \boxplus \Sigma_{212}$,

(2) $\mathrm{M}; \Sigma_{211}; \bullet \rhd_{\mathscr{A}} (\tau_1 \xrightarrow{\mathfrak{q}} \tau_2 \underset{f\,\mathbf{g}}{\Leftarrow})^{\bullet}\, \mathbf{v}_1 : \tau_1 \xrightarrow{\mathfrak{q}} \tau_2$ and

(3) $\mathrm{M}; \Sigma_{222}; \bullet \rhd_{\mathscr{A}} v_2 : \tau_1$ where

(4) $\tau = \tau_2$.

Then by inversion of rule RAT-WRAPPED,

(5) $\mathrm{M}; \Sigma_{211}; \bullet \rhd_{\mathscr{C}} \mathbf{v}_1 : \tau_1^{\mathscr{C}} \to \tau_2^{\mathscr{C}}$.

Then,

(6) $\mathrm{M}; \Sigma_{222}; \bullet \rhd_{\mathscr{C}} (\underset{\mathbf{g}f}{\Leftarrow} \tau_1) v_2 : \tau_1^{\mathscr{C}}$ by RCT-BOUNDARY,

(7) $\mathrm{M}; \Sigma_{22}; \bullet \rhd_{\mathscr{C}} \mathbf{v}_1((\underset{\mathbf{g}f}{\Leftarrow} \tau_1) v_2) : \tau_2^{\mathscr{C}}$ by RCT-APP,

(8) $\mathrm{M}; \Sigma_{22}; \bullet \rhd_{\mathscr{A}} (\tau_2 \underset{f\mathbf{g}}{\Leftarrow})(\mathbf{v}_1((\underset{\mathbf{g}f}{\Leftarrow} \tau_1) v_2)) : \tau_2$ by RAT-BOUNDARY.

Note that $\tau_2 = \tau$. □

B.4 Progress

In this section I prove a progress lemma, starting with several definitions.

DEFINITION B.17 (Closed configurations and module contexts)**.**

*A configuration C is **closed** when all locations in the expression and the store are mapped by the store. A module context* M *is **closed** when all module names occurring in* M *are also defined in* M*. A configuration C is **closed with respect to a module context** *M *when C is closed and all module names occurring in C are defined in* M*.*

DEFINITION B.18 (Redexes)**.**

*In the definition of the relation $\xrightarrow[\mathrm{M}]{}$, every rule has either the form $(s, \mathbf{e}) \xrightarrow[\mathrm{M}]{} C'$ or the form $(s, e) \xrightarrow[\mathrm{M}]{} C'$. We call the expressions $\mathbf{e}$ and e, respectively, $F_{\mathscr{C}}$ **redexes** and $F^{\mathscr{A}}$ **redexes**, and denote the classes of redexes with the metasyntactic variables $\mathbf{R}$ and R, respectively.*

LEMMA B.19 (Redexes and evaluation contexts)**.**

If $(s_1, \mathbf{e}_1) \xmapsto[\mathrm{M}]{} (s_2, \mathbf{e}_2)$, then either:

- *We can decompose $\mathbf{e}_1 = \mathbf{E}[\mathbf{R}]$ and $\mathbf{e}_2 = \mathbf{E}[\mathbf{e}_2']$. Then for any other evaluation context $\mathbf{E}'[]_{\mathscr{C}}$, we have that $(s_1, \mathbf{E}'[\mathbf{R}]) \xmapsto[\mathrm{M}]{} (s_2, \mathbf{E}'[\mathbf{e}_2'])$ as well.*

- *We can decompose* $\mathbf{e}_1 = \mathbf{E}[R]$ *and* $\mathbf{e}_2 = \mathbf{E}[e_2']$. *Then for any other evaluation context* $\mathbf{E}'[]_{\mathscr{A}}$, *we have that* $(s_1, \mathbf{E}'[R]) \longmapsto_{\mathrm{M}} (s_2, \mathbf{E}'[e_2'])$ *as well.*

Proof. By cases on the derivation of $(s_1, \mathbf{e}_1) \longmapsto_{\mathrm{M}} (s_2, \mathbf{e}_2)$. □

DEFINITION B.20 ($\mathrm{F}^{\mathscr{A}}$ reduction)**.**

While configurations contain a top-level $F_{\mathscr{C}}$ *expression, it will be helpful to define the reduction relation* $\longmapsto_{\mathrm{M}}$ *for configurations with an* $F^{\mathscr{A}}$ *expression—of the form* (s, e)*—as follows:*

$$\frac{(s,e) \longrightarrow_{\mathrm{M}} (s',e')}{(s,E[e]) \longmapsto_{\mathrm{M}} (s',E[e'])} \qquad \frac{(s,\mathbf{e}) \longrightarrow_{\mathrm{M}} (s',\mathbf{e}')}{(s,E[\mathbf{e}]) \longmapsto_{\mathrm{M}} (s',E[\mathbf{e}'])}$$

LEMMA B.21 ($\mathrm{F}^{\mathscr{A}}$ reduction)**.**

If $(s,e) \longmapsto_{\mathrm{M}} (s',e')$ *then* $(s,\mathbf{E}[e]) \longmapsto_{\mathrm{M}} (s',\mathbf{E}[e'])$.

Proof. By cases on $(s,e) \longmapsto_{\mathrm{M}} (s',e')$, following definition B.20:

Case $\dfrac{(s,e_0) \longrightarrow_{\mathrm{M}} (s',e_0')}{(s,E'[e_0]) \longmapsto_{\mathrm{M}} (s',E'[e_0'])}$**.**

Then $\mathbf{E}[E']$ is an evaluation context, and $\mathbf{E}[E'[e_0]] \longmapsto_{\mathrm{M}} \mathbf{E}[E'[e_0']]$ by rule CXT-A.

Case $\dfrac{(s,\mathbf{e}_0) \longrightarrow_{\mathrm{M}} (s',\mathbf{e}_0')}{(s,E'[\mathbf{e}_0]) \longmapsto_{\mathrm{M}} (s',E'[\mathbf{e}_0'])}$**.**

Then $\mathbf{E}[E']$ is an evaluation context, and $\mathbf{E}[E'[\mathbf{e}_0]] \longmapsto_{\mathrm{M}} \mathbf{E}[E'[\mathbf{e}_0']]$ by rule CXT-C. □

LEMMA 7.10 (Uniform evaluation, restated from p. 185)**.**

For any C *closed with respect to* M*, either* C *is faulty or an answer, or there exists some* C' *closed with respect to* M *such that* $C \longmapsto_{\mathrm{M}} C'$.

Proof. If $C = \mathsf{blame}\,\mathbf{g}$ for some module $\mathbf{g}$, then C is an answer. Otherwise, C must be of the form $(s, \mathbf{e})$.

We therefore generalize our induction hypothesis as follows.

1. For any s and $\mathbf{e}$, if the configuration $(s, \mathbf{e})$ is closed with respect to closed M, then one of:

   (Q) $\mathbf{e}$ is faulty with respect to s (and hence the configuration is faulty),

   (A) $\mathbf{e}$ is a value (and hence the configuration is an answer), or

   (R) there exist some s' and $\mathbf{e}'$ such that $(s, \mathbf{e})$ |-> $(s', \mathbf{e}')$, which is also closed with respect to M (let $C' = (s', \mathbf{e}')$).

2. For any s and e, if the configuration (s, e) is closed with respect to closed M, then one of:

   (Q) e is faulty with respect to s,

   (A) e is a value, or

   (R) there exist some s' and e' such that $(s, e) \xmapsto[\mathrm{M}]{} (s', e')$, which is also closed with respect to M.

We proceed by mutual induction on the structures of $\mathbf{e}$ and e:

1. Cases on $\mathbf{e}$:

   Case v.

   Then (A).

   Case x.

   Vacuous, because $\mathbf{e}$ is closed.

   Case g.

   Because C is closed in M, we know that there exists some $(\mathbf{f} : \boldsymbol{\tau} = \mathbf{v}) \in \mathrm{M}$, thus $(s, \mathbf{g}) \xmapsto[\mathrm{M}]{} (s, \mathbf{v})$. Because M is closed, we know that $\mathbf{v}$ is closed in M. Hence (R).

   Case $\mathbf{e}_1\,\boldsymbol{\tau}_2$.

   Consider the induction hypothesis at $\mathbf{e}_1$, noting that $\mathbf{E}_1 = []\,\boldsymbol{\tau}_2$ is an evaluation context.

(Q) Then (Q).

(A) Let $\mathbf{v}_1 = \mathbf{e}_1$. Now by cases on $\mathbf{v}_1$:

Case $\Lambda\boldsymbol{\alpha}.\mathbf{v}'_1$.

Then $(s,\mathbf{e}) \xrightarrow[\mathrm{M}]{} (s,\{\boldsymbol{\tau}_2/\boldsymbol{\alpha}\}\mathbf{v}'_1)$, hence (R).

Case $\lambda\mathbf{x}{:}\boldsymbol{\tau}'.\mathbf{e}'$.

Then $\mathbf{v}_1 \in \mathbf{Q}^{\Lambda}$, so (Q).

Case $(\underset{\mathbf{g}f}{\Leftarrow}\tau)^{\ell}\, v'_1$.

If $\tau = \forall\, \alpha^{\mathsf{q}}.\tau'$ for some α^{q} and τ', then

$$(s,\mathbf{e}) \xrightarrow[\mathrm{M}]{} \mathit{check}(s,\ell,\langle\tau'\rangle,(\underset{\mathbf{g}f}{\Leftarrow}\{\boldsymbol{\tau}^{\mathscr{A}}/\alpha^{\mathsf{q}}\}\tau')(v'_1\,\boldsymbol{\tau}^{\mathscr{A}}),\mathbf{g}),$$

hence (R); otherwise (Q).

(R) That is, $(s,\mathbf{e}_1) \xrightarrow[\mathrm{M}]{} (s',\mathbf{e}'_1)$. Then $(s,\mathbf{E}_1[\mathbf{e}_1]) \xrightarrow[\mathrm{M}]{} (s',\mathbf{E}_1[\mathbf{e}'_1])$ by lemma B.19, hence, (R).

Case $\mathbf{e}_1\,\mathbf{e}_2$.

Consider first the induction hypothesis at $\mathbf{e}_1$, noting that $\mathbf{E}_1 = []\,\mathbf{e}_2$ is an evaluation context.

(Q) Then (Q).

(A) Let $\mathbf{v}_1 = \mathbf{e}_1$, and note that $\mathbf{E}_2 = \mathbf{v}_1\,[]$ is an evaluation context. We now apply the induction hypothesis to $\mathbf{e}_2$:

(Q) Then (Q).

(A) Let $\mathbf{v}_2 = \mathbf{e}_2$. Now by cases on $\mathbf{v}_1$:

Case $\Lambda\boldsymbol{\alpha}.\mathbf{v}'_1$.

Then (Q).

Case $\lambda\mathbf{x}{:}\boldsymbol{\tau}.\mathbf{e}'_1$.

Then $(s,\mathbf{e}) \xrightarrow[\mathrm{M}]{} (s,\{\mathbf{v}_2/\mathbf{x}\}\mathbf{e}'_1)$, hence (R).

Case $(\underset{\mathbf{g}f}{\Leftarrow}\tau)^{\ell}\, v'_1$.

If $\tau = \tau_1 \xrightarrow[\mathsf{q}]{} \tau_2$, then

$$(s,\mathbf{e}) \xrightarrow[\mathrm{M}]{} \mathit{check}(s,\ell,\langle\tau'\rangle,(\underset{\mathbf{g}f}{\Leftarrow}\tau_2)(v'_1((\tau_1 \underset{f\mathbf{g}}{\Leftarrow})\mathbf{v}_2)),\mathbf{g})$$

hence (R); otherwise (Q).

(R) That is, $(s,\mathbf{e}_2) \xrightarrow[\mathrm{M}]{} (s',\mathbf{e}'_2)$. Then $(s,\mathbf{E}_2[\mathbf{e}_2]) \xrightarrow[\mathrm{M}]{} (s',\mathbf{E}_2[\mathbf{e}'_2])$ by lemma B.19, hence, (R).

(R) That is, $(s, \mathbf{e}_1) \xrightarrow[\mathrm{M}]{} (s', \mathbf{e}'_1)$. Then $(s, \mathbf{E}_1[\mathbf{e}_1]) \xrightarrow[\mathrm{M}]{} (s', \mathbf{E}_1[\mathbf{e}'_1])$ by lemma B.19, hence, (R).

Case $f^{\mathbf{g}}$.

Because C is closed in M, we know that there exists some $(f : \tau = v) \in \mathrm{M}$, thus $(s, f^{\mathbf{g}}) \xrightarrow[\mathrm{M}]{} (s, (\Leftarrow_{\mathbf{g}f} \tau) f)$; which is closed in M as well. Hence (R).

Case $(\Leftarrow_{\mathbf{g}f} \tau) e_1$.

Apply part 2 of the induction hypothesis to (s, e_1), noting that $\mathbf{E}' = (\Leftarrow_{\mathbf{g}f} \tau)[]$ is an evaluation context:

(Q) Then (Q).

(A) Let $v = e_1$. Then by cases on v:

Case $(\{\boldsymbol{\alpha}\} \Leftarrow_{f'\mathbf{g}'})^{\bullet} \mathbf{v}'$.

Then $(s, \mathbf{e}) \xrightarrow[\mathrm{M}]{} (s, \mathbf{v}')$ by rule C-UNWRAP.

Otherwise.

Then $(s, \mathbf{e}) \xrightarrow[\mathrm{M}]{} (s \uplus \{\ell \mapsto \mathrm{BLSSD}\}, (\Leftarrow_{\mathbf{g}f} \tau)^{\ell} v)$ by rule C-SEAL.

Hence, (R).

(R) That is, $(s, e_1) \xrightarrow[\mathrm{M}]{} (s', e'_1)$. Then by lemma B.21, $(s, \mathbf{e}) \xrightarrow[\mathrm{M}]{} (s', (\Leftarrow_{\mathbf{g}f} \tau) e'_1)$, so (R).

2. Cases on e:

Case v.

Then (A).

Case f.

As for $\mathrm{F}_{\mathscr{C}}$ in the previous case.

Case $e_1 \tau_2$.

This is the same as part 1, except that in the case where e_1 is a value, it might also be a location or pair. Both of these are in Q^{Λ}, hence for both of these possibilities, (Q).

Case $e_1\, e_2$**.**

This is the same as part 1, except that in the case where e_1 is a value, it might also be a location or pair. Both of these are in Q^λ, hence for both of these possibilities, (Q).

Case $\mathsf{new}\, e_1$**.**

By the induction hypothesis on e_1, noting that $E_1 = \mathsf{new}[]$ is an evaluation context:

(Q) Then (Q).

(A) Let $v = e_1$. Then $(s,e) \xrightarrow[\mathrm{M}]{} (s \uplus \{\ell \mapsto v\}, \ell)$.

(R) That is, $(s,e_1) \xrightarrow[\mathrm{M}]{} (s,e_1')$. Then $(s,e) \xrightarrow[\mathrm{M}]{} (s, \mathsf{new}\, e_1')$, hence (R).

Case $\mathsf{swap}\, e_1$**.**

By the induction hypothesis on e_1, noting that $E_1 = \mathsf{swap}[]$ is an evaluation context:

(Q) Then (Q).

(A) Let $v = e_1$. There are two possibilities, by cases on v:

Case $\langle \ell, v_2 \rangle$ where $\exists s', v_1$ s.t. $s = s' \uplus \{\ell \mapsto v_1\}$**.**

Then $(s,e) \xrightarrow[\mathrm{M}]{} (s' \uplus \{\ell \mapsto v_2\}, \langle \ell, v_1 \rangle)$.

Otherwise.

Then $v \in Q_s^{\leftrightarrows}$, hence (Q).

(R) That is, $(s,e_1) \xrightarrow[\mathrm{M}]{} (s,e_1')$. Then $(s,e) \xrightarrow[\mathrm{M}]{} (s, \mathsf{swap}\, e_1')$, hence (R).

Case $\mathsf{let}\, \langle x, y \rangle = e_1\, \mathsf{in}\, e_2$**.**

By the induction hypothesis at e_1, noting that $\mathsf{let}\, \langle x, y \rangle = []\, \mathsf{in}\, e_2$ is an evaluation context:

(Q) Then (Q).

(A) Then e_1 is a value. If e_1 has the form $\langle v_1, v_2 \rangle$, then $(s,e) \xrightarrow[\mathrm{M}]{} (s, \{v_1/x\}\{v_2/y\}e_2)$, hence (R). Otherwise, $e_1 \in Q^\otimes$, so (Q).

(R) Then (R).

Case $\mathbf{g}^f$**.**

Because C is closed in M, we know that one of:

- there is some $(\mathbf{g} : \boldsymbol{\tau} = \mathbf{v}) \in \mathrm{M}$, and thus $(s,e) \xrightarrow[\mathrm{M}]{} (s, (\boldsymbol{\tau}^{\mathscr{A}} \underset{f\,\mathbf{g}}{\Leftarrow})\mathbf{f})$, which is also closed in M; or

- there is some $(\mathbf{g} :> \tau = \mathbf{g}') \in \mathrm{M}$, and thus $(s,e) \xrightarrow[\mathrm{M}]{} (s,(\tau \underset{f\,\mathbf{g}}{\Leftarrow})\mathbf{g}')$, which is also closed in M;

hence (R).

Case $(\tau \underset{f\,\mathbf{g}}{\Leftarrow})\mathbf{e}_1$**.**

Apply part 1 of the induction hypothesis to $(s,\mathbf{e}_1)$, noting that $E' = (\tau \underset{f\,\mathbf{g}}{\Leftarrow})[]$ is an evaluation context:

(Q) Then (Q).

(A) Let $\mathbf{v} = \mathbf{e}_1$. Then by cases on $\mathbf{v}$:

Case $(\underset{\mathbf{g}'\,f'}{\Leftarrow} \rho)^{\ell}\, v'$**.**

Then $(s,e) \xrightarrow[\mathrm{M}]{} \mathit{check}(s,\ell,\langle\rho\rangle,v',\mathbf{g}')$, by rule A-UNSEAL.

Otherwise.

Then $(s,e) \xrightarrow[\mathrm{M}]{} (s,(\tau \underset{f\,\mathbf{g}}{\Leftarrow})^{\bullet}\,\mathbf{v})$ by rule A-WRAP.

Hence, (R).

(R) That is, $(s,\mathbf{e}_1) \xrightarrow[\mathrm{M}]{} (s',\mathbf{e}_1')$. Then by lemma B.21, $(s,e) \xrightarrow[\mathrm{M}]{} (s',(\tau \underset{f\,\mathbf{g}}{\Leftarrow})\mathbf{e}_1')$, so (R). □

LEMMA 7.11 (Faulty expressions are ill-typed, restated from p. 185)**.**

1. *For expression* $\mathbf{Q}_s$ *faulty with respect to s, there exist no* M, Σ_1, Σ_2, *and* $\boldsymbol{\tau}$ *such that*

   - $\mathrm{M};\Sigma_1 \rhd s : \Sigma_1 \boxplus \Sigma_2$ *and*
   - $\mathrm{M};\Sigma_2;\bullet \rhd_{\mathscr{C}} \mathbf{Q}_s : \boldsymbol{\tau}$.

2. *For expression* Q_s *faulty with respect to s, there exist no* M, Σ_1, Σ_2, *and* τ *such that*

   - $\mathrm{M};\Sigma_1 \rhd s : \Sigma_1 \boxplus \Sigma_2$ *and*
   - $\mathrm{M};\Sigma_2;\bullet \rhd_{\mathscr{A}} Q_s : \tau$.

Proof by contradiction. We proceed by mutual induction on the structure of $\mathbf{Q}_s$ and Q_s, taking the statement of the lemma as the induction hypothesis.

1. Suppose that $\mathrm{M};\Sigma_2;\bullet \rhd_{\mathscr{C}} \mathbf{Q}_s : \boldsymbol{\tau}$. Then by cases on $\mathbf{Q}_s$:

Case $\mathbf{Q}^{\Lambda}\,\boldsymbol{\tau}_2$.

This types only by rule RCT-TAPP, which requires that $\mathbf{Q}^{\Lambda}$ have a type $\forall\,\boldsymbol{\alpha}.\,\boldsymbol{\tau}_1$. By cases on $\mathbf{Q}^{\Lambda}$:

Case $\lambda\mathbf{x}{:}\boldsymbol{\tau}'.\,\mathbf{e}'$.

This has a function type.

Case $(\underset{\mathbf{g}f}{\Leftarrow}\tau')^{\ell}\,v'$ where $\tau' \not\approx \forall\,\alpha^{\mathfrak{q}}.\,\tau''$.

This does not have a universal type.

Thus, both cases lead to a contradiction.

Case $\mathbf{Q}^{\lambda}\,\mathbf{v}_2$.

This types only by rule RCT-APP, which requires that $\mathbf{Q}^{\lambda}$ have a type $\boldsymbol{\tau}_1 \to \boldsymbol{\tau}_2$. By cases on $\mathbf{Q}^{\lambda}$:

Case $\Lambda\boldsymbol{\alpha}.\,\mathbf{v}'$.

This has a universal type.

Case $(\underset{\mathbf{g}f}{\Leftarrow}\tau')^{\ell}\,v'$ where $\tau' \not\approx \tau'_1 \xrightarrow{\mathfrak{q}} \tau'_2$.

This does not have a function type.

Thus, both cases lead to a contradiction.

Case $\mathbf{E}[\mathbf{Q}'_s]$.

By our assumption,

(1) $\mathrm{M};\Sigma_1 \rhd s : \Sigma_1 \boxplus \Sigma_2$ and

(2) $\mathrm{M};\Sigma_2;\bullet \rhd_{\mathscr{C}} \mathbf{E}[\mathbf{Q}'_s] : \boldsymbol{\tau}$.

Then by lemma B.11,

(3) $\mathrm{M};\Sigma_{21};\bullet \rhd_{\mathscr{C}} \mathbf{Q}'_s : \boldsymbol{\tau}'$

for some $\boldsymbol{\tau}'$ and $\Sigma_2 \rightsquigarrow \Sigma_{21} \boxplus \Sigma_{22}$.

By weakening,

(4) $\mathrm{M};\Sigma_2;\bullet \rhd_{\mathscr{C}} \mathbf{Q}'_s : \boldsymbol{\tau}'$,

but by the induction hypothesis (part 1) this cannot be so.

Case $\mathbf{E}[Q'_s]$.

As in the previous case, using part 2 of the induction hypothesis.

2. Suppose that $\mathrm{M};\Sigma_1;\bullet \rhd_{\mathscr{A}} Q_s : \tau$. Then by cases on Q_s:

Case $Q^{\Lambda}\,\tau_2$**.**

By induction on the derivation, this must type by rule RAT-TAPP. (The two rules that apply are RAT-TAPP and RAT-SUBSUME, and apply the inner induction hypothesis to the latter until reaching RAT-TAPP.) Rule RAT-TAPP requires that Q^{Λ} have a type $\forall\,\alpha^{\mathsf{q}}.\,\tau_1$. By cases on Q^{Λ}:

Case $\lambda x{:}\tau'.\,e'$**.**

This has a function type.

Case $\langle v_1, v_2\rangle$**.**

This has a product type.

Case ℓ**.**

This has a reference type.

Case $(\tau' \underset{f\,\mathbf{g}}{\Leftarrow})\mathbf{e}'$ where $\tau' \not\approx \forall\,\beta^{\mathsf{q}'}.\,\tau''$**.**

This does not have a universal type.

Thus, all cases lead to a contradiction.

Case $Q^{\lambda}\,v_2$**.**

As in the previous case, showing the contradiction that Q^{λ} must have a function type but cannot.

Case $\mathsf{let}\,\langle x, y\rangle = Q^{\otimes}\,\mathsf{in}\,e_2$**.**

As in the previous case, showing the contradiction that $Q^{\otimes}$ must have a product type but cannot.

Case $\mathsf{swap}\,Q_s^{\leftrightarrows}$**.**

This types only by rule RAT-SWAP (again by induction on the derivation to deal with RAT-SUBSUME), which requires that $Q_s^{\leftrightarrows}$ have a type $\mathsf{ref}\,\tau_1 \otimes \tau_2$. By lemma B.16, for $Q_s^{\leftrightarrows}$ to have a product type it must be $\langle v_1, v_2\rangle$ for some v_1 and v_2. Then by inversion of rule RAT-PAIR, v_1 must have type $\mathsf{ref}\,\tau_1$, so by lemma B.16 again, v_1 must be some location ℓ. By inversion of rule RAT-LOC, $\ell : \tau_1 \in \Sigma_1$. The only possibility for $Q_s^{\leftrightarrows}$ in the shape of a pair is $\langle v_1, v_2\rangle$ such that v_1 is not a location $\ell \in \mathrm{dom}\,\Sigma$, which contradicts that $\ell : \tau_1 \in \Sigma_1$, since $\Sigma \rightsquigarrow \Sigma_1 \boxplus \Sigma_2$.

Case $E[Q'_s]$.

By our assumption,

(1) $\mathrm{M}; \Sigma_1 \rhd s : \Sigma_1 \boxplus \Sigma_2$ and

(2) $\mathrm{M}; \Sigma_2; \bullet \rhd_{\mathscr{A}} E[Q'_s] : \tau$.

Then by lemma B.11,

(3) $\mathrm{M}; \Sigma_{21}; \bullet \rhd_{\mathscr{A}} Q'_s : \tau'$

for some τ' and $\Sigma_2 \rightsquigarrow \Sigma_{21} \boxplus \Sigma_{22}$.

By weakening,

(4) $\mathrm{M}; \Sigma_2; \bullet \rhd_{\mathscr{A}} Q'_s : \tau'$,

but by the induction hypothesis (part 2) this cannot be so.

Case $E[\mathbf{Q}'_s]$.

As in the previous case, but using part 1 of the induction hypothesis.

□

The proof concludes with corollary 7.12 (Progress) on p. 185 and theorem 7.13 (Strong soundness) on p. 186.

APPENDIX C

Additional Proofs for Chapter 8

C.1 Properties of λ^{URAL}

In this section, I state and prove several propositions about λ^{URAL}, including two lemmas from §8.4.2.

LEMMA C.1 (Qualifier subsumption transitivity)**.**

If $\Delta \vdash \xi_1 \preceq \xi'$ *and* $\Delta \vdash \xi' \preceq \xi_2$ *then* $\Delta \vdash \xi_1 \preceq \xi_2$.

Proof. By cases on the derivation of $\Delta \vdash \xi_1 \preceq \xi'$:

Case $\dfrac{\Delta \vdash \xi : \mathsf{QUAL}}{\Delta \vdash \mathsf{U} \preceq \xi}$**.**

That is, $\xi_1 = \mathsf{U}$, so by rule QSUB-BOT.

Case $\dfrac{\Delta \vdash \xi : \mathsf{QUAL}}{\Delta \vdash \xi \preceq \mathsf{L}}$**.**

That is, $\xi' = \mathsf{L}$. By cases on the derivation of $\Delta \vdash \xi' \preceq \xi_2$:

Case $\dfrac{\Delta \vdash \xi : \mathsf{QUAL}}{\Delta \vdash \mathsf{U} \preceq \xi}$**.**

That is, $\xi' = \mathsf{U}$, but since $\xi' = \mathsf{L}$, this case is vacuous.

Case $\dfrac{\Delta \vdash \xi : \mathsf{QUAL}}{\Delta \vdash \xi \preceq \mathsf{L}}$**.**

That is, $\xi_2 = \mathsf{L}$, so by rule QSUB-TOP.

Case $\dfrac{\Delta \vdash \xi : \mathsf{QUAL}}{\Delta \vdash \xi \preceq \xi}$.

That is, $\xi_2 = \xi' = \mathsf{L}$, so by rule QSUB-TOP.

Case $\dfrac{\Delta \vdash \xi : \mathsf{QUAL}}{\Delta \vdash \xi \preceq \xi}$.

That is, $\xi_1 = \xi'$. Then by a simple substitution. □

LEMMA C.2 (Meet and join properties)**.**

Commutativity

- $\xi_1 \sqcap \xi_2 = \xi_2 \sqcap \xi_1$
- $\xi_1 \sqcup \xi_2 = \xi_2 \sqcup \xi_1$

Associativity

- $\xi_1 \sqcap (\xi_2 \sqcap \xi_3) = (\xi_1 \sqcap \xi_2) \sqcap \xi_3$
- $\xi_1 \sqcup (\xi_2 \sqcup \xi_3) = (\xi_1 \sqcup \xi_2) \sqcup \xi_3$

Completeness

If $\Delta \vdash \xi_1 \preceq \xi_2$ *then*

- $\xi_1 \sqcap \xi_2 = \xi_1$
- $\xi_1 \sqcup \xi_2 = \xi_2$

Soundness

If $\Delta \vdash \xi_1 : \mathsf{QUAL}$ *and* $\Delta \vdash \xi_2 : \mathsf{QUAL}$ *then*

- $\Delta \vdash \xi_1 \sqcap \xi_2 \preceq \xi_2$ *when* $\xi_1 \sqcap \xi_2$ *is defined*
- $\Delta \vdash \xi_1 \preceq \xi_1 \sqcup \xi_2$ *when* $\xi_1 \sqcup \xi_2$ *is defined*

Optimality

If $\Delta \vdash \xi_1 : \mathsf{QUAL}$ *and* $\Delta \vdash \xi_2 : \mathsf{QUAL}$ *then*

- *if* $\Delta \vdash \xi \preceq \xi_1$ *and* $\Delta \vdash \xi \preceq \xi_2$ *then* $\Delta \vdash \xi \preceq \xi_1 \sqcap \xi_2$ *whenever* $\xi_1 \sqcap \xi_2$ *is defined*
- *if* $\Delta \vdash \xi_1 \preceq \xi$ *and* $\Delta \vdash \xi_2 \preceq \xi$ *then* $\Delta \vdash \xi_1 \sqcup \xi_2 \preceq \xi$ *whenever* $\xi_1 \sqcup \xi_2$ *is defined*

Domain

The meet $\xi_1 \sqcap \xi_2$ is not defined if and only if one of:

- *ξ_1 is a variable and ξ_2 is* A *or* R*;*
- *ξ_2 is a variable and ξ_1 is* A *or* R*; or*
- *ξ_1 and ξ_2 are two distinct variables.*

Likewise for joins.

Substitution

Meet and join respect substitution:

- $\{\iota/\alpha\}(\xi_1 \sqcap \xi_2) = \{\iota/\alpha\}\xi_1 \sqcap \{\iota/\alpha\}\xi_2$ *when* $\xi_1 \sqcap \xi_2$ *is defined*
- $\{\iota/\alpha\}(\xi_1 \sqcup \xi_2) = \{\iota/\alpha\}\xi_1 \sqcup \{\iota/\alpha\}\xi_2$ *when* $\xi_1 \sqcup \xi_2$ *is defined*

Proof.

Commutativity By inspection.

Associativity By inspection. Any multi-meet containing only L s and one qualifier ξ (possibly repeated) is ξ; if it contains U at all or both A and R then it is U; otherwise it is undefined. Any multi-join containing only U s and one qualifier ξ (possibly repeated) is ξ; if it contains L at all or both A and R then it is L; otherwise it is undefined.

Completeness By induction on the qualifier subsumption derivation:

Case $\dfrac{\Delta \vdash \alpha : \mathsf{QUAL}}{\Delta \vdash \mathsf{U} \preceq \alpha}$**.**

Then $\mathsf{U} \sqcap \alpha = \mathsf{U}$ and $\mathsf{U} \sqcup \alpha = \alpha$.

Case $\dfrac{\mathfrak{q}_1 \preceq \mathfrak{q}_2}{\Delta \vdash \mathfrak{q}_1 \preceq \mathfrak{q}_2}$**.**

By cases on the derivation of $\mathfrak{q}_1 \preceq \mathfrak{q}_2$:

Case $\mathfrak{q} \preceq \mathfrak{q}$**.**

Then $\mathfrak{q} \sqcap \mathfrak{q} = \mathfrak{q} = \mathfrak{q} \sqcup \mathfrak{q}$.

Case $\mathsf{U} \preceq \mathfrak{q}$**.**

Then $\mathsf{U} \sqcap \mathfrak{q} = \mathsf{U}$ and $\mathsf{U} \sqcup \mathfrak{q} = \mathfrak{q}$.

Case $\mathfrak{q} \preceq \mathsf{L}$**.**

Then $\mathfrak{q} \sqcap \mathsf{L} = \mathfrak{q}$ and $\mathfrak{q} \sqcup \mathsf{L} = \mathsf{L}$.

Case $\dfrac{\Delta \vdash \alpha : \mathsf{QUAL}}{\Delta \vdash \alpha \preceq \mathsf{L}}$**.**

Then $\alpha \sqcap \mathsf{L} = \alpha$ and $\alpha \sqcup \mathsf{L} = \mathsf{L}$.

Case $\dfrac{\Delta \vdash \xi : \mathsf{QUAL}}{\Delta \vdash \xi \preceq \xi}$**.**

Then $\xi \sqcap \xi = \xi = \xi \sqcup \xi$.

Soundness

- We consider whether $\Delta \vdash \xi_1 \sqcap \xi_2 \preceq \xi_2$ by cases on ξ_1 and ξ_2:

$\Delta \vdash \xi_1 \sqcap \xi_2 \preceq \xi_2$		ξ_2				
		U	R	A	L	α
ξ_1	U	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{R}$	$\mathsf{U} \preceq \mathsf{A}$	$\mathsf{U} \preceq \mathsf{L}$	$\mathsf{U} \preceq \alpha$
	R	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{R} \preceq \mathsf{R}$	$\mathsf{U} \preceq \mathsf{A}$	$\mathsf{R} \preceq \mathsf{L}$	$\times$
	A	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{R}$	$\mathsf{A} \preceq \mathsf{A}$	$\mathsf{A} \preceq \mathsf{L}$	$\times$
	L	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{R} \preceq \mathsf{R}$	$\mathsf{A} \preceq \mathsf{A}$	$\mathsf{L} \preceq \mathsf{L}$	$\alpha \preceq \alpha$
	α	$\mathsf{U} \preceq \mathsf{U}$	$\times$	$\times$	$\alpha \preceq \mathsf{L}$	$\alpha \preceq \alpha$
	β	$\mathsf{U} \preceq \mathsf{U}$	$\times$	$\times$	$\beta \preceq \mathsf{L}$	$\times$

($\times$ indicates that $\xi_1 \sqcap \xi_2$ is undefined)

- We consider whether $\Delta \vdash \xi_1 \preceq \xi_1 \sqcup \xi_2$ by cases on ξ_1 and ξ_2:

$\Delta \vdash \xi_1 \preceq \xi_1 \sqcup \xi_2$		ξ_2				
		U	R	A	L	α
ξ_1	U	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{R}$	$\mathsf{U} \preceq \mathsf{A}$	$\mathsf{U} \preceq \mathsf{L}$	$\mathsf{U} \preceq \alpha$
	R	$\mathsf{R} \preceq \mathsf{R}$	$\mathsf{R} \preceq \mathsf{R}$	$\mathsf{R} \preceq \mathsf{L}$	$\mathsf{R} \preceq \mathsf{L}$	$\times$
	A	$\mathsf{A} \preceq \mathsf{A}$	$\mathsf{A} \preceq \mathsf{L}$	$\mathsf{A} \preceq \mathsf{A}$	$\mathsf{A} \preceq \mathsf{L}$	$\times$
	L	$\mathsf{L} \preceq \mathsf{L}$	$\mathsf{L} \preceq \mathsf{L}$	$\mathsf{L} \preceq \mathsf{L}$	$\mathsf{L} \preceq \mathsf{L}$	$\mathsf{L} \preceq \mathsf{L}$
	α	$\alpha \preceq \alpha$	$\times$	$\times$	$\alpha \preceq \mathsf{L}$	$\alpha \preceq \alpha$
	β	$\beta \preceq \beta$	$\times$	$\times$	$\beta \preceq \mathsf{L}$	$\times$

($\times$ indicates that $\xi_1 \sqcup \xi_2$ is undefined)

Optimality

Let $\Delta \vdash \xi_1 : \mathsf{QUAL}$ and $\Delta \vdash \xi_2 : \mathsf{QUAL}$. Then:

- Suppose that $\Delta \vdash \xi \preceq \xi_1$ and $\Delta \vdash \xi \preceq \xi_2$ and consider the possibilities by which $\xi_1 \sqcap \xi_2$ may be defined:

  Case $\mathsf{L} \sqcap \xi_2 = \xi_2$**.**

  By the assumption that $\Delta \vdash \xi \preceq \xi_2$.

  Case $\xi_1 \sqcap \mathsf{L} = \xi$**.**

  By symmetry.

  Case $\xi_1 \sqcap \xi_1 = \xi_1$**.**

  That is, $\xi_1 = \xi_2$. Then by the assumption that $\Delta \vdash \xi \preceq \xi_1$.

  Case $\mathsf{U} \sqcap \xi_2 = \mathsf{U}$**.**

  That is, $\xi_1 = \mathsf{U}$. Then $\Delta \vdash \xi \preceq \mathsf{U}$. By inspection of the rules for qualifier subsumption, this implies that $\xi = \mathsf{U}$, so by rule QSUB-BOT.

  Case $\xi_1 \sqcap \mathsf{U} = \mathsf{U}$**.**

  By symmetry.

  Case $\mathsf{A} \sqcap \mathsf{R} = \mathsf{U}$**.**

  By inspection of the rules for qualifier subsumption, $\Delta \vdash \xi \preceq \mathsf{A}$ only if ξ is U or A, and $\Delta \vdash \xi \preceq \mathsf{A}$ only if ξ is U or R. Then $\xi = \mathsf{U}$, so by rule QSUB-BOT.

  Case $\mathsf{R} \sqcap \mathsf{A} = \mathsf{U}$**.**

  By symmetry.

- By duality.

Domain

By inspection of the tables in the soundness case.

Substitution

By cases on the definition of meet:

Case $\mathsf{L} \sqcap \xi = \xi$**.**

Then $\{\iota/\alpha\}(\mathsf{L} \sqcap \xi) = \{\iota/\alpha\}\xi = \mathsf{L} \sqcap \{\iota/\alpha\}\xi = \{\iota/\alpha\}\mathsf{L} \sqcap \{\iota/\alpha\}\xi$.

Case $\xi \sqcap \mathsf{L} = \xi$**.**

By symmetry with the previous case.

Case $\xi \sqcap \xi = \xi$**.**

Then $\{\iota/\alpha\}(\xi \sqcap \xi) = \{\iota/\alpha\}\xi = \{\iota/\alpha\}\xi \sqcap \{\iota/\alpha\}\xi$.

Case $\mathsf{U} \sqcap \xi = \mathsf{U}$**.**

Then $\{\iota/\alpha\}(\mathsf{U} \sqcap \xi) = \{\iota/\alpha\}\mathsf{U} = \mathsf{U} = \mathsf{U} \sqcap \{\iota/\alpha\}\xi = \{\iota/\alpha\}\mathsf{U} \sqcap \{\iota/\alpha\}\xi$.

Case $\xi \sqcap \mathsf{U} = \mathsf{U}$**.**

By symmetry with the previous case.

Case $\mathsf{A} \sqcap \mathsf{R} = \mathsf{U}$**.**

Then $\{\iota/\alpha\}(\mathsf{A} \sqcap \mathsf{R}) = \{\iota/\alpha\}\mathsf{U} = \mathsf{U} = \mathsf{A} \sqcap \mathsf{R} = \{\iota/\alpha\}\mathsf{A} \sqcap \{\iota/\alpha\}\mathsf{R}$.

Case $\mathsf{R} \sqcap \mathsf{A} = \mathsf{U}$**.**

By symmetry with the previous case.

Dually for join. □

LEMMA C.3 (Lower bound of undefined meets)**.**

If $\xi_1 \sqcap \xi_2$ *is undefined,* $\Delta \vdash \xi \preceq \xi_1$*, and* $\Delta \vdash \xi \preceq \xi_2$*, then* $\xi = \mathsf{U}$.

Proof. If either ξ_1 or ξ_2 is U or L, or if $\xi_1 = \xi_2$, or if one is A and the other R, then the meet is defined. That leaves only two possibilities:

- One is A or R and the other is a variable α. The only possibilities for ξ to be less than α are if ξ is α or U. But since α is not less than A nor R, we know that $\xi = \mathsf{U}$.
- They are different variables α and β. As before, the only ways that $\Delta \vdash \xi \preceq \alpha$ is if ξ is either α or U. Similarly, for $\Delta \vdash \xi \preceq \beta$, ξ must be β or U. Since $\alpha \neq \beta$, we know that $\xi = \mathsf{U}$ □

Note that we do not simply define the meet in such cases to be U, because then meets would not be preserved by substitution.

LEMMA C.4 (Properties of bounds)**.**

1. *If* $\Delta \vdash \xi_1 \preceq \xi_2$ *then* $\Delta \vdash \xi_1 : \mathsf{QUAL}$ *and* $\Delta \vdash \xi_2 : \mathsf{QUAL}$.
2. *If* $\Delta \vdash \tau \preceq \xi$ *then* $\Delta \vdash \tau : \star$.

3. *If* $\Delta \vdash \tau : \star$ *then* $\Delta \vdash \tau \preceq \mathsf{L}$.

4. *If* $\Delta \vdash \Gamma \preceq \xi$ *and* $x{:}\tau \in \Gamma$ *then* $\Delta \vdash \tau \preceq \xi$.

5. *If* $\Delta \vdash \tau : \star$ *for all* τ *such that* $x{:}\tau \in \Gamma$, *then* $\Delta \vdash \Gamma \preceq \mathsf{L}$.

Proof.

1. By induction on the derivation.

2. By cases on the derivation and the previous part.

3. If τ is a type variable, then by rule B-VAR. Otherwise, by inversion of rules K-TYPE, QSUB-TOP, and B-TYPE.

4. By induction on the derivation.

5. By induction on Γ and the rules for context bounding. □

LEMMA 8.4 (Value strengthening, restated from p. 223)**.**

Any qualifier that upper bounds the type of a value also bounds the portion of the type context necessary for typing that value. That is, if $\Delta;\Gamma \vdash v : \tau$ *and* $\Delta \vdash \tau \preceq \xi$ *then there exist some* Γ_1 *and* Γ_2 *such that*

- $\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$,
- $\Delta;\Gamma_1 \vdash v : \tau$,
- $\Delta \vdash \Gamma_1 \preceq \xi$, *and*
- $\Delta \vdash \Gamma_2 \preceq \mathsf{A}$.

Proof. By induction on the derivation of $\Delta;\Gamma \vdash v : \tau$:

Case $\dfrac{\Delta \vdash \tau : \star}{\Delta;\bullet,x{:}\tau \vdash x : \tau}$.

Let $\Gamma_1 = \bullet,x{:}\tau$ and $\Gamma_2 = \bullet$.

Case $\dfrac{\Delta \vdash \xi : \mathsf{QUAL} \qquad \Delta \vdash \Gamma \preceq \xi \qquad \Delta;\Gamma,x{:}\tau_1 \vdash e : \tau_2}{\Delta;\Gamma \vdash \lambda x.e : {}^{\xi}(\tau_1 \multimap \tau_2)}$.

Let $\Gamma_1 = \Gamma$ and $\Gamma_2 = \bullet$.

Case $\dfrac{\Delta \vdash \xi : \mathsf{QUAL} \qquad \Delta \vdash \Gamma \preceq \xi \qquad \Delta, \alpha{:}\kappa; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda.e :^{\xi} \forall \alpha{:}\kappa.\tau}$.

Let $\Gamma_1 = \Gamma$ and $\Gamma_2 = \bullet$.

Case $\dfrac{\Delta \vdash \tau_1 \preceq \xi \qquad \Delta \vdash \tau_2 : \star \qquad \Delta; \Gamma \vdash v_1 : \tau_1}{\Delta; \Gamma \vdash \mathsf{inl}\, v_1 :^{\xi} (\tau_1 \oplus \tau_2)}$.

By the induction hypothesis, there exist some Γ_1 and Γ_2 such that

(1) $\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$,

(2) $\Delta; \Gamma_1 \vdash v_1 : \tau_1$,

(3) $\Delta \vdash \Gamma_2 \preceq \mathsf{A}$, and

(4) $\Delta \vdash \Gamma_1 \preceq \xi$.

Then,

(5) $\Delta; \Gamma_1 \vdash \mathsf{inl}\, v_1 :^{\xi} (\tau_1 \oplus \tau_2)$ by rule T-INL .

Case $\dfrac{\Delta \vdash \tau_2 \preceq \xi \qquad \Delta \vdash \tau_1 : \star \qquad \Delta; \Gamma \vdash v_2 : \tau_2}{\Delta; \Gamma \vdash \mathsf{inr}\, v_2 :^{\xi} (\tau_1 \oplus \tau_2)}$.

By symmetry with the previous case.

Case $\dfrac{\begin{array}{c}\Delta \vdash \xi' : \mathsf{QUAL} \qquad \Delta; \Gamma \vdash v_1 :^{\xi_1} (\tau_1 \multimap \tau) \\ \Delta \vdash \xi_1 \preceq \xi \qquad \Delta; \Gamma \vdash v_2 :^{\xi_2} (\tau_2 \multimap \tau) \qquad \Delta \vdash \xi_2 \preceq \xi\end{array}}{\Delta; \Gamma \vdash [v_1, v_2] :^{\xi} (^{\xi'}(\tau_1 \oplus \tau_2) \multimap \tau)}$.

By the induction hypothesis, there exist some Γ_{11} and Γ_{21} such that

(1) $\Delta \vdash \Gamma \rightsquigarrow \Gamma_{11} \boxplus \Gamma_{21}$,

(2) $\Delta; \Gamma_{12} \vdash v_1 :^{\xi_1} (\tau_1 \multimap \tau)$,

(3) $\Delta \vdash \Gamma_{21} \preceq \mathsf{A}$, and

(4) $\Delta \vdash \Gamma_{11} \preceq \xi_1$.

Likewise, by the induction hypothesis, there exist some Γ_{12} and Γ_{22} such that

(5) $\Delta \vdash \Gamma \rightsquigarrow \Gamma_{12} \boxplus \Gamma_{22}$,

(6) $\Delta; \Gamma_{13} \vdash v_2 : {}^{\xi_2}(\tau_2 \multimap \tau)$,

(7) $\Delta \vdash \Gamma_{22} \preceq \mathsf{A}$, and

(8) $\Delta \vdash \Gamma_{12} \preceq \xi_2$.

Then let $\Gamma_1 = \Gamma_{11} \cup \Gamma_{12}$ and let $\Gamma_2 = \Gamma_{21} \cap \Gamma_{22}$. Note that because each pair is split from the same Γ, they agree everywhere that they are defined. Furthermore, note that if ξ_1 upper bounds the qualifiers of the codomain of Γ_{11} and ξ_2 upper bounds the qualifiers of the codomain of Γ_{12}, then ξ upper bounds the qualifiers of the codomain of Γ_1:

(9) $\Delta \vdash \Gamma_1 \preceq \xi$.

Furthermore,

(10) $\Delta; \Gamma_1 \vdash v_1 : {}^{\xi_1}(\tau_1 \multimap \tau)$ by weak.

(11) $\Delta; \Gamma_1 \vdash v_2 : {}^{\xi_2}(\tau_2 \multimap \tau)$ by weak.

(12) $\Delta; \Gamma_1 \vdash [v_1, v_2] : {}^{\xi}({}^{\xi'}(\tau_1 \oplus \tau_2) \multimap \tau)$ by rule T-SUME.

Case $\dfrac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Delta; \Gamma_1 \vdash v_1 : \tau_1 \qquad \Delta \vdash \tau_1 \preceq \xi \qquad \Delta; \Gamma_2 \vdash v_2 : \tau_2 \qquad \Delta \vdash \tau_2 \preceq \xi}{\Delta; \Gamma \vdash \langle v_1, v_2 \rangle : {}^{\xi}(\tau_1 \otimes \tau_2)}$.

By the induction hypothesis, there exist some Γ_{11} and Γ_{21} such that

(1) $\Delta \vdash \Gamma_1 \rightsquigarrow \Gamma_{11} \boxplus \Gamma_{12}$,

(2) $\Delta; \Gamma_{11} \vdash v_1 : \tau_1$,

(3) $\Delta \vdash \Gamma_{12} \preceq \mathsf{A}$, and

(4) $\Delta \vdash \Gamma_{11} \preceq \xi$.

Likewise, by the induction hypothesis, there exist some Γ_{21} and Γ_{21} such that

(5) $\Delta \vdash \Gamma_2 \rightsquigarrow \Gamma_{21} \boxplus \Gamma_{22}$,

(6) $\Delta; \Gamma_{21} \vdash v_2 : \tau_2$,

(7) $\Delta \vdash \Gamma_{22} \preceq \mathsf{A}$, and

(8) $\Delta \vdash \Gamma_{21} \preceq \xi$.

Then let $\Gamma_1 = \Gamma_{11} \cup \Gamma_{21}$ and let $\Gamma_2 = \Gamma_{12} \cup \Gamma_{22}$. Note that because each pair is split from the same Γ, they agree everywhere that they are defined. Note also that Γ can be split as

(9) $\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$.

Furthermore, note that because ξ upper bounds the qualifiers of the codomains of both Γ_{11} and Γ_{21}, it also upper bounds the qualifiers of the codomain of Γ_1:

(10) $\Delta \vdash \Gamma_1 \preceq \xi$.

Furthermore,

(11) $\Delta; \Gamma_1 \vdash v_1 : \tau_1$ by weak.

(12) $\Delta; \Gamma_1 \vdash v_2 : \tau_2$ by weak.

(13) $\Delta; \Gamma_1 \vdash \langle v_1, v_2 \rangle : {}^{\xi}(\tau_1 \otimes \tau_2)$ by rule T-PROD.

Case $\dfrac{\Delta \vdash \xi' : \mathsf{QUAL} \qquad \Delta; \Gamma \vdash v : {}^{\xi}(\tau_1 \multimap {}^{\xi}(\tau_2 \multimap \tau))}{\Delta; \Gamma \vdash \mathsf{uncurry}\ v : {}^{\xi}({}^{\xi'}(\tau_1 \otimes \tau_2) \multimap \tau)}$.

As in the $\mathsf{inl}\ v$ case.

Case $\dfrac{\Delta \vdash \xi : \mathsf{QUAL}}{\Delta; \bullet \vdash \langle\rangle : {}^{\xi} 1}$.

Let $\Gamma_1 = \Gamma_2 = \bullet$.

Case $\dfrac{\Delta \vdash \xi : \mathsf{QUAL} \qquad \Delta \vdash \tau : \star \qquad \Delta;\Gamma \vdash v :^{\xi'} 1}{\Delta;\Gamma \vdash \mathsf{ignore}\ v :^{\xi} (\tau \multimap \tau)}$.

As in the $\mathsf{inl}\ v$ case.

Case $\dfrac{\Delta \vdash \Gamma \rightsquigarrow \Gamma'_1 \boxplus \Gamma'_2 \qquad \Delta;\Gamma'_1 \vdash v : \tau \qquad \Delta \vdash \Gamma'_2 \preceq \mathsf{A}}{\Delta;\Gamma \vdash v : \tau}$.

By the induction hypothesis, there exist some Γ_{11} and Γ_{12} such that

(1) $\Delta \vdash \Gamma'_1 \rightsquigarrow \Gamma_{11} \boxplus \Gamma_{12}$,

(2) $\Delta;\Gamma_{11} \vdash v : \tau$,

(3) $\Delta \vdash \Gamma_{12} \preceq \mathsf{A}$, and

(4) $\Delta \vdash \Gamma_{11} \preceq \xi$.

Then let $\Gamma_1 = \Gamma_{11}$ and let $\Gamma_2 = \Gamma'_2 \cup \Gamma_{12}$. Note that because $\Delta \vdash \Gamma'_2 \preceq \mathsf{A}$ and $\Delta \vdash \Gamma_{12} \preceq \mathsf{A}$, we know that $\Delta \vdash \Gamma_2 \preceq \mathsf{A}$ as well. □

LEMMA 8.3 (Dereliction, restated from p. 223)**.**

If $\Delta;\Gamma \vdash v :^{\xi} (\tau_1 \multimap \tau_2)$ *and* $\Delta \vdash \xi \preceq \xi'$ *then* $\Delta;\Gamma \vdash \lambda x.\, v\, x :^{\xi'} (\tau_1 \multimap \tau_2)$.

Proof. By lemma 8.4, there exist some Γ_1 and Γ_2 such that:

(1) $\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$,

(2) $\Delta;\Gamma_1 \vdash v :^{\xi} (\tau_1 \multimap \tau_2)$,

(3) $\Delta \vdash \Gamma_2 \preceq \mathsf{A}$, and

(4) $\Delta \vdash \Gamma_1 \preceq \xi$.

Then:

(5) $\Delta;\bullet, x{:}\tau_1 \vdash x : \tau_1$ by rule T-VAR

(6) $\Delta;\Gamma_1, x{:}\tau_1 \vdash v\, x : \tau_2$ by rule T-APP

(7) $\Delta \vdash \Gamma_1 \preceq \xi'$ by ind. Γ_1, trans.

(8) $\Delta;\Gamma_1 \vdash \lambda x.\, v\, x :^{\xi'} (\tau_1 \multimap \tau_2)$ by rule T-ABS. □

C.2 Properties of $\lambda^{\mathsf{URAL}}(\mathcal{C})$ and the Translation

In this section, I prove several propositions about $\lambda^{\mathsf{URAL}}(\mathcal{C})$ and the CCoS translation, including those from §8.4.

Lemma C.5 ($\lambda^{\mathsf{URAL}}(\mathcal{C})$ Regularity)**.**

1. *If* $\mathrm{D} \vdash_{\mathcal{C}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_1$ *and* $x{:}t' \in \mathrm{G}$, G_1, *or* G_2, *then* $\mathrm{D} \vdash_{\mathcal{C}} t' : \star$.
2. *If* $\mathrm{D}; \mathrm{G} \vdash_{\mathcal{C}} e : t \,;\, c$ *and* $x{:}t' \in \mathrm{G}$ *then* $\mathrm{D} \vdash_{\mathcal{C}} t' : \star$
3. *If* $\mathrm{D}; \mathrm{G} \vdash_{\mathcal{C}} e : t \,;\, c$ *then* $\mathrm{D} \vdash_{\mathcal{C}} t : \star$
4. *If* $\mathrm{D}; \mathrm{G} \vdash_{\mathcal{C}} e : t \,;\, c$ *then* $\mathrm{D} \vdash_{\mathcal{C}} c : \mathsf{CTL}$

Proof.

1. By induction on the derivation.
2. By induction on the derivation, using the previous part.
3. By induction on the derivation, using the previous part.
4. By induction on the derivation, considering that "derivations" with malformed effect sequences are not valid derivations. □

Lemma 8.5 (Translation of kinding, restated from p. 227)**.**

For all D, *i, and k, if* $\mathrm{D} \vdash_{\mathcal{C}} i : k$ *then* $\mathrm{D}^* \vdash i^* : k^*$.

Proof. By simple induction on the kinding derivation:

Case $\dfrac{\alpha{:}k \in \mathrm{D}}{\mathrm{D} \vdash_{\mathcal{C}} \alpha : k}$**.**

Then

(1) $\alpha{:}k^* \in \mathrm{D}^*$ by def. D^* and

(2) $\alpha^* = \alpha$,

and thus $\mathrm{D}^* \vdash \alpha^* : k^*$ by rule K-Var.

Case $\mathrm{D} \vdash_{\mathcal{C}} \mathfrak{q} : \mathsf{QUAL}$**.**

By rule K-QUAL.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{C}} \overline{t} : \overline{\star} \qquad \mathrm{D} \vdash_{\mathcal{C}} \xi : \mathsf{QUAL}}{\mathrm{D} \vdash_{\mathcal{C}} {}^{\xi}\overline{t} : \star}$**.**

(1) $\mathrm{D}^* \vdash \overline{t}^* : \overline{\star}$ by IH,

(2) $\mathrm{D}^* \vdash \xi^* : \mathsf{QUAL}$ by IH,

(3) $({}^{\xi}\overline{t})^* = {}^{\xi}(\overline{t}^*)$ by def. t^*, and

(4) $\xi^* = \xi$ by def. ξ^*.

Thus, by rule K-TYPE.

Case $\mathrm{D} \vdash_{\mathcal{C}} 1 : \overline{\star}$**.**

By rule K-UNIT.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{C}} t_1 : \star \qquad \mathrm{D} \vdash_{\mathcal{C}} t_2 : \star}{\mathrm{D} \vdash_{\mathcal{C}} t_1 \otimes t_2 : \overline{\star}}$**.**

(1) $\mathrm{D}^* \vdash {t_1}^* : \star$ by IH and

(2) $\mathrm{D}^* \vdash {t_2}^* : \star$ by IH.

Thus by rule K-PROD.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{C}} t_1 : \star \qquad \mathrm{D} \vdash_{\mathcal{C}} t_2 : \star}{\mathrm{D} \vdash_{\mathcal{C}} t_1 \oplus t_2 : \overline{\star}}$**.**

Likewise.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{C}} t_1 : \star \qquad \mathrm{D} \vdash_{\mathcal{C}} t_2 : \star \qquad \mathrm{D} \vdash_{\mathcal{C}} c : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{C}} t_1 \overset{c}{\multimap} t_2 : \overline{\star}}$**.**

(1) $\mathrm{D}^*, \alpha{:}\star \vdash {t_1}^* : \star$ by IH and weak.

(2) $\mathrm{D}^*, \alpha{:}\star \vdash {t_2}^* : \star$ by IH and weak.

(3) $\mathrm{D}^*, \alpha{:}\star \vdash c^* : \mathsf{QUAL}$ by IH and weak.

(4) $\mathrm{D}^*, \alpha{:}\star \vdash \mathsf{L} : \mathsf{QUAL}$ by rule K-QUAL

(5) $\mathrm{D}^*, \alpha{:}\star \vdash \alpha : \star$ by rule K-VAR

(6) $\mathrm{D}^*, \alpha{:}\star \vdash \langle\!\langle \alpha, c \rangle\!\rangle^-_{\mathcal{C}} : \star$ by (5), property 8.4.2

(7) $\mathrm{D}^*, \alpha{:}\star \vdash t_2{}^* \multimap \langle\!\langle \alpha, c \rangle\!\rangle^-_{\mathcal{C}} : \star$ by (2, 6), rule K-ARR

(8) $\mathrm{D}^*, \alpha{:}\star \vdash {}^{c^*}(t_2{}^* \multimap \langle\!\langle \alpha, c \rangle\!\rangle^-_{\mathcal{C}}) : \star$ by (3, 7), rule K-TYPE

(9) $\mathrm{D}^*, \alpha{:}\star \vdash \langle\!\langle \alpha, c \rangle\!\rangle^+_{\mathcal{C}} : \star$ by (5), property 8.4.2

(10) $\mathrm{D}^*, \alpha{:}\star \vdash {}^{c^*}(t_2{}^* \multimap \langle\!\langle \alpha, c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c \rangle\!\rangle^+_{\mathcal{C}} : \overline{\star}$ by (8–9), rule K-ARR

(11) $\mathrm{D}^*, \alpha{:}\star \vdash {}^{\mathsf{L}}({}^{c^*}(t_2{}^* \multimap \langle\!\langle \alpha, c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c \rangle\!\rangle^+_{\mathcal{C}}) : \star$
by (4, 10), rule K-TYPE

(12) $\mathrm{D}^*, \alpha{:}\star \vdash t_1{}^* \multimap {}^{\mathsf{L}}({}^{c^*}(t_2{}^* \multimap \langle\!\langle \alpha, c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c \rangle\!\rangle^+_{\mathcal{C}}) : \overline{\star}$
by (1, 11), rule K-ARR

(13) $\mathrm{D}^*, \alpha{:}\star \vdash {}^{\mathsf{L}}(t_1{}^* \multimap {}^{\mathsf{L}}({}^{c^*}(t_2{}^* \multimap \langle\!\langle \alpha, c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c \rangle\!\rangle^+_{\mathcal{C}})) : \star$
by (4, 12), rule K-TYPE

(14) $\mathrm{D}^* \vdash \forall \alpha{:}\star . {}^{\mathsf{L}}(t_1{}^* \multimap {}^{\mathsf{L}}({}^{c^*}(t_2{}^* \multimap \langle\!\langle \alpha, c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c \rangle\!\rangle^+_{\mathcal{C}})) : \overline{\star}$
by (13), rule K-ALL

(15) $\mathrm{D}^* \vdash (t_1 \overset{c}{\multimap} t_2)^* : \overline{\star}$ by (14), def. $\overline{t}^*$.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{C}} t : \star}{\mathrm{D} \vdash_{\mathcal{C}} \mathsf{ref}\ t : \overline{\star}}$.

As in product and sum cases.

Case $\dfrac{\mathrm{D}, \beta{:}k \vdash_{\mathcal{C}} t : \star \qquad \mathrm{D} \vdash_{\mathcal{C}} c : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{C}} \forall^c \beta{:}k . t : \overline{\star}}$.

(1) $\mathrm{D}, \alpha{:}\star, \beta{:}k \vdash_{\mathcal{C}} c : \mathsf{CTL}$ by prem., weak.

(2) $\mathrm{D}^*, \alpha{:}\star, \beta{:}k^* \vdash \alpha : \star$ by rule K-VAR

(3) $\mathrm{D}^*, \alpha{:}\star, \beta{:}k^* \vdash t^* : \star$ by IH, weak.

(4) $\mathrm{D}^*, \alpha{:}\star, \beta{:}k^* \vdash \langle\!\langle \alpha, c \rangle\!\rangle^-_{\mathcal{C}} : \star$ by (1–2), property 8.4.2

(5) $\mathrm{D}^*, \alpha{:}\star, \beta{:}k^* \vdash t^* \multimap \langle\!\langle \alpha, c \rangle\!\rangle^-_{\mathcal{C}} : \overline{\star}$ by (3–4), rule K-ARR

(6) $\mathrm{D}^*, \alpha{:}\star, \beta{:}k^* \vdash c^* : \mathsf{QUAL}$ by IH, weak.

(7) $\mathrm{D}^*, \alpha{:}\star, \beta{:}k^* \vdash {}^{c^*}(t^* \multimap \langle\!\langle \alpha, c \rangle\!\rangle^-_{\mathcal{C}}) : \star$ by (5–6), rule K-TYPE

(8) $\mathrm{D}^*, \alpha{:}\star, \beta{:}k^* \vdash \langle\!\langle \alpha, c \rangle\!\rangle^+_{\mathcal{C}} : \star$ by (1–2), property 8.4.2

(9) $\mathrm{D}^*, \alpha{:}\star, \beta{:}k^* \vdash {}^{c^*}(t^* \multimap \langle\!\langle \alpha, c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c \rangle\!\rangle^+_{\mathcal{C}} : \overline{\star}$ by (7–8), rule K-ARR

(10) $\mathrm{D}^*, \alpha{:}\star, \beta{:}k^* \vdash {}^{\mathsf{L}}({}^{c^*}(t^* \multimap \langle\!\langle \alpha, c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c \rangle\!\rangle^+_{\mathcal{C}}) : \star$ by (9), rule K-TYPE

(11) $\mathrm{D}^*, \alpha{:}\star \vdash \forall \beta{:}k^*.\, {}^{\mathsf{L}}({}^{c^*}(t^* \multimap \langle\!\langle \alpha, c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c \rangle\!\rangle^+_{\mathcal{C}}) : \star$ by (10), rule K-ALL

(12) $\mathrm{D}^*, \alpha{:}\star \vdash {}^{\mathsf{L}}\forall \beta{:}k^*.\, {}^{\mathsf{L}}({}^{c^*}(t^* \multimap \langle\!\langle \alpha, c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c \rangle\!\rangle^+_{\mathcal{C}}) : \star$ by (11), rule K-TYPE

(13) $\mathrm{D}^* \vdash \forall \alpha{:}\star.\, {}^{\mathsf{L}}\forall \beta{:}k^*.\, {}^{\mathsf{L}}({}^{c^*}(t^* \multimap \langle\!\langle \alpha, c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c \rangle\!\rangle^+_{\mathcal{C}}) : \overline{\star}$ by (12), rule K-ALL

(14) $\mathrm{D}^* \vdash (\forall\, {}^{c}\beta{:}k.\, t)^* : \overline{\star}$ by (13), def. $\overline{t}^*$. □

LEMMA C.6 (Translation of qualifier judgments)**.**

1. *If* $\mathrm{D} \vdash_{\mathcal{C}} \xi_1 \preceq \xi_2$ *then* $\mathrm{D}^* \vdash \xi_1 \preceq \xi_2$.
2. *If* $\mathrm{D} \vdash_{\mathcal{C}} t \preceq \xi$ *then* $\mathrm{D}^* \vdash t^* \preceq \xi$.
3. *If* $\mathrm{D} \vdash_{\mathcal{C}} \mathrm{G} \preceq \xi$ *then* $\mathrm{D}^* \vdash \mathrm{G}^* \preceq \xi$.
4. *If* $\mathrm{D} \vdash_{\mathcal{C}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_2$ *then* $\mathrm{D}^* \vdash \mathrm{G}^* \rightsquigarrow {\mathrm{G}_1}^* \boxplus {\mathrm{G}_2}^*$

Proof. By simple induction on each derivation. □

LEMMA 8.6 (Translation of effect bounds, restated from p. 228)**.**

If $\mathrm{D} \vdash_{\mathcal{C}} c \succeq \xi$ *then* $\mathrm{D}^* \vdash \xi \preceq c^*$.

Proof. By cases on the derivation:

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{C}} \xi : \mathsf{QUAL}}{\mathrm{D} \vdash_{\mathcal{C}} \bot_{\mathcal{D}} \succeq \xi}$**.**

Then ${\bot_{\mathcal{C}}}^* = \mathsf{L}$, so by rule QSUB-TOP.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{C}} c : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{C}} c \succeq \mathsf{U}}$**.**

By rule QSUB-BOT.

Additional cases must be proved for new rules added by specific control effect instances. □

LEMMA 8.7 (Translation of effect subsumption, restated from p. 228)**.**

If $\mathrm{D} \vdash_{\mathcal{C}} c_1 \preceq c_2$ *then* $\mathrm{D}^* \vdash {c_2}^* \preceq {c_1}^*$.

Proof. By induction on the derivation:

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{C}} c : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{C}} c \preceq c}$**.**

By rule QSUB-REFL.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{C}} c_1 \preceq c' \qquad \mathrm{D} \vdash_{\mathcal{C}} c' \preceq c_2}{\mathrm{D} \vdash_{\mathcal{C}} c_1 \preceq c_2}$**.**

By the induction hypothesis twice and lemma C.1.

Additional cases must be proved for new rules added by specific control effect instances. □

LEMMA 8.8 (Translation of term typing, restated from p. 228)**.**

If $\mathrm{D}; \mathrm{G} \vdash_{\mathcal{C}} e : t \,; c$ *then*

$$\mathrm{D}^*; \mathrm{G}^* \vdash [\![e]\!]_{\mathcal{C}} : {}^{\mathsf{L}}({c^*}(t^* \multimap \langle\!\langle t^*, c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle t^*, c \rangle\!\rangle^+_{\mathcal{C}}).$$

Proof. We generalize the lemma to the following induction hypothesis:

> If $\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} e : t ; c$, then for all τ_0 such that $\mathrm{D}^* \vdash \tau_0 : \star$, and for all ξ_0 such that $\mathrm{D}^* \vdash \xi_0 \preceq c^*$, it is the case that $\mathrm{D}^*;\mathrm{G}^* \vdash [\![e]\!]_{\mathcal{C}} : {}^{\mathsf{L}}(^{\xi_0}(t^* \multimap \langle\!\langle \tau_0, c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau_0, c \rangle\!\rangle^+_{\mathcal{C}})$

We use lexical induction on the pair of: 1) the size of e, using size defined as follows, and 2) the height of the typing derivation for $\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} e : t ; c$. The size of an expression is given by:

$$
\begin{aligned}
|x| &= 1 & |\lambda x.\, e'| &= 1 + |e'| \\
|\Lambda.\, e'| &= 1 + |e'| & |\mathsf{inl}\ v| &= 1 + |v| \\
|\mathsf{inr}\ v| &= 1 + |v| & |[v_1, v_2]| &= |v_1| + |v_2| \\
|\langle v_1, v_2 \rangle| &= |v_1| + |v_2| & |\mathsf{uncurry}\ v| &= \mathbf{3} + |v| \\
|\langle\rangle| &= 1 & |\mathsf{ignore}\ v| &= 1 + |v| \\
|e_1\, e_2| &= |e_1| + |e_2| & |e'\ \_| &= 1 + |e'| \\
|\mathsf{new}^{\mathsf{q}}\, e'| &= 1 + |e'| & |\mathsf{delete}\ e'| &= 1 + |e'| \\
|\mathsf{read}\ e'| &= 1 + |e'| & |\mathsf{swap}\ e_1\, e_2| &= |e_1| + |e_2| \\
|e_1\ \mathsf{handle}\ \psi \to e_2| &= |e_1| + |e_2| & |\mathsf{raise}\ \psi| &= 1 \\
|\mathsf{reset}\ e| &= 1 + |e| & |\mathsf{shift}\, x\, \mathsf{in}\, e| &= 1 + |e|
\end{aligned}
$$

In particular, this means that we can apply the induction hypothesis to any expression smaller than e, or to the same expression e provided that we use a subderivation of the derivation at hand. We proceed by cases on the conclusion of the typing derivation. We start with the cases that apply to values, since those have much in common:

Case v.

> By inspection, note that in all rules for typing values, the effect is pure, and thus:
>
> (1) $c = \bot_{\mathcal{C}}$
>
> Note further that by property 8.4.1,

(2) $\langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}} = \langle\!\langle \tau', c \rangle\!\rangle^+_{\mathcal{C}} = \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}$.

Furthermore, by the definition of $[\![v]\!]_{\mathcal{C}}$, we know that

(3) $[\![e]\!]_{\mathcal{C}} = \lambda y.\, y\, v^*$.

Thus, it is sufficient to show that $\mathrm{D}^*;\mathrm{G}^* \vdash \lambda y.\, y\, v^* : {}^{\mathsf{L}}(^{\xi_0}(t^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}})$ (where y is fresh for v). Suppose that (4) $\mathrm{D}^*;\mathrm{G}^* \vdash v^* : t^*$. Then:

(5) $\mathrm{D}^* \vdash \mathsf{L} : \mathsf{QUAL}$ — by rule K-QUAL

(6) $\mathrm{D} \vdash_{\mathcal{C}} t : \star$ — by lemma C.5

(7) $\mathrm{D}^* \vdash t^* : \star$ — by lemma 8.5

(8) $\mathrm{D}^* \vdash {}^{\xi_0}(t^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) : \star$ — by (5, 7), property 8.4.2

(9) $\mathrm{D}^*; \bullet, y{:}^{\xi_0}(t^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \vdash y : {}^{\xi_0}(t^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}})$ — by (8)

(10) $\mathrm{D}^* \vdash \mathrm{G}^* \rightsquigarrow \bullet \boxplus \mathrm{G}^*$ — by ind. G^*, rule S-CONSR

(11) $\mathrm{D}^* \vdash \mathrm{G}^*, y{:}^{\xi_0}(t^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \rightsquigarrow \bullet, y{:}^{\xi_0}(t^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \boxplus \mathrm{G}^*$ — by (10), rule S-CONSL

(12) $\mathrm{D}^*;\mathrm{G}^*, y{:}^{\xi_0}(t^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \vdash y\, v^* : \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}$ — by (4, 9, 11)

(13) $\mathrm{D}^*;\mathrm{G}^* \vdash \lambda y.\, y\, v^* : {}^{\mathsf{L}}(^{\xi_0}(t^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}})$ — by (5, 12).

Therefore, it is sufficient to show (4) $\mathrm{D}^*;\mathrm{G}^* \vdash v^* : t^*$. We proceed by a nested induction on the structure of v, considering the possible typing derivations:

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{C}} t : \star}{\mathrm{D}; \bullet, x{:}t \vdash_{\mathcal{C}} x : t ; \bot_{\mathcal{C}}}$.

By rule T-VAR.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{C}} \xi : \mathsf{QUAL} \qquad \mathrm{D} \vdash_{\mathcal{C}} \mathrm{G} \preceq \xi \qquad \mathrm{D}; \mathrm{G}, x{:}t_1 \vdash_{\mathcal{C}} e : t_2 ; c'}{\mathrm{D}; \mathrm{G} \vdash_{\mathcal{C}} \lambda x.\, e : {}^{\xi}(t_1 \xrightarrow{c'}\!\!\circ\; t_2) ; \bot_{\mathcal{C}}}$.

We want to show that $\mathrm{D}^*;\mathrm{G}^* \vdash (\lambda x.\, e)^* : {}^{\xi}((t_1 \xrightarrow{c'}\!\!\circ\; t_2)^*)$. Note that

(1) $(t_1 \xrightarrow{c'}\!\!\circ\, t_2)^* = \forall\alpha{:}\star.\,{}^{\mathsf{L}}(t_1{}^* \multimap {}^{\mathsf{L}}({}^{c'*}(t_2{}^* \multimap \langle\!\langle \alpha, c'\rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c'\rangle\!\rangle^+_{\mathcal{C}}))$ by def. $\overline{t}^*$ and

(2) $(\lambda x.e)^* = \Lambda.\lambda x.[\![e]\!]_{\mathcal{C}}$ by def. v^*.

Then

(3) $\mathrm{D}, \alpha{:}\star; \mathrm{G}, x{:}t_1 \vdash_{\mathcal{C}} e : t_2 ; c'$ by weak.

(4) $\mathrm{D}^*, \alpha{:}\star \vdash \alpha : \star$ by rule K-VAR

(5) $\mathrm{D}^*, \alpha{:}\star; \mathrm{G}^*, x{:}t_1{}^* \vdash [\![e]\!]_{\mathcal{C}} : {}^{\mathsf{L}}({}^{c'*}(t_2{}^* \multimap \langle\!\langle \alpha, c'\rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c'\rangle\!\rangle^+_{\mathcal{C}})$ by IH, (3–4)

(6) $\mathrm{D}^*, \alpha{:}\star \vdash \mathrm{G}^* \preceq \mathsf{L}$ by lemma C.4

(7) $\mathrm{D}^*, \alpha{:}\star; \mathrm{G}^* \vdash \lambda x.[\![e]\!]_{\mathcal{C}} : {}^{\mathsf{L}}(t_1{}^* \multimap {}^{\mathsf{L}}({}^{c'*}(t_2{}^* \multimap \langle\!\langle \alpha, c'\rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c'\rangle\!\rangle^+_{\mathcal{C}}))$ by (5–6)

(8) $\mathrm{D}^* \vdash \mathrm{G}^* \preceq \xi$ by lemma C.6

(9) $\mathrm{D}^*; \mathrm{G}^* \vdash \Lambda.\lambda x.[\![e]\!]_{\mathcal{C}} : {}^{\xi}\forall\alpha{:}\star.\,{}^{\mathsf{L}}(t_1{}^* \multimap {}^{\mathsf{L}}({}^{c'*}(t_2{}^* \multimap \langle\!\langle \alpha, c'\rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c'\rangle\!\rangle^+_{\mathcal{C}}))$ by (7–8)

(10) $\mathrm{D}^*; \mathrm{G}^* \vdash (\lambda x.e)^* : {}^{\xi}((t_1 \xrightarrow{c'}\!\!\circ\, t_2)^*)$ by (1–2, 9).

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{C}} \xi : \mathsf{QUAL} \qquad \mathrm{D} \vdash_{\mathcal{C}} \mathrm{G} \preceq \xi \qquad \mathrm{D}, \beta{:}k; \mathrm{G} \vdash_{\mathcal{C}} e : t ; c'}{\mathrm{D}; \mathrm{G} \vdash_{\mathcal{C}} \Lambda.e : {}^{\xi}\forall\,{}^{c'}\beta{:}k.\,t ; \bot_{\mathcal{C}}}$.

We want to show that $\mathrm{D}^*; \mathrm{G}^* \vdash (\Lambda.e)^* : {}^{\xi}((\forall\,{}^{c'}\beta{:}k.\,t)^*)$. Note that

(1) $(\forall\,{}^{c'}\beta{:}k.\,t)^* = \forall\alpha{:}\star.\,{}^{\mathsf{L}}\forall\beta{:}k^*.\,{}^{\mathsf{L}}({}^{c'*}(t^* \multimap \langle\!\langle \alpha, c'\rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c'\rangle\!\rangle^+_{\mathcal{C}})$ by def. $\overline{t}^*$ and

(2) $(\Lambda.e)^* = \Lambda.\Lambda.[\![e]\!]_{\mathcal{C}}$ by def. v^*.

Then

(3) $\mathrm{D}, \beta{:}k, \alpha{:}\star; \mathrm{G} \vdash_{\mathcal{C}} e : t ; c'$ by weak.

(4) $\mathrm{D}^*, \beta{:}k^*, \alpha{:}\star \vdash \alpha : \star$ by rule K-VAR

(5) $\mathrm{D}^*, \beta{:}k^*, \alpha{:}\star; \mathrm{G}^* \vdash [\![e]\!]_{\mathcal{C}} : {}^{\mathsf{L}}({}^{c'*}(t^* \multimap \langle\!\langle \alpha, c'\rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c'\rangle\!\rangle^+_{\mathcal{C}})$ by IH, (3–4)

(6) $\mathrm{D}^*, \beta{:}k^*, \alpha{:}\star \vdash \mathrm{G}^* \preceq \mathsf{L}$ by lemma C.4

(7) $\mathrm{D}^*, \alpha{:}\star; \mathrm{G}^* \vdash \Lambda.[\![e]\!]_{\mathcal{C}} : {}^{\mathsf{L}}\forall\beta{:}k^*.\,{}^{\mathsf{L}}({}^{c'*}(t^* \multimap \langle\!\langle \alpha, c'\rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c'\rangle\!\rangle^+_{\mathcal{C}})$ by (5–6)

(8) $\mathrm{D}^* \vdash \mathrm{G}^* \preceq \xi$ by lemma C.6

(9) $\mathrm{D}^*;\mathrm{G}^* \vdash \Lambda.\Lambda.[\![e]\!]_{\mathcal{C}} : {}^{\xi}\forall\alpha{:}\star.{}^{\mathsf{L}}\forall\beta{:}k^*.{}^{\mathsf{L}}(c'^*(t^* \multimap \langle\!\langle \alpha, c' \rangle\!\rangle_{\mathcal{C}}^{-}) \multimap \langle\!\langle \alpha, c' \rangle\!\rangle_{\mathcal{C}}^{+})$ by (7–8)

(10) $\mathrm{D}^*;\mathrm{G}^* \vdash (\Lambda.e)^* : {}^{\xi}((\forall^{c'}\beta{:}k.t)^*)$ by (1–2, 9).

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{C}} t_1 \preceq \xi \qquad \mathrm{D} \vdash_{\mathcal{C}} t_2 : \star \qquad \mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} v_1 : t_1 ; \bot_{\mathcal{C}}}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} \mathsf{inl}\, v_1 : {}^{\xi}(t_1 \oplus t_2) ; \bot_{\mathcal{C}}}$.

We want to show that $\mathrm{D}^*;\mathrm{G}^* \vdash (\mathsf{inl}\, v_1)^* : {}^{\xi}((t_1 \oplus t_2)^*)$. Note that

(1) $(t_1 \oplus t_2)^* = t_1{}^* \oplus t_2{}^*$ by def. $\overline{t}^*$ and

(2) $(\mathsf{inl}\, v_1)^* = \mathsf{inl}\, v_1{}^*$ by def. v^*.

Then

(3) $\mathrm{D}^* \vdash t_1{}^* \preceq \xi$ by lemma C.6

(4) $\mathrm{D}^* \vdash t_2{}^* : \star$ by lemma 8.5

(5) $\mathrm{D}^*;\mathrm{G}^* \vdash v_1{}^* : t_1{}^*$ by IH (inner)

(6) $\mathrm{D}^*;\mathrm{G}^* \vdash \mathsf{inl}\, v_1{}^* : {}^{\xi}(t_1{}^* \oplus t_2{}^*)$ by (3–5)

(7) $\mathrm{D}^*;\mathrm{G}^* \vdash (\mathsf{inl}\, v_1)^* : ({}^{\xi}(t_1 \oplus t_2))^*$ by (1–2, 6).

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{C}} t_2 \preceq \xi \qquad \mathrm{D} \vdash_{\mathcal{C}} t_1 : \star \qquad \mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} v_2 : t_2 ; \bot_{\mathcal{C}}}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} \mathsf{inr}\, v_2 : {}^{\xi}(t_1 \oplus t_2) ; \bot_{\mathcal{C}}}$.

By symmetry with the previous case.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{C}} \xi' : \mathsf{QUAL} \qquad \mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} v_1 : {}^{\xi_1}(t_1 \xrightarrow{c'}\!\!\circ\, t) ; \bot_{\mathcal{C}} \qquad \mathrm{D} \vdash_{\mathcal{C}} \xi_1 \preceq \xi \qquad \mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} v_2 : {}^{\xi_2}(t_2 \xrightarrow{c'}\!\!\circ\, t) ; \bot_{\mathcal{C}} \qquad \mathrm{D} \vdash_{\mathcal{C}} \xi_2 \preceq \xi}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} [v_1, v_2] : {}^{\xi}({}^{\xi'}(t_1 \oplus t_2) \xrightarrow{c'}\!\!\circ\, t) ; \bot_{\mathcal{C}}}$.

We want to show that $\mathrm{D}^*;\mathrm{G}^* \vdash [v_1, v_2]^* : {}^{\xi}(({}^{\xi'}(t_1 \oplus t_2) \xrightarrow{c'}\!\!\circ\, t)^*)$. Note that

(1) $({}^{\xi'}(t_1 \oplus t_2) \xrightarrow{c'}\!\!\circ\, t)^* = \forall\alpha{:}\star.{}^{\mathsf{L}}({}^{\xi'}(t_1{}^* \oplus t_2{}^*) \multimap {}^{\mathsf{L}}(c'^*(t^* \multimap \langle\!\langle \alpha, c' \rangle\!\rangle_{\mathcal{C}}^{-}) \multimap \langle\!\langle \alpha, c' \rangle\!\rangle_{\mathcal{C}}^{+}))$ by def. $\overline{t}^*$ and

(2) $[v_1, v_2]^* = \Lambda.[\lambda x.v_1{}^*\_x, \lambda x.v_2{}^*\_x]$ by def. v^*.

Then:

(3) $\mathrm{D}^*, \alpha{:}\star \vdash \xi' : \mathsf{QUAL}$ by lemma 8.5, weak.

(4) $\mathrm{D}^*, \alpha{:}\star; \bullet, x{:}{t_1}^* \vdash x : {t_1}^*$ by rule T-VAR

(5) $\mathrm{D}^*, \alpha{:}\star; \mathrm{G}^* \vdash {v_1}^* : {}^{\xi_1}((t_1 \xrightarrow{c'} t)^*)$ by IH (inner), weak.

(6) $\mathrm{D}^*, \alpha{:}\star; \mathrm{G}^* \vdash {v_2}^* : {}^{\xi_2}((t_2 \xrightarrow{c'} t)^*)$ by IH (inner), weak.

By lemma 8.4, there exist some Γ_{11} and Γ_{12} such that

(7) $\mathrm{D}^*, \alpha{:}\star \vdash \mathrm{G}^* \rightsquigarrow \Gamma_{11} \boxplus \Gamma_{12}$,

(8) $\mathrm{D}^*, \alpha{:}\star; \Gamma_{11} \vdash {v_1}^* : {}^{\xi_1}((t_1 \xrightarrow{c'} t)^*)$,

(9) $\mathrm{D}^*, \alpha{:}\star \vdash \Gamma_{12} \preceq \mathsf{A}$, and

(10) $\mathrm{D}^*, \alpha{:}\star \vdash \Gamma_{11} \preceq \xi_1$,

and likewise, there exist some Γ_{21} and Γ_{22} such that

(11) $\mathrm{D}^*, \alpha{:}\star \vdash \mathrm{G}^* \rightsquigarrow \Gamma_{21} \boxplus \Gamma_{22}$,

(12) $\mathrm{D}^*, \alpha{:}\star; \Gamma_{21} \vdash {v_2}^* : {}^{\xi_2}((t_2 \xrightarrow{c'} t)^*)$,

(13) $\mathrm{D}^*, \alpha{:}\star \vdash \Gamma_{22} \preceq \mathsf{A}$, and

(14) $\mathrm{D}^*, \alpha{:}\star \vdash \Gamma_{21} \preceq \xi_2$.

Let $\Gamma_1 = \Gamma_{11} \cup \Gamma_{21}$. Then:

(15) $\mathrm{D}^*, \alpha{:}\star \vdash \Gamma_1 \preceq \xi$ by (10, 14)

(16) $\mathrm{D}^*, \alpha{:}\star; \Gamma_1 \vdash {v_1}^* : {}^{\xi_1}((t_1 \xrightarrow{c'} t)^*)$ by (5), weak.

(17) $\mathrm{D}^*, \alpha{:}\star; \Gamma_1 \vdash {v_1}^* : {}^{\xi_1}\forall \beta{:}\star.\, {}^{\mathsf{L}}({t_1}^* \multimap {}^{\mathsf{L}}({c'}^*(t^* \multimap \langle\!\langle \beta, c' \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \beta, c' \rangle\!\rangle^+_{\mathcal{C}}))$ by def. $\overline{t}^*$, (5)

(18) $\mathrm{D}^*, \alpha{:}\star; \Gamma_1 \vdash {v_1}^*\ \_ : {}^{\mathsf{L}}({t_1}^* \multimap {}^{\mathsf{L}}({c'}^*(t^* \multimap \langle\!\langle \alpha, c' \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c' \rangle\!\rangle^+_{\mathcal{C}}))$ by (17)

(19) $\mathrm{D}^*, \alpha{:}\star \vdash \Gamma_1, x{:}{t_1}^* \rightsquigarrow \Gamma_1 \boxplus \bullet, x{:}{t_1}^*$ by rules S-CONSL and S-CONSR

(20) $\mathrm{D}^*, \alpha{:}\star; \Gamma_1, x{:}{t_1}^* \vdash {v_1}^*\ \_\, x : {}^{\mathsf{L}}({c'}^*(t^* \multimap \langle\!\langle \alpha, c' \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c' \rangle\!\rangle^+_{\mathcal{C}})$ by (4, 18–19)

(21) $\mathrm{D}^*, \alpha{:}\star; \Gamma_1 \vdash \lambda x.\, {v_1}^*\ \_\, x : {}^{\mathsf{L}}({t_1}^* \multimap {}^{\mathsf{L}}({c'}^*(t^* \multimap \langle\!\langle \alpha, c' \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c' \rangle\!\rangle^+_{\mathcal{C}}))$ by (20)

(22) $\mathrm{D}^*, \alpha{:}\star; \Gamma_1 \vdash \lambda x.\, {v_2}^*\ \_\, x : {}^{\mathsf{L}}({t_2}^* \multimap {}^{\mathsf{L}}({c'}^*(t^* \multimap \langle\!\langle \alpha, c' \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c' \rangle\!\rangle^+_{\mathcal{C}}))$ by symmetry

(23) $\mathrm{D}^*, \alpha{:}\star; \Gamma_1 \vdash [\lambda x.{v_1}^*\_x, \lambda x.{v_2}^*\_x] : {}^{\mathsf{L}}(\xi'({t_1}^* \oplus {t_2}^*) \multimap {}^{\mathsf{L}}({c'}^*(t^* \multimap \langle\!\langle \alpha, c' \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c' \rangle\!\rangle^+_{\mathcal{C}}))$ by (3, 21–22)

(24) $\mathrm{D}^*; \Gamma_1 \vdash \Lambda.[\lambda x.{v_1}^*\_x, \lambda x.{v_2}^*\_x] : {}^{\xi}((\xi'(t_1 \oplus t_2) \xrightarrow{c'}\!\!\circ\, t)^*)$ by (1, 15, 23), rule T-TAbs

(25) $\mathrm{D}^*; \Gamma_1 \vdash [v_1, v_2]^* : ({}^{\xi}(\xi'(t_1 \oplus t_2) \xrightarrow{c'}\!\!\circ\, t))^*$ by (2).

$$\textbf{Case}\ \frac{\begin{array}{c}\mathrm{D} \vdash_{\mathcal{C}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_2 \\ \mathrm{D}; \mathrm{G}_1 \vdash_{\mathcal{C}} v_1 : t_1 ; \bot_{\mathcal{C}} \qquad \mathrm{D} \vdash_{\mathcal{C}} t_1 \preceq \xi \\ \mathrm{D}; \mathrm{G}_2 \vdash_{\mathcal{C}} v_2 : t_2 ; \bot_{\mathcal{C}} \qquad \mathrm{D} \vdash_{\mathcal{C}} t_2 \preceq \xi\end{array}}{\mathrm{D}; \mathrm{G} \vdash_{\mathcal{C}} \langle v_1, v_2 \rangle : {}^{\xi}(t_1 \otimes t_2); \bot_{\mathcal{C}}}.$$

We want to show that $\mathrm{D}^*; \mathrm{G}^* \vdash \langle v_1, v_2 \rangle^* : {}^{\xi}((t_1 \otimes t_2)^*)$. Note that

(1) $(t_1 \otimes t_2)^* = {t_1}^* \otimes {t_2}^*$ by def. $\overline{t}^*$ and

(2) $\langle v_1, v_2 \rangle^* = \langle {v_1}^*, {v_2}^* \rangle$ by def. v^*.

Then

(3) $\mathrm{D}^* \vdash \mathrm{G}^* \rightsquigarrow {\mathrm{G}_1}^* \boxplus {\mathrm{G}_2}^*$ by lemma C.6

(4) $\mathrm{D}^*; {\mathrm{G}_1}^* \vdash {v_1}^* : {t_1}^*$ by IH (inner)

(5) $\mathrm{D}^* \vdash {t_1}^* \preceq \xi$ by lemma C.6

(6) $\mathrm{D}^*; {\mathrm{G}_2}^* \vdash {v_2}^* : {t_2}^*$ by IH (inner)

(7) $\mathrm{D}^* \vdash {t_2}^* \preceq \xi$ by lemma C.6

(8) $\mathrm{D}^*; \mathrm{G}^* \vdash \langle {v_1}^*, {v_2}^* \rangle : {}^{\xi}({t_1}^* \otimes {t_2}^*)$ by (3)–(7)

(9) $\mathrm{D}^*; \mathrm{G}^* \vdash \langle v_1, v_2 \rangle^* : ({}^{\xi}(t_1 \otimes t_2))^*$ by (1–2, 8).

$$\textbf{Case}\ \frac{\begin{array}{c}\mathrm{D} \vdash_{\mathcal{C}} \xi' : \mathsf{QUAL} \\ \mathrm{D}; \mathrm{G} \vdash_{\mathcal{C}} v : {}^{\xi}(t_1 \xrightarrow{c_1}\!\!\circ\, {}^{\xi}(t_2 \xrightarrow{c_2}\!\!\circ\, t)); \bot_{\mathcal{C}} \qquad \mathrm{D} \vdash_{\mathcal{C}} c_1 \otimes c_2 : \mathsf{CTL}\end{array}}{\mathrm{D}; \mathrm{G} \vdash_{\mathcal{C}} \mathsf{uncurry}\ v : {}^{\xi}(\xi'(t_1 \otimes t_2) \xrightarrow{c_1 \otimes c_2}\!\!\circ\, t); \bot_{\mathcal{C}}}.$$

We want to show that $\mathrm{D}^*; \mathrm{G}^* \vdash (\mathsf{uncurry}\ v)^* : {}^{\xi}((\xi'(t_1 \otimes t_2) \xrightarrow{c_1 \otimes c_2}\!\!\circ\, t)^*)$. Then:

(1) $(\xi'(t_1 \otimes t_2) \xrightarrow{c_1 \otimes c_2}\!\!\circ\, t)^* = \forall \alpha{:}\star.\, {}^{\mathsf{L}}(\xi'({t_1}^* \otimes {t_2}^*) \multimap {}^{\mathsf{L}}({(c_1 \otimes c_2)}^*(t^* \multimap \langle\!\langle \alpha, c_1 \otimes c_2 \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c_1 \otimes c_2 \rangle\!\rangle^+_{\mathcal{C}}))$ by def. $\overline{t}^*$ and

(2) $(\mathsf{uncurry}\ v)^* = \Lambda.\ \mathsf{uncurry}(\lambda x_1.\lambda x_2.[\![v\,x_1\,x_2]\!]_{\mathcal{C}})$ by def. v^*.

(3) $\mathrm{D}, \alpha{:}\star; \mathrm{G} \vdash_{\mathcal{C}} v : {}^{\xi}(t_1 \xrightarrow{c_1}\!\circ\ {}^{\xi}(t_2 \xrightarrow{c_2}\!\circ\ t)); \bot_{\mathcal{C}}$ by prem., weak.

(4) $\mathrm{D}, \alpha{:}\star; \bullet, x_1{:}t_1 \vdash_{\mathcal{C}} x_1 : t_1; \bot_{\mathcal{C}}$ by rule C-T-VAR

(5) $\mathrm{D}, \alpha{:}\star; \bullet, x_2{:}t_2 \vdash_{\mathcal{C}} x_2 : t_2; \bot_{\mathcal{C}}$ by rule C-T-VAR

(6) $\mathrm{D}, \alpha{:}\star; \mathrm{G}, x_1{:}t_1 \vdash_{\mathcal{C}} v\,x_1 : {}^{\xi}(t_2 \xrightarrow{c_2}\!\circ\ t); c_1$ by (3–4), rule C-T-APP

(7) $\mathrm{D}, \alpha{:}\star; \mathrm{G}, x_1{:}t_1, x_2{:}t_2 \vdash_{\mathcal{C}} v\,x_1\,x_2 : t; c_1 \otimes c_2$ by (5–6), rule C-T-APP

(8) $|v\,x_1\,x_2| = 2 + |v|$ by def. $|\cdot|$

(9) $|\mathsf{uncurry}\ v| = 3 + |v|$ by def. $|\cdot|$

(10) $|v\,x_1\,x_2| < |\mathsf{uncurry}\ v|$ by (8–9)

(11) $\mathrm{D}^* \vdash (c_1 \otimes c_2)^* \preceq (c_1 \otimes c_2)^*$ by rule QSUB-REFL

(12) $\mathrm{D}^*, \alpha{:}\star; \mathrm{G}^*, x_1{:}{t_1}^*, x_2{:}{t_2}^* \vdash [\![v\,x_1\,x_2]\!]_{\mathcal{C}} : {}^{\mathsf{L}}({}^{(c_1 \otimes c_2)^*}(t^* \multimap \langle\!\langle \alpha, c_1 \otimes c_2 \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c_1 \otimes c_2 \rangle\!\rangle^+_{\mathcal{C}})$ by IH (outer), (10–11)

(13) $\mathrm{D}^*, \alpha{:}\star; \mathrm{G}^* \vdash \lambda x_1.\lambda x_2.[\![v\,x_1\,x_2]\!]_{\mathcal{C}} : {}^{\mathsf{L}}({t_1}^* \multimap {}^{\mathsf{L}}({t_2}^* \multimap {}^{\mathsf{L}}({}^{(c_1 \otimes c_2)^*}(t^* \multimap \langle\!\langle \alpha, c_1 \otimes c_2 \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c_1 \otimes c_2 \rangle\!\rangle^+_{\mathcal{C}})))$ by (12), rule T-ABS^2

(14) $\mathrm{D}^*, \alpha{:}\star \vdash \xi' : \mathsf{QUAL}$ by lemma 8.5, weak.

(15) $\mathrm{D}^*, \alpha{:}\star; \mathrm{G}^* \vdash \mathsf{uncurry}(\lambda x_1.\lambda x_2.[\![v\,x_1\,x_2]\!]_{\mathcal{C}}) : {}^{\mathsf{L}}({}^{\xi'}({t_1}^* \otimes {t_2}^*) \multimap {}^{\mathsf{L}}({}^{(c_1 \otimes c_2)^*}(t^* \multimap \langle\!\langle \alpha, c_1 \otimes c_2 \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c_1 \otimes c_2 \rangle\!\rangle^+_{\mathcal{C}}))$ by (13–14)

By lemma 8.4, there exist some Γ_1 and Γ_2 such that

(16) $\mathrm{D}^*, \alpha{:}\star \vdash \mathrm{G}^* \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$,

(17) $\mathrm{D}^*, \alpha{:}\star; \Gamma_1 \vdash \mathsf{uncurry}(\lambda x_1.\lambda x_2.[\![v\,x_1\,x_2]\!]_{\mathcal{C}}) : {}^{\mathsf{L}}({}^{\xi'}({t_1}^* \otimes {t_2}^*) \multimap {}^{\mathsf{L}}({}^{(c_1 \otimes c_2)^*}(t^* \multimap \langle\!\langle \alpha, c_1 \otimes c_2 \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c_1 \otimes c_2 \rangle\!\rangle^+_{\mathcal{C}}))$

(18) $\mathrm{D}^*, \alpha{:}\star \vdash \Gamma_2 \preceq \mathsf{A}$, and

(19) $\mathrm{D}^*, \alpha{:}\star \vdash \Gamma_1 \preceq \xi$.

Then:

(20) $\mathrm{D}^*;\Gamma_1 \vdash \Lambda.\,\mathsf{uncurry}(\lambda x_1.\,\lambda x_2.\,[\![v\,x_1\,x_2]\!]_{\mathcal{C}}) : {}^{\xi}\forall\alpha{:}\star.\,{}^{\mathsf{L}}({}^{\xi'}(t_1{}^* \otimes t_2{}^*) \multimap {}^{\mathsf{L}}({}^{(c_1 \otimes c_2)^*}(t^* \multimap \langle\!\langle \alpha, c_1 \otimes c_2 \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c_1 \otimes c_2 \rangle\!\rangle^+_{\mathcal{C}}))$
by (17, 19)

(21) $\mathrm{D}^*;\Gamma_1 \vdash (\mathsf{uncurry}\ v)^* : ({}^{\xi}(({}^{\xi'}(t_1 \otimes t_2) \xrightarrow{c_1 \otimes c_2}\!\circ\, t)))^*$
by (1–2, 20)

(22) $\mathrm{D}^*;\mathrm{G}^* \vdash (\mathsf{uncurry}\ v)^* : ({}^{\xi}(({}^{\xi'}(t_1 \otimes t_2) \xrightarrow{c_1 \otimes c_2}\!\circ\, t)))^*$
by weak., (18, 21)

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{C}} \xi : \mathsf{QUAL}}{\mathrm{D};\bullet \vdash_{\mathcal{C}} \langle\rangle : {}^{\xi}1;\bot_{\mathcal{C}}}$.

We want to show that $\mathrm{D}^*;\mathrm{G}^* \vdash \langle\rangle^* : {}^{\xi}(1^*)$. Note that

(1) $1^* = 1$ by def. $\overline{t}^*$ and

(2) $\langle\rangle^* = \langle\rangle$ by def. v^*.

Then:

(3) $\mathrm{D}^* \vdash \xi : \mathsf{QUAL}$ by lemma 8.5

(4) $\mathrm{D}^*;\bullet \vdash \langle\rangle^* : {}^{\xi}1^*$ by rule T-UNIT, (1–2).

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{C}} \xi : \mathsf{QUAL} \qquad \mathrm{D} \vdash_{\mathcal{C}} t : \star \qquad \mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} v : {}^{\xi'}1;\bot_{\mathcal{C}}}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} \mathsf{ignore}\ v : {}^{\xi}(t \xrightarrow{\bot_{\mathcal{C}}}\!\circ\, t);\bot_{\mathcal{C}}}$.

We want to show that $\mathrm{D}^*;\mathrm{G}^* \vdash (\mathsf{ignore}\ v)^* : {}^{\xi}((t \xrightarrow{\bot_{\mathcal{C}}}\!\circ\, t)^*)$. Note that

(1) $(t \xrightarrow{\bot_{\mathcal{C}}}\!\circ\, t)^* = \forall\alpha{:}\star.\,{}^{\mathsf{L}}(t^* \multimap {}^{\mathsf{L}}({}^{\mathsf{L}}(t^* \multimap \langle\!\langle\alpha\rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle\alpha\rangle\!\rangle_{\mathcal{C}}))$
by def. $\overline{t}^*$ and

(2) $(\mathsf{ignore}\ v)^* = \Lambda.\,\lambda x.\,\mathsf{ignore}\ v^*\,[\![x]\!]_{\mathcal{C}}$ by def. v^*.

Then:

(3) $\mathrm{D}^* \vdash \xi : \mathsf{QUAL}$ by lemma 8.5

(4) $\mathrm{D}^* \vdash t^* : \star$ by lemma 8.5

(5) $\mathrm{D}^*, \alpha{:}\star;\mathrm{G}^* \vdash v^* : {}^{\xi'}1$ by IH (inner), def. $\overline{t}^*$

(6) $\mathrm{D}^*, \alpha{:}\star \vdash {}^{\mathsf{L}}({}^{\mathsf{L}}(t^* \multimap \langle\!\langle\alpha\rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle\alpha\rangle\!\rangle_{\mathcal{C}}) : \star$ by (4), rule K-VAR, property 8.4.2, ...

(7) $\mathrm{D}^*, \alpha{:}\star; \mathrm{G}^* \vdash \mathsf{ignore}\ v^* : {}^{\xi}(\mathsf{L}(\mathsf{L}(t^* \multimap \langle\!\langle \alpha \rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle \alpha \rangle\!\rangle_{\mathcal{C}}) \multimap \mathsf{L}(\mathsf{L}(t^* \multimap \langle\!\langle \alpha \rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle \alpha \rangle\!\rangle_{\mathcal{C}}))$ by (5) and rule T-UNITE

By lemma 8.4, there exist some Γ_1 and Γ_2 such that

(8) $\mathrm{D}^*, \alpha{:}\star \vdash \mathrm{G}^* \rightsquigarrow \Gamma_1 \boxplus \Gamma_2$,

(9) $\mathrm{D}^*, \alpha{:}\star; \Gamma_1 \vdash \mathsf{ignore}\ v^* : {}^{\xi}(\mathsf{L}(\mathsf{L}(t^* \multimap \langle\!\langle \alpha \rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle \alpha \rangle\!\rangle_{\mathcal{C}}) \multimap \mathsf{L}(\mathsf{L}(t^* \multimap \langle\!\langle \alpha \rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle \alpha \rangle\!\rangle_{\mathcal{C}}))$

(10) $\mathrm{D}^*, \alpha{:}\star \vdash \Gamma_2 \preceq \mathsf{A}$, and

(11) $\mathrm{D}^*, \alpha{:}\star \vdash \Gamma_1 \preceq \xi$.

Then:

(12) $\mathrm{D}, \alpha{:}\star; \bullet, x{:}t \vdash_{\mathcal{C}} x : t ; \bot_{\mathcal{C}}$ by rule C-T-VAR

(13) $\mathrm{D}^*, \alpha{:}\star \vdash \alpha : \star$ by rule K-TYPE

(14) $|x| = 1 < 1 + |v| = |\mathsf{ignore}\ v|$ by def. $|\cdot|$

(15) $\mathrm{D}^* \vdash \mathsf{L} \preceq \bot_{\mathcal{C}}{}^*$ by rule QSUB-REFL

(16) $\mathrm{D}^*, \alpha{:}\star; \bullet, x{:}t^* \vdash [\![x]\!]_{\mathcal{C}} : \mathsf{L}(\mathsf{L}(t^* \multimap \langle\!\langle \alpha \rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle \alpha \rangle\!\rangle_{\mathcal{C}})$ by IH (outer), (12–15)

(17) $\mathrm{D}^*, \alpha{:}\star \vdash \Gamma_1, x{:}t^* \rightsquigarrow \Gamma_1 \boxplus \bullet, x{:}t^*$ by rules S-CONSR and S-CONSL, ...

(18) $\mathrm{D}^*, \alpha{:}\star; \Gamma_1, x{:}t^* \vdash \mathsf{ignore}\ v^*\ [\![x]\!]_{\mathcal{C}} : \mathsf{L}(\mathsf{L}(t^* \multimap \langle\!\langle \alpha \rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle \alpha \rangle\!\rangle_{\mathcal{C}})$ by (7, 16–17)

(19) $\mathrm{D}^*, \alpha{:}\star; \Gamma_1 \vdash \lambda x.\ \mathsf{ignore}\ v^*\ [\![x]\!]_{\mathcal{C}} : \mathsf{L}(t^* \multimap \mathsf{L}(\mathsf{L}(t^* \multimap \langle\!\langle \alpha \rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle \alpha \rangle\!\rangle_{\mathcal{C}}))$ by (18)

(20) $\mathrm{D}^*; \Gamma_1 \vdash \Lambda.\, \lambda x.\ \mathsf{ignore}\ v^*\ [\![x]\!]_{\mathcal{C}} : {}^{\xi}\forall \alpha{:}\star.\, \mathsf{L}(t^* \multimap \mathsf{L}(\mathsf{L}(t^* \multimap \langle\!\langle \alpha \rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle \alpha \rangle\!\rangle_{\mathcal{C}}))$ by (11, 20)

(21) $\mathrm{D}^*; \Gamma_1 \vdash (\mathsf{ignore}\ v)^* : ({}^{\xi}(t \overset{\bot_{\mathcal{C}}}{\multimap} t))^*$ by (1–2)

(22) $\mathrm{D}^*; \mathrm{G}^* \vdash (\mathsf{ignore}\ v)^* : ({}^{\xi}(t \overset{\bot_{\mathcal{C}}}{\multimap} t))^*$ by weak. and (10).

That completes the value case. We continue with the non-value expressions, again considering type derivations:

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{C}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_2 \qquad \mathrm{D}; \mathrm{G}_1 \vdash_{\mathcal{C}} e : t \,;\, c \qquad \mathrm{D} \vdash_{\mathcal{C}} \mathrm{G}_2 \preceq \mathsf{A}}{\mathrm{D}; \mathrm{G} \vdash_{\mathcal{C}} e : t \,;\, c}$.

By lemma C.6 and IH.

Case $\dfrac{\mathrm{D}; \mathrm{G} \vdash_{\mathcal{C}} e : t \,;\, c' \qquad \mathrm{D} \vdash_{\mathcal{C}} c' \preceq c}{\mathrm{D}; \mathrm{G} \vdash_{\mathcal{C}} e : t \,;\, c}$.

(1) $\mathrm{D}^* \vdash c^* \preceq c'^*$ by property 8.4.3

(2) $\mathrm{D}^* \vdash \xi \preceq c^*$ by antecedent

(3) $\mathrm{D}^* \vdash \xi \preceq c'^*$ by (1–2), lemma C.1

(4) $\exists \tau''. \langle\!\langle \tau'', c' \rangle\!\rangle^-_{\mathcal{C}} = \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}$ and $\langle\!\langle \tau'', c' \rangle\!\rangle^+_{\mathcal{C}} = \langle\!\langle \tau', c \rangle\!\rangle^+_{\mathcal{C}}$ by (1), property 8.4.4

(5) $\mathrm{D}^*; \mathrm{G}^* \vdash [\![e]\!]_{\mathcal{C}} : \mathsf{L}(^{\xi}(t^* \multimap \langle\!\langle \tau'', c' \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau'', c' \rangle\!\rangle^+_{\mathcal{C}})$ by IH(τ''), (3)

(6) $\mathrm{D}^*; \mathrm{G}^* \vdash [\![e]\!]_{\mathcal{C}} : \mathsf{L}(^{\xi}(t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c \rangle\!\rangle^+_{\mathcal{C}})$ by (4–5).

Case $\dfrac{\begin{array}{c}\mathrm{D} \vdash_{\mathcal{C}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_2 \qquad \mathrm{D}; \mathrm{G}_1 \vdash_{\mathcal{C}} e_1 : {}^{\xi_1}(t_1 \xrightarrow{c'} t_2) \,;\, c'_1 \qquad \mathrm{D}; \mathrm{G}_2 \vdash_{\mathcal{C}} e_2 : t_1 \,;\, c'_2 \\ \mathrm{D} \vdash_{\mathcal{C}} \mathrm{G}_2 \preceq \xi_2 \qquad \mathrm{D} \vdash_{\mathcal{C}} c'_1 \succeq \xi_2 \qquad \mathrm{D} \vdash_{\mathcal{C}} c'_2 \succeq \xi_1 \qquad \mathrm{D} \vdash_{\mathcal{C}} c'_1 \oslash c'_2 \oslash c' : \mathsf{CTL}\end{array}}{\mathrm{D}; \mathrm{G} \vdash_{\mathcal{C}} e_1\, e_2 : t_2 \,;\, c'_1 \oslash c'_2 \oslash c'}$.

We want to show that

$$\mathrm{D}^*; \mathrm{G}^* \vdash [\![e_1\, e_2]\!]_{\mathcal{C}} : \mathsf{L}(^{\xi_0}({t_2}^* \multimap \langle\!\langle \tau', c'_1 \oslash c'_2 \oslash c' \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c'_1 \oslash c'_2 \oslash c' \rangle\!\rangle^+_{\mathcal{C}}).$$

Consider whether the term $e_1\, e_2$ has a control effect:

Case $c'_1 \oslash c'_2 \oslash c' \neq \bot_{\mathcal{C}}$.

By property 8.4.2, there exist some $c_1 \neq \bot_{\mathcal{C}}$, $c_2 \neq \bot_{\mathcal{C}}$, and $c \neq \bot_{\mathcal{C}}$ such that

(1) $\mathrm{D} \vdash_{\mathcal{C}} c'_1 \preceq c_1$,

(2) $\mathrm{D} \vdash_{\mathcal{C}} c'_2 \preceq c_2$,

(3) $\mathrm{D} \vdash_{\mathcal{C}} c' \preceq c$,

(4) $c_1 \otimes c_2 \otimes c = c'_1 \otimes c'_2 \otimes c'$, and

(5) $\mathrm{D} \vdash_{\mathcal{C}} c_1 \otimes c_2 \otimes c : \mathsf{CTL}$.

From the antecedent of the lemma to be proved,

(6) $\mathrm{D}^* \vdash \xi_0 \preceq (c'_1 \otimes c'_2 \otimes c')^*$.

Then:

(7) $[\![e_1\, e_2]\!]_{\mathcal{C}} = \lambda y.\, [\![e_1]\!]_{\mathcal{C}}(\lambda x_1.\, [\![e_2]\!]_{\mathcal{C}}(\lambda x_2.\, x_1\_x_2(\lambda x.\, y\, x)))$
by def. $[\![e]\!]_{\mathcal{C}}$

(8) $\exists \tau''_1.\langle\!\langle \tau''_1, c'_1 \rangle\!\rangle^-_{\mathcal{C}} = \langle\!\langle \tau', c_1 \rangle\!\rangle^-_{\mathcal{C}}$ and $\langle\!\langle \tau''_1, c'_1 \rangle\!\rangle^+_{\mathcal{C}} = \langle\!\langle \tau', c_1 \rangle\!\rangle^+_{\mathcal{C}}$
by (1), property 8.4.4

(9) $\exists \tau''_2.\langle\!\langle \tau''_2, c'_2 \rangle\!\rangle^-_{\mathcal{C}} = \langle\!\langle \tau', c_2 \rangle\!\rangle^-_{\mathcal{C}}$ and $\langle\!\langle \tau''_2, c'_2 \rangle\!\rangle^+_{\mathcal{C}} = \langle\!\langle \tau', c_2 \rangle\!\rangle^+_{\mathcal{C}}$
by (2), property 8.4.4

(10) $\exists \tau''.\langle\!\langle \tau'', c' \rangle\!\rangle^-_{\mathcal{C}} = \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}$ and $\langle\!\langle \tau'', c' \rangle\!\rangle^+_{\mathcal{C}} = \langle\!\langle \tau', c \rangle\!\rangle^+_{\mathcal{C}}$
by (3), property 8.4.4

(11) $\mathrm{D}^*; \mathrm{G}_1{}^* \vdash [\![e_1]\!]_{\mathcal{C}} : {}^{\mathsf{L}}({c'_1}^*({}^{\xi_1}((t_1 \xrightarrow{c'} t_2)^*) \multimap \langle\!\langle \tau', c_1 \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c_1 \rangle\!\rangle^+_{\mathcal{C}})$
by IH(τ''_1), (8)

(12) $\mathrm{D}^*; \mathrm{G}_2{}^* \vdash [\![e_2]\!]_{\mathcal{C}} : {}^{\mathsf{L}}({c'_2}^*(t_1{}^* \multimap \langle\!\langle \tau', c_2 \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c_2 \rangle\!\rangle^+_{\mathcal{C}})$
by IH(τ''_2), (9)

(13) $\mathrm{D}^*; \bullet, x_1{:}^{\xi_1}((t_1 \xrightarrow{c'} t_2)^*) \vdash x_1 : {}^{\xi_1}((t_1 \xrightarrow{c'} t_2)^*)$
by rule T-VAR

(14) $\mathrm{D}^*; \bullet, x_1{:}^{\xi_1}((t_1 \xrightarrow{c'} t_2)^*) \vdash x_1 : {}^{\xi_1}(\forall \alpha{:}\star.\, {}^{\mathsf{L}}(t_1{}^* \multimap {}^{\mathsf{L}}({c'}^*(t_2{}^* \multimap \langle\!\langle \alpha, c' \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c' \rangle\!\rangle^+_{\mathcal{C}})))$
by (13), def. $\overline{t}^*$

(15) $\mathrm{D}^*; \bullet, x_1{:}^{\xi_1}((t_1 \xrightarrow{c'} t_2)^*) \vdash x_1\_ : {}^{\mathsf{L}}(t_1{}^* \multimap {}^{\mathsf{L}}({c'}^*(t_2{}^* \multimap \langle\!\langle \tau'', c' \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau'', c' \rangle\!\rangle^+_{\mathcal{C}}))$
by (14), rule T-TAPP

(16) $\mathrm{D}^*; \bullet, x_1{:}^{\xi_1}((t_1 \xrightarrow{c'} t_2)^*) \vdash x_1\_ : {}^{\mathsf{L}}(t_1{}^* \multimap {}^{\mathsf{L}}({c'}^*(t_2{}^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c \rangle\!\rangle^+_{\mathcal{C}}))$
by (10, 15)

(17) $\mathrm{D}^*; \bullet, x_2{:}t_1{}^* \vdash x_2 : t_1{}^*$
by rule T-VAR

(18) $\mathrm{D}^*; \bullet, x_1{:}^{\xi_1}((t_1 \xrightarrow{c'} t_2)^*), x_2{:}t_1{}^* \vdash x_1\_x_2 : {}^{\mathsf{L}}({c'}^*(t_2{}^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c \rangle\!\rangle^+_{\mathcal{C}})$
by (16–17), rule T-APP

(19) $\mathrm{D}^*; \bullet, y{:}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau', c_1 \otimes c_2 \otimes c \rangle\!\rangle^-_{\mathcal{C}}) \vdash y : {}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau', c_1 \otimes c_2 \otimes c \rangle\!\rangle^-_{\mathcal{C}})$ by rule T-VAR

(20) $\mathrm{D}^* \vdash (c_1' \otimes c_2' \otimes c')^* \preceq c'^*$ by property 3

(21) $\mathrm{D}^* \vdash \xi_0 \preceq c'^*$ by (6, 20), lemma C.1

(22) $\mathrm{D}^*; \bullet, y{:}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau', c_1 \otimes c_2 \otimes c \rangle\!\rangle^-_{\mathcal{C}}) \vdash \lambda x.\, y\, x : {}^{c'^*}(t_2{}^* \multimap \langle\!\langle \tau', c_1 \otimes c_2 \otimes c \rangle\!\rangle^-_{\mathcal{C}})$ by (19, 21), lemma 8.3

(23) $\mathrm{D}^*; \bullet, y{:}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau', c_1 \otimes c_2 \otimes c \rangle\!\rangle^-_{\mathcal{C}}) \vdash \lambda x.\, y\, x : {}^{c'^*}(t_2{}^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}})$ by (22), property 8.4.3a

(24) $\mathrm{D}^*; \bullet, x_1{:}^{\xi_1}((t_1 \xrightarrow{c'} t_2)^*), x_2{:}t_1{}^*, y{:}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau', c_1 \otimes c_2 \otimes c \rangle\!\rangle^-_{\mathcal{C}}) \vdash x_1\_x_2(\lambda x.\, y\, x) : \langle\!\langle \tau', c \rangle\!\rangle^+_{\mathcal{C}}$ by (18, 23), rule T-APP

(25) $\mathrm{D}^* \vdash \xi_1 \preceq c_2'{}^*$ by property 8.4.2

(26) $\mathrm{D}^* \vdash \xi_0 \preceq c_2'{}^*$ by (6), property 3, lemma C.1

(27) $\mathrm{D}^* \vdash \bullet, x_1{:}^{\xi_1}((t_1 \xrightarrow{c'} t_2)^*), y{:}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau', c_1 \otimes c_2 \otimes c \rangle\!\rangle^-_{\mathcal{C}}) \preceq c_2'{}^*$ by (25–26)

(28) $\mathrm{D}^*; \bullet, x_1{:}^{\xi_1}((t_1 \xrightarrow{c'} t_2)^*), y{:}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau', c_1 \otimes c_2 \otimes c \rangle\!\rangle^-_{\mathcal{C}}) \vdash \lambda x_2.\, x_1\_x_2(\lambda x.\, y\, x) : {}^{c_2'{}^*}(t_1{}^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^+_{\mathcal{C}})$ by (24, 27), rule T-ABS

(29) $\mathrm{D}^*; \bullet, x_1{:}^{\xi_1}((t_1 \xrightarrow{c'} t_2)^*), y{:}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau', c_1 \otimes c_2 \otimes c \rangle\!\rangle^-_{\mathcal{C}}) \vdash \lambda x_2.\, x_1\_x_2(\lambda x.\, y\, x) : {}^{c_2'{}^*}(t_1{}^* \multimap \langle\!\langle \tau', c_2 \rangle\!\rangle^-_{\mathcal{C}})$ by (28), property 8.4.3c

(30) $\mathrm{D}^*; \mathrm{G}_2{}^*, x_1{:}^{\xi_1}((t_1 \xrightarrow{c'} t_2)^*), y{:}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau', c_1 \otimes c_2 \otimes c \rangle\!\rangle^-_{\mathcal{C}}) \vdash [\![e_2]\!]_{\mathcal{C}}(\lambda x_2.\, x_1\_x_2(\lambda x.\, y\, x)) : \langle\!\langle \tau', c_2 \rangle\!\rangle^+_{\mathcal{C}}$ by (12, 29), rule T-APP

(31) $\mathrm{D}^* \vdash \xi_2 \preceq c_1'{}^*$ by property 8.4.2

(32) $\mathrm{D}^* \vdash \mathrm{G}_2{}^* \preceq c_1'{}^*$ by (31), lemma C.6

(33) $\mathrm{D}^* \vdash \xi_0 \preceq c_1'{}^*$ by property 3, lemma C.1

(34) $\mathrm{D}^* \vdash \mathrm{G}_2{}^*, y{:}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau', c_1 \otimes c_2 \otimes c \rangle\!\rangle^-_{\mathcal{C}}) \preceq c_1'{}^*$ by (32–33)

(35) $D^*; G_2{}^*, y{:}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau', c_1 \oslash c_2 \oslash c \rangle\!\rangle^-_{\mathcal{C}}) \vdash$
$\lambda x_1.[\![e_2]\!]_{\mathcal{C}}(\lambda x_2.x_1\_x_2(\lambda x.yx)) : {c_1'}^*(^{\xi_1}((t_1 \xrightarrow{c'}\!\!\circ\; t_2)^*) \multimap$
$\langle\!\langle \tau', c_2 \rangle\!\rangle^+_{\mathcal{C}})$ by (30, 34), rule T-ABS

(36) $D^*; G_2{}^*, y{:}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau', c_1 \oslash c_2 \oslash c \rangle\!\rangle^-_{\mathcal{C}}) \vdash$
$\lambda x_1.[\![e_2]\!]_{\mathcal{C}}(\lambda x_2.x_1\_x_2(\lambda x.yx)) : {c_1'}^*(^{\xi_1}((t_1 \xrightarrow{c'}\!\!\circ\; t_2)^*) \multimap$
$\langle\!\langle \tau', c_1 \rangle\!\rangle^-_{\mathcal{C}})$ by (35),
property 8.4.3c

(37) $D^* \vdash G^*, y{:}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau', c_1 \oslash c_2 \oslash c \rangle\!\rangle^-_{\mathcal{C}}) \rightsquigarrow$
$G_1{}^* \boxplus G_2{}^*, y{:}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau', c_1 \oslash c_2 \oslash c \rangle\!\rangle^-_{\mathcal{C}})$
by lemma C.6,
rule S-CONSR

(38) $D^*; G^*, y{:}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau', c_1 \oslash c_2 \oslash c \rangle\!\rangle^-_{\mathcal{C}}) \vdash$
$[\![e_1]\!]_{\mathcal{C}}(\lambda x_1.[\![e_2]\!]_{\mathcal{C}}(\lambda x_2.x_1\_x_2(\lambda x.yx))) : \langle\!\langle \tau', c_1 \rangle\!\rangle^+_{\mathcal{C}}$
by (11, 36–37),
rule T-APP

(39) $D^*; G^* \vdash \lambda y.[\![e_1]\!]_{\mathcal{C}}(\lambda x_1.[\![e_2]\!]_{\mathcal{C}}(\lambda x_2.x_1\_x_2(\lambda x.yx))) :$
$\mathsf{L}(^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau', c_1 \oslash c_2 \oslash c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c_1 \rangle\!\rangle^+_{\mathcal{C}})$
by (38), rule T-ABS

(40) $D^*; G^* \vdash [\![e_1\, e_2]\!]_{\mathcal{C}} : \mathsf{L}(^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau', c_1 \oslash c_2 \oslash c \rangle\!\rangle^-_{\mathcal{C}}) \multimap$
$\langle\!\langle \tau', c_1 \oslash c_2 \oslash c \rangle\!\rangle^+_{\mathcal{C}})$ by (39),
property 8.4.3b, (7).

(41) $D^*; G^* \vdash [\![e_1\, e_2]\!]_{\mathcal{C}} : \mathsf{L}(^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau', c_1' \oslash c_2' \oslash c' \rangle\!\rangle^-_{\mathcal{C}}) \multimap$
$\langle\!\langle \tau', c_1' \oslash c_2' \oslash c' \rangle\!\rangle^+_{\mathcal{C}})$ by (4, 40).

Case $c_1' \oslash c_2' \oslash c' = \bot_{\mathcal{C}}$**.**

By property 8.4.1, $c_1 = c_2 = c = \bot_{\mathcal{C}}$. This is similar to the previous case, but much of the effect management goes away:

(1) $[\![e_1\, e_2]\!]_{\mathcal{C}} = \lambda y.[\![e_1]\!]_{\mathcal{C}}(\lambda x_1.[\![e_2]\!]_{\mathcal{C}}(\lambda x_2.x_1\_x_2(\lambda x.yx)))$
by def. $[\![e]\!]_{\mathcal{C}}$

(2) $D^*; G_1{}^* \vdash [\![e_1]\!]_{\mathcal{C}} : \mathsf{L}(^{\mathsf{L}}(^{\xi_1}((t_1 \xrightarrow{\bot_{\mathcal{C}}}\!\!\circ\; t_2)^*) \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}})$
by IH

(3) $D^*; G_2{}^* \vdash [\![e_2]\!]_{\mathcal{C}} : \mathsf{L}(^{\mathsf{L}}(t_1{}^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}})$
by IH

(4) $\mathrm{D}^*;\bullet, x_1{:}^{\xi_1}((t_1 \xrightarrow{\perp_{\mathcal{C}}} t_2)^*) \vdash x_1 : {}^{\xi_1}((t_1 \xrightarrow{\perp_{\mathcal{C}}} t_2)^*)$
by rule T-VAR

(5) $\mathrm{D}^*;\bullet, x_1{:}^{\xi_1}((t_1 \xrightarrow{\perp_{\mathcal{C}}} t_2)^*) \vdash x_1 : {}^{\xi_1}(\forall \alpha{:}\star.\, {}^{\mathsf{L}}(t_1{}^* \multimap {}^{\mathsf{L}}({}^{\mathsf{L}}(t_2{}^* \multimap \langle\!\langle \alpha \rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle \alpha \rangle\!\rangle_{\mathcal{C}})))$
by (4), def. $\overline{t}^*$

(6) $\mathrm{D}^*;\bullet, x_1{:}^{\xi_1}((t_1 \xrightarrow{\perp_{\mathcal{C}}} t_2)^*) \vdash x_1\_ : {}^{\mathsf{L}}(t_1{}^* \multimap {}^{\mathsf{L}}({}^{\mathsf{L}}(t_2{}^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}))$
by (5), rule T-TAPP

(7) $\mathrm{D}^*;\bullet, x_2{:}t_1{}^* \vdash x_2 : t_1{}^*$
by rule T-VAR

(8) $\mathrm{D}^*;\bullet, x_1{:}^{\xi_1}((t_1 \xrightarrow{\perp_{\mathcal{C}}} t_2)^*), x_2{:}t_1{}^* \vdash x_1\_x_2 : {}^{\mathsf{L}}({}^{\mathsf{L}}(t_2{}^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}})$
by (6–7), rule T-APP

(9) $\mathrm{D}^*;\bullet, y{:}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \vdash y : {}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}})$
by rule T-VAR

(10) $\mathrm{D}^* \vdash \xi_0 \preceq \mathsf{L}$
by rule QSUB-TOP

(11) $\mathrm{D}^*;\bullet, y{:}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \vdash \lambda x.\, y\, x : {}^{\mathsf{L}}(t_2{}^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}})$
by (9–10), lemma 8.3

(12) $\mathrm{D}^*;\bullet, x_1{:}^{\xi_1}((t_1 \xrightarrow{\perp_{\mathcal{C}}} t_2)^*), x_2{:}t_1{}^*, y{:}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \vdash x_1\_x_2(\lambda x.\, y\, x) : \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}$
by (8, 11), rule T-APP

(13) $\mathrm{D}^*;\bullet, x_1{:}^{\xi_1}((t_1 \xrightarrow{\perp_{\mathcal{C}}} t_2)^*), y{:}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \vdash \lambda x_2.\, x_1\_x_2(\lambda x.\, y\, x) : {}^{\mathsf{L}}(t_1{}^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}})$
by (12), rule T-ABS

(14) $\mathrm{D}^*;\mathrm{G}_2{}^*, x_1{:}^{\xi_1}((t_1 \xrightarrow{\perp_{\mathcal{C}}} t_2)^*), y{:}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \vdash [\![e_2]\!]_{\mathcal{C}}(\lambda x_2.\, x_1\_x_2(\lambda x.\, y\, x)) : \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}$
by (3, 13), rule T-APP

(15) $\mathrm{D}^*;\mathrm{G}_2{}^*, y{:}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \vdash \lambda x_1.\, [\![e_2]\!]_{\mathcal{C}}(\lambda x_2.\, x_1\_x_2(\lambda x.\, y\, x)) : {}^{\mathsf{L}}({}^{\xi_1}((t_1 \xrightarrow{\perp_{\mathcal{C}}} t_2)^*) \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}})$
by (14), rule T-ABS

(16) $\mathrm{D}^* \vdash \mathrm{G}^*, y{:}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \rightsquigarrow \mathrm{G}_1{}^* \boxplus \mathrm{G}_2{}^*, y{:}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}})$
by lemma C.6, rule S-CONSR

(17) $\mathrm{D}^*;\mathrm{G}^*, y{:}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \vdash [\![e_1]\!]_{\mathcal{C}}(\lambda x_1.\, [\![e_2]\!]_{\mathcal{C}}(\lambda x_2.\, x_1\_x_2(\lambda x.\, y\, x))) : \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}$
by (2, 15–16), rule T-APP

(18) $\mathrm{D}^*;\mathrm{G}^* \vdash \lambda y.\, [\![e_1]\!]_{\mathcal{C}}(\lambda x_1.\, [\![e_2]\!]_{\mathcal{C}}(\lambda x_2.\, x_1\_x_2(\lambda x.\, y\, x))) : {}^{\mathsf{L}}({}^{\xi_0}(t_2{}^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}})$
by (17), rule T-ABS

(19) $\mathrm{D}^*;\mathrm{G}^* \vdash [\![e_1\, e_2]\!]_{\mathcal{C}} : {}^{\mathsf{L}}(\xi_0({t_2}^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}})$
by (18), (1).

Case $\dfrac{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} e : {}^{\xi}\forall\, {}^{c'_2}\beta{:}k.\, t\,;c'_1 \qquad \mathrm{D} \vdash_{\mathcal{C}} i : k \qquad \mathrm{D} \vdash_{\mathcal{C}} c'_1 \oslash c'_2 : \mathsf{CTL}}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} e\ \_ : \{i/\beta\}t\,;c'_1 \oslash c'_2}$.

We want to show that

$$\mathrm{D}^*;\mathrm{G}^* \vdash [\![e\ \_]\!]_{\mathcal{C}} : {}^{\mathsf{L}}({}^{\xi_0}(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau', c'_1 \oslash c'_2 \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c'_1 \oslash c'_2 \rangle\!\rangle^+_{\mathcal{C}}).$$

Consider whether the term $e\ \_$ has a control effect:

Case $c'_1 \oslash c'_2 \neq \bot_{\mathcal{C}}$.

By property 8.4.2, there exist some $c'_1 \neq \bot_{\mathcal{C}}$ and $c'_2 \neq \bot_{\mathcal{C}}$ such that

(1) $\mathrm{D} \vdash_{\mathcal{C}} c'_1 \preceq c_1$,

(2) $\mathrm{D} \vdash_{\mathcal{C}} c'_2 \preceq c_2$,

(3) $c_1 \oslash c_2 = c'_1 \oslash c'_2$, and

(4) $\mathrm{D} \vdash_{\mathcal{C}} c_1 \oslash c_2 : \mathsf{CTL}$.

From the antecedent of the lemma to be proved,

(5) $\mathrm{D}^* \vdash \xi_0 \preceq (c'_1 \oslash c'_2)^*$.

Then:

(6) $[\![e\ \_]\!]_{\mathcal{C}} = \lambda y.\, [\![e]\!]_{\mathcal{C}}(\lambda x_1.\, x_1\ \_\ \_(\lambda x.\, y\, x))$ by def. $[\![e]\!]_{\mathcal{C}}$

(7) $\exists \tau''_1.\langle\!\langle \tau''_1, c'_1 \rangle\!\rangle^-_{\mathcal{C}} = \langle\!\langle \tau', c_1 \rangle\!\rangle^-_{\mathcal{C}}$ and $\langle\!\langle \tau''_1, c'_1 \rangle\!\rangle^+_{\mathcal{C}} = \langle\!\langle \tau', c_1 \rangle\!\rangle^+_{\mathcal{C}}$
by (1), property 8.4.4

(8) $\exists \tau''_2.\langle\!\langle \tau''_2, c'_2 \rangle\!\rangle^-_{\mathcal{C}} = \langle\!\langle \tau', c_2 \rangle\!\rangle^-_{\mathcal{C}}$ and $\langle\!\langle \tau''_2, c'_2 \rangle\!\rangle^+_{\mathcal{C}} = \langle\!\langle \tau', c_2 \rangle\!\rangle^+_{\mathcal{C}}$
by (2), property 8.4.4

(9) $\mathrm{D}^*;\mathrm{G}^* \vdash [\![e]\!]_{\mathcal{C}} : {}^{\mathsf{L}}({}^{{c'_1}^*}({}^{\xi}((\forall\, {}^{c'_2}\beta{:}k.\, t)^*) \multimap \langle\!\langle \tau', c_1 \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c_1 \rangle\!\rangle^+_{\mathcal{C}})$
by IH(τ''_1), (7)

(10) $\mathrm{D}^*;\bullet, x_1{:}^{\xi}((\forall\, {}^{c'_2}\beta{:}k.\, t)^*) \vdash x_1 : {}^{\xi}((\forall\, {}^{c'_2}\beta{:}k.\, t)^*)$
by rule T-VAR

(11) $\mathrm{D}^*;\bullet, x_1{:}^{\xi}((\forall\, {}^{c'_2}\beta{:}k.\, t)^*) \vdash x_1 : {}^{\xi}\forall \alpha{:}\star.\, {}^{\mathsf{L}}\forall \beta{:}k^*.\, {}^{\mathsf{L}}({}^{{c'_2}^*}(t^* \multimap$
$\langle\!\langle \alpha, c'_2 \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \alpha, c'_2 \rangle\!\rangle^+_{\mathcal{C}})$ by def. $\overline{t}^*$

(12) $\mathrm{D}^*;\bullet, x_1{:}^{\xi}((\forall\,{}^{c'_2}\beta{:}k.\,t)^*) \vdash x_1 : {}^{\mathsf{L}}\forall\beta{:}k^*.\,{}^{\mathsf{L}}({c'_2}^*(t^* \multimap \langle\!\langle \tau''_2, c'_2 \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau''_2, c'_2 \rangle\!\rangle^+_{\mathcal{C}})$ by (11), rule T-TApp

(13) $\mathrm{D}^*;\bullet, x_1{:}^{\xi}((\forall\,{}^{c'_2}\beta{:}k.\,t)^*) \vdash x_1\_ : {}^{\mathsf{L}}\forall\beta{:}k^*.\,{}^{\mathsf{L}}({c'_2}^*(t^* \multimap \langle\!\langle \tau', c_2 \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c_2 \rangle\!\rangle^+_{\mathcal{C}})$ by (8, 12)

(14) $\mathrm{D}^*;\bullet, x_1{:}^{\xi}((\forall\,{}^{c'_2}\beta{:}k.\,t)^*) \vdash x_1\_\_ : {}^{\mathsf{L}}({c'_2}^*(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau', c_2 \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c_2 \rangle\!\rangle^+_{\mathcal{C}})$ by (13), rule T-TApp

(15) $\mathrm{D}^*;\bullet, y{:}^{\xi_0}(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau', c_1 \otimes c_2 \rangle\!\rangle^-_{\mathcal{C}}) \vdash y : {}^{\xi_0}(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau', c_1 \otimes c_2 \rangle\!\rangle^-_{\mathcal{C}})$ by rule T-Var

(16) $\mathrm{D}^* \vdash \xi_0 \preceq {c'_2}^*$ by (5), property 3

(17) $\mathrm{D}^*;\bullet, y{:}^{\xi_0}(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau', c_1 \otimes c_2 \rangle\!\rangle^-_{\mathcal{C}}) \vdash \lambda x.\,y\,x : {}^{{c'_2}^*}(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau', c_1 \otimes c_2 \rangle\!\rangle^-_{\mathcal{C}})$ by (15–16), lemma 8.3

(18) $\mathrm{D}^*;\bullet, y{:}^{\xi_0}(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau', c_1 \otimes c_2 \rangle\!\rangle^-_{\mathcal{C}}) \vdash \lambda x.\,y\,x : {}^{{c'_2}^*}(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau', c_2 \rangle\!\rangle^-_{\mathcal{C}})$ by (17), property 8.4.3a

(19) $\mathrm{D}^*;\bullet, x_1{:}^{\xi}((\forall\,{}^{c'_2}\beta{:}k.\,t)^*), y{:}^{\xi_0}(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau', c_1 \otimes c_2 \rangle\!\rangle^-_{\mathcal{C}}) \vdash x_1\_\_(\lambda x.\,y\,x) : \langle\!\langle \tau', c_2 \rangle\!\rangle^+_{\mathcal{C}}$ by (14, 18), rule T-App

(20) $\mathrm{D}^* \vdash \xi_0 \preceq {c'_1}^*$ by (5), property 3

(21) $\mathrm{D}^*;\bullet, y{:}^{\xi_0}(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau', c_1 \otimes c_2 \rangle\!\rangle^-_{\mathcal{C}}) \vdash \lambda x_1.\,x_1\_\_(\lambda x.\,y\,x) : {}^{{c'_1}^*}({}^{\xi}((\forall\,{}^{c'_2}\beta{:}k.\,t)^*) \multimap \langle\!\langle \tau', c_2 \rangle\!\rangle^+_{\mathcal{C}})$ by (19–20), rule T-Abs

(22) $\mathrm{D}^*;\bullet, y{:}^{\xi_0}(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau', c_1 \otimes c_2 \rangle\!\rangle^-_{\mathcal{C}}) \vdash \lambda x_1.\,x_1\_\_(\lambda x.\,y\,x) : {}^{{c'_1}^*}({}^{\xi}((\forall\,{}^{c'_2}\beta{:}k.\,t)^*) \multimap \langle\!\langle \tau', c_1 \rangle\!\rangle^-_{\mathcal{C}})$ by property 8.4.3c

(23) $\mathrm{D}^*;\mathrm{G}^*, y{:}^{\xi_0}(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau', c_1 \otimes c_2 \rangle\!\rangle^-_{\mathcal{C}}) \vdash [\![e]\!]_{\mathcal{C}}(\lambda x_1.\,x_1\_\_(\lambda x.\,y\,x)) : \langle\!\langle \tau', c_1 \rangle\!\rangle^+_{\mathcal{C}}$ by (9, 22), rule T-App

(24) $\mathrm{D}^*;\mathrm{G}^* \vdash \lambda y.\,[\![e]\!]_{\mathcal{C}}(\lambda x_1.\,x_1\_\_(\lambda x.\,y\,x)) : {}^{\mathsf{L}}({}^{\xi_0}(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau', c_1 \otimes c_2 \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c_1 \rangle\!\rangle^+_{\mathcal{C}})$ by (23), rule T-Abs

(25) $\mathrm{D}^*;\mathrm{G}^* \vdash [\![e\,\_]\!]_{\mathcal{C}} : {}^{\mathsf{L}}({}^{\xi_0}(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau', c_1 \otimes c_2 \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c_1 \otimes c_2 \rangle\!\rangle^+_{\mathcal{C}})$ by (6, 24), property 8.4.3b

(26) $\mathrm{D}^*;\mathrm{G}^* \vdash [\![e\,\_]\!]_{\mathcal{C}} : {}^{\mathsf{L}}({}^{\xi_0}(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau', c'_1 \otimes c'_2 \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c'_1 \otimes c'_2 \rangle\!\rangle^+_{\mathcal{C}})$ by (3, 25)

Case $c'_1 \oslash c'_2 = \bot_{\mathcal{C}}$.

By property 8.4.1, $c'_1 = c'_2 = \bot_{\mathcal{C}}$. This is similar to the previous case, but much of the effect management goes away:

(1) $[\![e\ \_]\!]_{\mathcal{C}} = \lambda y.\, [\![e]\!]_{\mathcal{C}}(\lambda x_1.\, x_1\ \_\ \_(\lambda x.\, y\, x))$ by def. $[\![e]\!]_{\mathcal{C}}$

(2) $\mathrm{D}^*;\mathrm{G}^* \vdash [\![e]\!]_{\mathcal{C}} : \mathsf{L}(\mathsf{L}(^{\xi}((\forall\, ^{\bot_{\mathcal{C}}}\beta{:}k.\, t)^*) \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}})$ by IH

(3) $\mathrm{D}^*;\bullet, x_1{:}^{\xi}((\forall\, ^{\bot_{\mathcal{C}}}\beta{:}k.\, t)^*) \vdash x_1 : {}^{\xi}((\forall\, ^{\bot_{\mathcal{C}}}\beta{:}k.\, t)^*)$ by rule T-VAR

(4) $\mathrm{D}^*;\bullet, x_1{:}^{\xi}((\forall\, ^{\bot_{\mathcal{C}}}\beta{:}k.\, t)^*) \vdash x_1 : {}^{\xi}\forall \alpha{:}\star.\, \mathsf{L}\forall \beta{:}k^*.\, \mathsf{L}(\mathsf{L}(t^* \multimap \langle\!\langle \alpha \rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle \alpha \rangle\!\rangle_{\mathcal{C}})$ by def. $\overline{t}^*$

(5) $\mathrm{D}^*;\bullet, x_1{:}^{\xi}((\forall\, ^{\bot_{\mathcal{C}}}\beta{:}k.\, t)^*) \vdash x_1\ \_ : \mathsf{L}\forall \beta{:}k^*.\, \mathsf{L}(\mathsf{L}(t^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}})$ by (4), rule T-TAPP

(6) $\mathrm{D}^*;\bullet, x_1{:}^{\xi}((\forall\, ^{\bot_{\mathcal{C}}}\beta{:}k.\, t)^*) \vdash x_1\ \_\ \_ : \mathsf{L}(\mathsf{L}(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}})$ by (5), rule T-TAPP

(7) $\mathrm{D}^*;\bullet, y{:}^{\xi_0}(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \vdash y : {}^{\xi_0}(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}})$ by rule T-VAR

(8) $\mathrm{D}^* \vdash \xi_0 \preceq \mathsf{L}$ by rule QSUB-TOP

(9) $\mathrm{D}^*;\bullet, y{:}^{\xi_0}(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \vdash \lambda x.\, y\, x : \mathsf{L}(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}})$ by (7–8), lemma 8.3

(10) $\mathrm{D}^*;\bullet, x_1{:}^{\xi}((\forall\, ^{\bot_{\mathcal{C}}}\beta{:}k.\, t)^*), y{:}^{\xi_0}(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \vdash x_1\ \_\ \_(\lambda x.\, y\, x) : \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}$ by (6, 9), rule T-APP

(11) $\mathrm{D}^*;\bullet, y{:}^{\xi_0}(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \vdash \lambda x_1.\, x_1\ \_\ \_(\lambda x.\, y\, x) : \mathsf{L}(^{\xi}((\forall\, ^{\bot_{\mathcal{C}}}\beta{:}k.\, t)^*) \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}})$ by (8, 10), rule T-ABS

(12) $\mathrm{D}^*;\mathrm{G}^*, y{:}^{\xi_0}(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \vdash [\![e]\!]_{\mathcal{C}}(\lambda x_1.\, x_1\ \_\ \_(\lambda x.\, y\, x)) : \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}$ by (2, 11), rule T-APP

(13) $\mathrm{D}^*;\mathrm{G}^* \vdash \lambda y.\, [\![e]\!]_{\mathcal{C}}(\lambda x_1.\, x_1\ \_\ \_(\lambda x.\, y\, x)) : \mathsf{L}(^{\xi_0}(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}})$ by (12), rule T-ABS

(14) $\mathrm{D}^*;\mathrm{G}^* \vdash [\![e\ \_]\!]_{\mathcal{C}} : \mathsf{L}(^{\xi_0}(\{i^*/\beta\}t^* \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle \tau' \rangle\!\rangle_{\mathcal{C}})$ by (1, 13)

Case $\dfrac{\mathsf{q} \preceq \mathsf{A} \qquad \mathrm{D};\mathrm{G} \vdash_{\mathbb{c}} e : t ; c \qquad \mathrm{D} \vdash_{\mathbb{c}} t \preceq \mathsf{A}}{\mathrm{D};\mathrm{G} \vdash_{\mathbb{c}} \mathsf{new}^{\mathsf{q}}\, e : {}^{\mathsf{q}}\mathsf{ref}\, t ; c}$.

We want to show that $\mathrm{D}^*;\mathrm{G}^* \vdash [\![\mathsf{new}^{\mathsf{q}}\, e]\!]_{\mathbb{c}} : {}^{\mathsf{L}}(^{\xi_0}({}^{\mathsf{q}}\mathsf{ref}\, t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathbb{c}}) \multimap \langle\!\langle \tau', c \rangle\!\rangle^+_{\mathbb{c}})$. Then:

(1) $[\![\mathsf{new}^{\mathsf{q}}\, e]\!]_{\mathbb{c}} = \lambda y.\, [\![e]\!]_{\mathbb{c}}\, (\lambda x.\, y\, (\mathsf{new}^{\mathsf{q}}\, x))$ by def. $[\![e]\!]_{\mathbb{c}}$

(2) $\mathrm{D}^*;\mathrm{G}^* \vdash [\![e]\!]_{\mathbb{c}} : {}^{\mathsf{L}}(^{\xi_0}(t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathbb{c}}) \multimap \langle\!\langle \tau', c \rangle\!\rangle^+_{\mathbb{c}})$ by IH

(3) $\mathrm{D}^*;\bullet, y{:}^{\xi_0}({}^{\mathsf{q}}\mathsf{ref}\, t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathbb{c}}) \vdash y : {}^{\xi_0}({}^{\mathsf{q}}\mathsf{ref}\, t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathbb{c}})$ by rule T-VAR

(4) $\mathrm{D}^*;\bullet, x{:}t^* \vdash x : t^*$ by rule T-VAR

(5) $\mathrm{D}^* \vdash t^* \preceq \mathsf{A}$ by lemma C.6

(6) $\mathrm{D}^*;\bullet, x{:}t^* \vdash \mathsf{new}^{\mathsf{q}}\, x : {}^{\xi}\mathsf{ref}\, t^*$ by (4–5), rule T-NEWUA

(7) $\mathrm{D}^*;\bullet, x{:}t^*, y{:}^{\xi_0}({}^{\xi}\mathsf{ref}\, t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathbb{c}}) \vdash y\, (\mathsf{new}^{\mathsf{q}}\, x) : \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathbb{c}}$ by (3, 6), rule T-APP

(8) $\mathrm{D}^* \vdash \bullet, y{:}^{\xi_0}({}^{\xi}\mathsf{ref}\, t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathbb{c}}) \preceq \xi_0$ by rule B-TYPE

(9) $\mathrm{D}^*;\bullet, y{:}^{\xi_0}({}^{\xi}\mathsf{ref}\, t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathbb{c}}) \vdash \lambda x.\, y\, (\mathsf{new}^{\mathsf{q}}\, x) : {}^{\xi_0}(t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathbb{c}})$ by (7–8), rule T-ABS

(10) $\mathrm{D}^*;\mathrm{G}^*, y{:}^{\xi_0}({}^{\xi}\mathsf{ref}\, t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathbb{c}}) \vdash [\![e]\!]_{\mathbb{c}}\, (\lambda x.\, y\, (\mathsf{new}^{\mathsf{q}}\, x)) : \langle\!\langle \tau', c \rangle\!\rangle^+_{\mathbb{c}}$ by (2, 9), rule T-APP

(11) $\mathrm{D}^*;\mathrm{G}^* \vdash \lambda y.\, [\![e]\!]_{\mathbb{c}}\, (\lambda x.\, y\, (\mathsf{new}^{\mathsf{q}}\, x)) : {}^{\mathsf{L}}(^{\xi_0}({}^{\xi}\mathsf{ref}\, t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathbb{c}}) \multimap \langle\!\langle \tau', c \rangle\!\rangle^+_{\mathbb{c}})$ by (10), rule T-ABS

(12) $\mathrm{D}^*;\mathrm{G}^* \vdash [\![\mathsf{new}^{\mathsf{q}}\, e]\!]_{\mathbb{c}} : {}^{\mathsf{L}}(^{\xi_0}({}^{\xi}\mathsf{ref}\, t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathbb{c}}) \multimap \langle\!\langle \tau', c \rangle\!\rangle^+_{\mathbb{c}})$ by (1, 11).

Case $\dfrac{\mathsf{R} \preceq \mathsf{q} \qquad \mathrm{D};\mathrm{G} \vdash_{\mathbb{c}} e : t ; c}{\mathrm{D};\mathrm{G} \vdash_{\mathbb{c}} \mathsf{new}^{\mathsf{q}}\, e : {}^{\mathsf{q}}\mathsf{ref}\, t ; c}$.

As in the previous case.

Case $\dfrac{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} e : {}^{\xi}\mathsf{ref}\ t\,;c \qquad \mathrm{D} \vdash_{\mathcal{C}} \mathsf{A} \preceq \xi}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} \mathsf{delete}\ e : t\,;c}$.

We want to show that $\mathrm{D}^*;\mathrm{G}^* \vdash [\![\mathsf{delete}\ e]\!]_{\mathcal{C}} : {}^{\mathsf{L}}(^{\xi_0}(t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c \rangle\!\rangle^+_{\mathcal{C}})$.
Then:

(1) $[\![\mathsf{delete}\ e]\!]_{\mathcal{C}} = \lambda y.\, [\![e]\!]_{\mathcal{C}}\, (\lambda x.\, y\, (\mathsf{delete}\ x))$ by def. $[\![e]\!]_{\mathcal{C}}$

(2) $\mathrm{D}^*;\mathrm{G}^* \vdash [\![e]\!]_{\mathcal{C}} : {}^{\mathsf{L}}(^{\xi_0}(^{\xi}\mathsf{ref}\ t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c \rangle\!\rangle^+_{\mathcal{C}})$
by IH

(3) $\mathrm{D}^*;\bullet, y{:}^{\xi_0}(t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}) \vdash y : {}^{\xi_0}(t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}})$
by rule T-VAR

(4) $\mathrm{D}^*;\bullet, x{:}^{\xi}\mathsf{ref}\ t^* \vdash x : {}^{\xi}\mathsf{ref}\ t^*$ by rule T-VAR

(5) $\mathrm{D}^* \vdash \mathsf{A} \preceq \xi$ by lemma C.6

(6) $\mathrm{D}^*;\bullet, x{:}^{\xi}\mathsf{ref}\ t^* \vdash \mathsf{delete}\ x : t^*$ by (4–5), rule T-DELETE

(7) $\mathrm{D}^*;\bullet, x{:}^{\xi}\mathsf{ref}\ t^*, y{:}^{\xi_0}(t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}) \vdash y\,(\mathsf{delete}\ x) : \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}$
by (3, 6), rule T-APP

(8) $\mathrm{D}^* \vdash \bullet, y{:}^{\xi_0}(t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}) \preceq \xi_0$ by rule B-TYPE

(9) $\mathrm{D}^*;\bullet, y{:}^{\xi_0}(t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}) \vdash \lambda x.\, y\,(\mathsf{delete}\ x) : {}^{\xi_0}(^{\xi}\mathsf{ref}\ t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}})$
by (7–8), rule T-ABS

(10) $\mathrm{D}^*;\mathrm{G}^*, y{:}^{\xi_0}(t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}) \vdash [\![e]\!]_{\mathcal{C}}\, (\lambda x.\, y\,(\mathsf{delete}\ x)) : \langle\!\langle \tau', c \rangle\!\rangle^+_{\mathcal{C}}$
by (2, 9), rule T-APP

(11) $\mathrm{D}^*;\mathrm{G}^* \vdash \lambda y.\, [\![e]\!]_{\mathcal{C}}\, (\lambda x.\, y\,(\mathsf{delete}\ x)) : {}^{\mathsf{L}}(^{\xi_0}(t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c \rangle\!\rangle^+_{\mathcal{C}})$
by (10), rule T-ABS

(12) $\mathrm{D}^*;\mathrm{G}^* \vdash [\![\mathsf{delete}\ e]\!]_{\mathcal{C}} : {}^{\mathsf{L}}(^{\xi_0}(t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c \rangle\!\rangle^+_{\mathcal{C}})$
by (1, 11).

Case $\dfrac{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} e : {}^{\xi}\mathsf{ref}\ t\,;c \qquad \mathrm{D} \vdash_{\mathcal{C}} t \preceq \mathsf{R}}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} \mathsf{read}\ e : t\,;c}$.

We want to show that $\mathrm{D}^*;\mathrm{G}^* \vdash [\![\mathsf{read}\ e]\!]_{\mathcal{C}} : {}^{\mathsf{L}}(^{\xi_0}(t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c \rangle\!\rangle^+_{\mathcal{C}})$.
Then:

(1) $[\![\mathsf{read}\ e]\!]_{\mathcal{C}} = \lambda y.\, [\![e]\!]_{\mathcal{C}}\,(\lambda x.\, y\,(\mathsf{read}\ x))$ by def. $[\![e]\!]_{\mathcal{C}}$

(2) $\mathrm{D}^*;\mathrm{G}^* \vdash [\![e]\!]_{\mathcal{C}} : {}^{\mathsf{L}}(^{\xi_0}(^{\xi}\mathsf{ref}\ t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c \rangle\!\rangle^+_{\mathcal{C}})$ by IH

(3) $\mathrm{D}^*;\bullet, y{:}^{\xi_0}(t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}) \vdash y : {}^{\xi_0}(t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}})$ by rule T-VAR

(4) $\mathrm{D}^*;\bullet, x{:}^{\xi}\mathsf{ref}\ t^* \vdash x : {}^{\xi}\mathsf{ref}\ t^*$ by rule T-VAR

(5) $\mathrm{D}^* \vdash t^* \preceq \mathsf{R}$ by lemma C.6

(6) $\mathrm{D}^*;\bullet, x{:}^{\xi}\mathsf{ref}\ t^* \vdash \mathsf{read}\ x : t^*$ by (4–5), rule T-READ

(7) $\mathrm{D}^*;\bullet, x{:}^{\xi}\mathsf{ref}\ t^*, y{:}^{\xi_0}(t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}) \vdash y\,(\mathsf{read}\ x) : \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}$ by (3, 6), rule T-APP

(8) $\mathrm{D}^* \vdash \bullet, y{:}^{\xi_0}(t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}) \preceq \xi_0$ by rule B-TYPE

(9) $\mathrm{D}^*;\bullet, y{:}^{\xi_0}(t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}) \vdash \lambda x.\, y\,(\mathsf{read}\ x) : {}^{\xi_0}(^{\xi}\mathsf{ref}\ t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}})$ by (7–8), rule T-ABS

(10) $\mathrm{D}^*;\mathrm{G}^*, y{:}^{\xi_0}(t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}) \vdash [\![e]\!]_{\mathcal{C}}\,(\lambda x.\, y\,(\mathsf{read}\ x)) : \langle\!\langle \tau', c \rangle\!\rangle^+_{\mathcal{C}}$ by (2, 9), rule T-APP

(11) $\mathrm{D}^*;\mathrm{G}^* \vdash \lambda y.\, [\![e]\!]_{\mathcal{C}}\,(\lambda x.\, y\,(\mathsf{read}\ x)) : {}^{\mathsf{L}}(^{\xi_0}(t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c \rangle\!\rangle^+_{\mathcal{C}})$ by (10), rule T-ABS

(12) $\mathrm{D}^*;\mathrm{G}^* \vdash [\![\mathsf{read}\ e]\!]_{\mathcal{C}} : {}^{\mathsf{L}}(^{\xi_0}(t^* \multimap \langle\!\langle \tau', c \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c \rangle\!\rangle^+_{\mathcal{C}})$ by (1, 11).

Case $\dfrac{\begin{array}{c}\mathrm{D} \vdash_{\mathcal{C}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_2 \qquad \mathrm{D};\mathrm{G}_1 \vdash_{\mathcal{C}} e_1 : {}^{\xi_1}\mathsf{ref}\ t_1\,;c_1' \\ \mathrm{D};\mathrm{G}_2 \vdash_{\mathcal{C}} e_2 : t_2\,;c_2' \qquad \mathrm{D} \vdash_{\mathcal{C}} \mathrm{G}_2 \preceq \xi_2 \qquad \mathrm{D} \vdash_{\mathcal{C}} c_1' \succeq \xi_2 \\ \mathrm{D} \vdash_{\mathcal{C}} c_2' \succeq \xi_1 \qquad \mathrm{D} \vdash_{\mathcal{C}} \mathsf{A} \preceq \xi_1 \qquad \mathrm{D} \vdash_{\mathcal{C}} t_2 \preceq \xi_1 \qquad \mathrm{D} \vdash_{\mathcal{C}} c_1' \otimes c_2' : \mathsf{CTL}\end{array}}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} \mathsf{swap}\ e_1\, e_2 : {}^{\mathsf{L}}(^{\xi}\mathsf{ref}\ t_2 \otimes t_1)\,;c_1' \otimes c_2'}$.

We want to show that

$$\mathrm{D}^*;\mathrm{G}^* \vdash e : {}^{\mathsf{L}}(^{\xi_0}({}^{\mathsf{L}}(^{\xi}\mathsf{ref}\ {t_2}^* \otimes {t_1}^*) \multimap \langle\!\langle \tau', c_1 \otimes c_2 \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c_1 \otimes c_2 \rangle\!\rangle^+_{\mathcal{C}}),$$

where $e = [\![\mathsf{swap}\ e_1\, e_2]\!]_{\mathcal{C}}$. Consider whether the term $\mathsf{swap}\ e_1\, e_2$ has a control effect:

Case $c_1' \oslash c_2' \neq \bot_{\mathcal{C}}$**.**

By property 8.4.2, there exist some $c_1 \neq \bot_{\mathcal{C}}$ and $c_2 \neq \bot_{\mathcal{C}}$ such that

(1) $\mathrm{D} \vdash_{\mathcal{C}} c_1' \preceq c_1$,

(2) $\mathrm{D} \vdash_{\mathcal{C}} c_2' \preceq c_2$,

(3) $c_1 \oslash c_2 = c_1' \oslash c_2'$,and

(4) $\mathrm{D} \vdash_{\mathcal{C}} c_1 \oslash c_2 : \mathsf{CTL}$.

From the antecedent of the lemma to be proved,

(5) $\mathrm{D}^* \vdash \xi_0 \preceq (c_1' \oslash c_2')^*$.

Then:

(6) $[\![\mathsf{swap}\ e_1\ e_2]\!]_{\mathcal{C}} = \lambda y.\, [\![e_1]\!]_{\mathcal{C}}\, (\lambda x_1.\, [\![e_2]\!]_{\mathcal{C}}\, (\lambda x_2.\, y\, (\mathsf{swap}\ x_1\ x_2)))$
by def. $[\![e]\!]_{\mathcal{C}}$

(7) $\exists \tau_1''.\langle\!\langle \tau_1'', c_1' \rangle\!\rangle_{\mathcal{C}}^- = \langle\!\langle \tau', c_1 \rangle\!\rangle_{\mathcal{C}}^-$ and $\langle\!\langle \tau_1'', c_1' \rangle\!\rangle_{\mathcal{C}}^+ = \langle\!\langle \tau', c_1 \rangle\!\rangle_{\mathcal{C}}^+$
by (1), property 8.4.4

(8) $\exists \tau_2''.\langle\!\langle \tau_2'', c_2' \rangle\!\rangle_{\mathcal{C}}^- = \langle\!\langle \tau', c_2 \rangle\!\rangle_{\mathcal{C}}^-$ and $\langle\!\langle \tau_2'', c_2' \rangle\!\rangle_{\mathcal{C}}^+ = \langle\!\langle \tau', c_2 \rangle\!\rangle_{\mathcal{C}}^+$
by (2), property 8.4.4

(9) $\mathrm{D}^*; \mathrm{G}_1{}^* \vdash [\![e_1]\!]_{\mathcal{C}} : {}^{\mathsf{L}}({}^{c_1'{}^*}({}^{\xi_1}\mathsf{ref}\ t_1{}^* \multimap \langle\!\langle \tau', c_1 \rangle\!\rangle_{\mathcal{C}}^-) \multimap \langle\!\langle \tau', c_1 \rangle\!\rangle_{\mathcal{C}}^+)$
by IH(τ_1''), (7)

(10) $\mathrm{D}^*; \mathrm{G}_2{}^* \vdash [\![e_2]\!]_{\mathcal{C}} : {}^{\mathsf{L}}({}^{c_2'{}^*}(t_2{}^* \multimap \langle\!\langle \tau', c_2 \rangle\!\rangle_{\mathcal{C}}^-) \multimap \langle\!\langle \tau', c_2 \rangle\!\rangle_{\mathcal{C}}^+)$
by IH(τ_1''), (8)

(11) $\mathrm{D}^*; \bullet, x_1{:}^{\xi_1}\mathsf{ref}\ t_1{}^* \vdash x_1 : {}^{\xi_1}\mathsf{ref}\ t_1{}^*$ by rule T-VAR

(12) $\mathrm{D}^*; \bullet, x_2{:}t_2{}^* \vdash x_2 : t_2{}^*$ by rule T-VAR

(13) $\mathrm{D}^* \vdash \mathsf{A} \preceq \xi_1$ by lemma C.6

(14) $\mathrm{D}^* \vdash t_2{}^* \preceq \xi_1$ by lemma C.6

(15) $\mathrm{D}^*; \bullet, x_1{:}^{\xi_1}\mathsf{ref}\ t_1{}^*, x_2{:}t_2{}^* \vdash \mathsf{swap}\ x_1\ x_2 : {}^{\mathsf{L}}({}^{\xi}\mathsf{ref}\ t_2{}^* \otimes t_1{}^*)$
by (11–13), (14), rule T-SWAPSTRONG

(16) $\mathrm{D}^*; \bullet, y{:}^{\xi_0}({}^{\mathsf{L}}({}^{\xi}\mathsf{ref}\ t_2{}^* \otimes t_1{}^*) \multimap \langle\!\langle \tau', c_1 \oslash c_2 \rangle\!\rangle_{\mathcal{C}}^-) \vdash y :$
${}^{\xi_0}({}^{\mathsf{L}}({}^{\xi}\mathsf{ref}\ t_2{}^* \otimes t_1{}^*) \multimap \langle\!\langle \tau', c_1 \oslash c_2 \rangle\!\rangle_{\mathcal{C}}^-)$ by rule T-VAR

(17) $\mathrm{D}^*; \bullet, x_1{:}^{\xi_1}\mathsf{ref}\ t_1{}^*, x_2{:}t_2{}^*, y{:}^{\xi_0}({}^{\mathsf{L}}({}^{\xi}\mathsf{ref}\ t_2{}^* \otimes t_1{}^*) \multimap$
$\langle\!\langle \tau', c_1 \oslash c_2 \rangle\!\rangle_{\mathcal{C}}^-) \vdash y\, (\mathsf{swap}\ x_1\ x_2) : \langle\!\langle \tau', c_1 \oslash c_2 \rangle\!\rangle_{\mathcal{C}}^-$
by (15–16), rule T-APP

(18) $\mathrm{D}^* \vdash \xi_1 \preceq {c_2'}^*$ by property 8.4.2

(19) $\mathrm{D}^* \vdash \xi_0 \preceq {c_2'}^*$ by (5), property 3

(20) $\mathrm{D}^* \vdash \bullet, x_1{:}^{\xi_1}\mathsf{ref}\ {t_1}^*, y{:}^{\xi_0}(\mathsf{L}(^{\xi}\mathsf{ref}\ {t_2}^* \otimes {t_1}^*) \multimap \langle\!\langle \tau', c_1 \oslash c_2 \rangle\!\rangle^-_{\mathcal{C}}) \preceq {c_2'}^*$ by rule B-Cons, (18–19)

(21) $\mathrm{D}^*; \bullet, x_1{:}^{\xi_1}\mathsf{ref}\ {t_1}^*, y{:}^{\xi_0}(\mathsf{L}(^{\xi}\mathsf{ref}\ {t_2}^* \otimes {t_1}^*) \multimap \langle\!\langle \tau', c_1 \oslash c_2 \rangle\!\rangle^-_{\mathcal{C}}) \vdash \lambda x_2.\, y(\mathsf{swap}\ x_1\, x_2) : {}^{{c_2'}^*}({t_2}^* \multimap \langle\!\langle \tau', c_1 \oslash c_2 \rangle\!\rangle^-_{\mathcal{C}})$ by (17, 20), rule T-Abs

(22) $\mathrm{D}^*; \bullet, x_1{:}^{\xi_1}\mathsf{ref}\ {t_1}^*, y{:}^{\xi_0}(\mathsf{L}(^{\xi}\mathsf{ref}\ {t_2}^* \otimes {t_1}^*) \multimap \langle\!\langle \tau', c_1 \oslash c_2 \rangle\!\rangle^-_{\mathcal{C}}) \vdash \lambda x_2.\, y(\mathsf{swap}\ x_1\, x_2) : {}^{{c_2'}^*}({t_2}^* \multimap \langle\!\langle \tau', c_2 \rangle\!\rangle^-_{\mathcal{C}})$ by (21), property 8.4.3a

(23) $\mathrm{D}^*; {\mathrm{G}_2}^*, x_1{:}^{\xi_1}\mathsf{ref}\ {t_1}^*, y{:}^{\xi_0}(\mathsf{L}(^{\xi}\mathsf{ref}\ {t_2}^* \otimes {t_1}^*) \multimap \langle\!\langle \tau', c_1 \oslash c_2 \rangle\!\rangle^-_{\mathcal{C}}) \vdash [\![e_2]\!]_{\mathcal{C}}(\lambda x_2.\, y(\mathsf{swap}\ x_1\, x_2)) : \langle\!\langle \tau', c_2 \rangle\!\rangle^+_{\mathcal{C}}$ by (10, 22), rule T-App

(24) $\mathrm{D}^* \vdash \xi_2 \preceq {c_1'}^*$ by property 8.4.2

(25) $\mathrm{D}^* \vdash {\mathrm{G}_2}^* \preceq \xi_2$ by lemma C.6

(26) $\mathrm{D}^* \vdash {\mathrm{G}_2}^* \preceq {c_1'}^*$ by (24–25), ind. ${\mathrm{G}_2}^*$, ...

(27) $\mathrm{D}^* \vdash \xi_0 \preceq {c_1'}^*$ by (5), property 3

(28) $\mathrm{D}^* \vdash {\mathrm{G}_2}^*, y{:}^{\xi_0}(\mathsf{L}(^{\xi}\mathsf{ref}\ {t_2}^* \otimes {t_1}^*) \multimap \langle\!\langle \tau', c_1 \oslash c_2 \rangle\!\rangle^-_{\mathcal{C}}) \preceq {c_1'}^*$ by rule B-Cons, (24, 27), ...

(29) $\mathrm{D}^*; {\mathrm{G}_2}^*, y{:}^{\xi_0}(\mathsf{L}(^{\xi}\mathsf{ref}\ {t_2}^* \otimes {t_1}^*) \multimap \langle\!\langle \tau', c_1 \oslash c_2 \rangle\!\rangle^-_{\mathcal{C}}) \vdash \lambda x_1.\, [\![e_2]\!]_{\mathcal{C}}(\lambda x_2.\, y(\mathsf{swap}\ x_1\, x_2)) : {}^{{c_1'}^*}(^{\xi_1}\mathsf{ref}\ {t_1}^* \multimap \langle\!\langle \tau', c_2 \rangle\!\rangle^+_{\mathcal{C}})$ by (23, 28), rule T-Abs

(30) $\mathrm{D}^*; {\mathrm{G}_2}^*, y{:}^{\xi_0}(\mathsf{L}(^{\xi}\mathsf{ref}\ {t_2}^* \otimes {t_1}^*) \multimap \langle\!\langle \tau', c_1 \oslash c_2 \rangle\!\rangle^-_{\mathcal{C}}) \vdash \lambda x_1.\, [\![e_2]\!]_{\mathcal{C}}(\lambda x_2.\, y(\mathsf{swap}\ x_1\, x_2)) : {}^{{c_1'}^*}(^{\xi_1}\mathsf{ref}\ {t_1}^* \multimap \langle\!\langle \tau', c_1 \rangle\!\rangle^-_{\mathcal{C}})$ by (29), property 8.4.3c

(31) $\mathrm{D}^*; \mathrm{G}^*, y{:}^{\xi_0}(\mathsf{L}(^{\xi}\mathsf{ref}\ {t_2}^* \otimes {t_1}^*) \multimap \langle\!\langle \tau', c_1 \oslash c_2 \rangle\!\rangle^-_{\mathcal{C}}) \vdash [\![e_1]\!]_{\mathcal{C}}(\lambda x_1.\, [\![e_2]\!]_{\mathcal{C}}(\lambda x_2.\, y(\mathsf{swap}\ x_1\, x_2))) : \langle\!\langle \tau', c_1 \rangle\!\rangle^+_{\mathcal{C}}$ by (9, 30), rule T-App

(32) $\mathrm{D}^*;\mathrm{G}^* \vdash \lambda y.\, [\![e_1]\!]_{\mathcal{C}}\,(\lambda x_1.\, [\![e_2]\!]_{\mathcal{C}}\,(\lambda x_2.\, y\,(\mathsf{swap}\; x_1\, x_2))) :$
$\mathsf{L}^{(\xi_0}(\mathsf{L}^{(\xi}\mathsf{ref}\; t_2{}^* \otimes t_1{}^*) \multimap \langle\!\langle \tau', c_1 \oslash c_2 \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c_1 \rangle\!\rangle^+_{\mathcal{C}})$
by (31), rule T-ABS

(33) $\mathrm{D}^*;\mathrm{G}^* \vdash [\![\mathsf{swap}\; e_1\, e_2]\!]_{\mathcal{C}} : \mathsf{L}^{(\xi_0}(\mathsf{L}^{(\xi}\mathsf{ref}\; t_2{}^* \otimes t_1{}^*) \multimap$
$\langle\!\langle \tau', c_1 \oslash c_2 \rangle\!\rangle^-_{\mathcal{C}}) \multimap \langle\!\langle \tau', c_1 \oslash c_2 \rangle\!\rangle^+_{\mathcal{C}})$ by (32), property 8.4.3b, (6).

Case $c_1' \oslash c_2' = \bot_{\mathcal{C}}$**.**

As in the previous case, but with all the control effect manipulation elided, because all the control effects are pure. This follows from the impure case just as the pure cases for application and type application from their impure cases.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{C}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_2 \qquad \mathrm{D};\mathrm{G}_1 \vdash_{\mathcal{C}} e_1 : {}^{\xi_1}\mathsf{ref}\; t\,;c_1 \qquad \mathrm{D};\mathrm{G}_2 \vdash_{\mathcal{C}} e_2 : t\,;c_2 \qquad \mathrm{D} \vdash_{\mathcal{C}} \mathrm{G}_2 \preceq \xi_2 \qquad \mathrm{D} \vdash_{\mathcal{C}} c_1 \succeq \xi_2 \qquad \mathrm{D} \vdash_{\mathcal{C}} c_2 \succeq \xi_1 \qquad \mathrm{D} \vdash_{\mathcal{C}} c_1 \oslash c_2 : \mathsf{CTL}}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} \mathsf{swap}\; e_1\, e_2 : \mathsf{L}^{(\xi}\mathsf{ref}\; t \otimes t)\,;c_1 \oslash c_2}$**.**

As in the previous case. □

COROLLARY 8.9 (Translation of program typing, restated from p. 230)**.**

If $\mathrm{D};\mathrm{G} \vdash_{\mathcal{C}} e : t\,;\bot_{\mathcal{C}}$ *where* $\mathrm{D} \vdash_{\mathcal{C}} t \preceq \mathsf{A}$*, then*

$$\mathrm{D}^*;\mathrm{G}^* \vdash [\![e]\!]_{\mathcal{C}}\, \mathsf{done}_{\mathcal{C}} : \langle\!\langle t^* \rangle\!\rangle_{\mathcal{C}}.$$

Proof. Then:

(1) $\mathrm{D}^* \vdash t^* \preceq \mathsf{A}$ by lemma C.6

(2) $\mathrm{D}^* \vdash t^* : \star$ by lemma C.5, lemma 8.5

(3) $\mathrm{D} \vdash_{\mathcal{C}} \mathsf{L} \preceq \bot_{\mathcal{C}}{}^*$ by rule QSUB-REFL

(4) $\mathrm{D}^*;\mathrm{G}^* \vdash [\![e]\!]_{\mathcal{C}} : \mathsf{L}^{(\mathsf{L}^{(}t^* \multimap \langle\!\langle t^* \rangle\!\rangle_{\mathcal{C}}) \multimap \langle\!\langle t^* \rangle\!\rangle_{\mathcal{C}})$ by prem., (2–3), lemma 8.8

(5) $\mathrm{D}^*;\bullet \vdash \mathsf{done}_{\mathcal{C}} : \mathsf{L}^{(}t^* \multimap \langle\!\langle t^* \rangle\!\rangle_{\mathcal{C}})$ by (1–2), property 2

(6) $\mathrm{D}^* \vdash \mathrm{G}^* \rightsquigarrow \mathrm{G}^* \boxplus \bullet$ by ind. G^*, rule S-CONSL

(7) $\mathrm{D}^*;\mathrm{G}^* \vdash [\![e]\!]_{\mathcal{C}}\,\mathsf{done}_{\mathcal{C}} : \langle\!\langle t^* \rangle\!\rangle_{\mathcal{C}}$ by (4–6), rule T-APP. □

C.3 Proofs for Example Control Effects

In this section, I prove that each of the control effects in §8.5 meets the control effect parameter soundness criteria.

C.3.1 Delimited Continuation Effects

In this section, we consider the delimited continuation effects from §8.5.1.

LEMMA C.7 (Top)**.**

If $d^* = \mathsf{U}$ *then* $(d \sqcup d')^* = \mathsf{U}$.

Proof. Assuming $d^* = \mathsf{U}$, we proceed by cases on $d \sqcup d'$:

Case α**.**

Then $(d \sqcup d')^* = \alpha$. By the quotienting of d, $d \sqcup d' = \alpha$ only if d is α or $\bot_{\mathcal{D}}$. If $d = \alpha$ then $d^* = \alpha$, which contradicts our assumption. If $d = \bot_{\mathcal{D}}$ then $d^* = \mathsf{L}$, which also contradicts our assumption.

Case $\bot_{\mathcal{D}}$**.**

Then $(d \sqcup d')^* = \mathsf{L}$. By the quotienting of d, $d \sqcup d' = \bot_{\mathcal{D}}$ only if $d = \bot_{\mathcal{D}}$, which means that $d^* = \mathsf{L}$, which contradicts our assumption.

Case $\overline{\xi}$**.**

Then $(d \sqcup d')^* = \xi$. By the quotienting of d, we can have $d \sqcup d' = \overline{\xi}$ in one of two ways:

Case $d = \overline{\xi_1}$ and $d' = \overline{\xi_2}$ and $\xi_1 \sqcap \xi_2 = \xi$**.**

That is, $(d \sqcup d')^* = \xi_1 \sqcap \xi_2$ and $d^* = \xi_1$. Then $\xi_1 = \mathsf{U}$, which means that $(d \sqcup d')^* = \xi_1 \sqcap \xi_2 = \mathsf{U} \sqcap \xi_2 = \mathsf{U}$.

Case $d = d' = \overline{\xi}$**.**

This is subsumed by the previous case.

Otherwise.

Then $(d \sqcup d')^* = \mathsf{U}$, which is the desired conclusion. □

THEOREM 8.13 (Delimited continuation properties, restated from p. 234)**.**

Delimited continuation effects $(\mathcal{D}, \bot_{\mathcal{D}}, \sqcup)$ *satisfy properties 1–5.*

Proof.

Property 1 (Answer types). All properties here are trivial because $\langle\!\langle \tau, d \rangle\!\rangle^-_{\mathcal{D}} = \langle\!\langle \tau, d \rangle\!\rangle^+_{\mathcal{D}} = {}^{\mathsf{U}}1$.

Property 2 (Done). Then:

(1) $\Delta; \bullet \vdash \langle\rangle : {}^{\mathsf{U}}1$	by rules T-UNIT and K-QUAL
(2) $\Delta \vdash \bullet, x{:}\tau \preceq \mathsf{A}$	by rules B-NIL and B-CONS
(3) $\Delta \vdash \bullet, x{:}\tau \rightsquigarrow \bullet \boxplus \bullet, x{:}\tau$	by rules S-NIL and S-CONSR
(4) $\Delta; \bullet, x{:}\tau \vdash \langle\rangle : {}^{\mathsf{U}}1$	by (1–3), rule T-WEAK
(5) $\Delta; \bullet \vdash \lambda x.\langle\rangle : {}^{\mathsf{L}}(\tau \multimap {}^{\mathsf{U}}1)$	by (4), rule T-ABS
(6) $\langle\!\langle \tau \rangle\!\rangle_{\mathcal{D}} = {}^{\mathsf{U}}1$	by def. $\langle\!\langle \tau \rangle\!\rangle_{\mathcal{D}}$
(7) $\Delta; \bullet \vdash \lambda x.\langle\rangle : {}^{\mathsf{L}}(\tau \multimap \langle\!\langle \tau \rangle\!\rangle_{\mathcal{D}})$	by (5–6).

Property 3 (Effect sequencing). Let $\mathrm{D} \vdash_{\mathcal{E}} d_1 \sqcup d_2 : \mathsf{CTL}$. We must show that $\mathrm{D}^* \vdash (d_1 \sqcup d_2)^* \preceq {d_1}^*$ and $\mathrm{D}^* \vdash (d_1 \sqcup d_2)^* \preceq {d_2}^*$. By symmetry, it suffices to show the former. Then:

(1) $\mathrm{D} \vdash_{\mathcal{D}} d_1 \preceq d_1$	by rule CSUB-TRANS
(2) $\mathrm{D} \vdash_{\mathcal{D}} \bot_{\mathcal{D}} \preceq d_2$	by rule DSUB-BOT
(3) $\mathrm{D} \vdash_{\mathcal{D}} d_1 \sqcup \bot_{\mathcal{D}} \preceq d_1 \sqcup d_2$	by (1–2), rule DSUB-JOIN
(4) $\mathrm{D} \vdash_{\mathcal{D}} d_1 \preceq d_1 \sqcup d_2$	by (3), quotient

(5) $\mathrm{D}^* \vdash (d_1 \sqcup d_2)^* \preceq {d_1}^*$ by lemma 8.7.

Property 4 (Bottom and lifting).

1. To show that $d_1 \sqcup d_2 = \bot_{\mathcal{D}}$ if and only if $d_1 = d_2 = \bot_{\mathcal{D}}$, we consider the quotienting of $\mathcal{D}$.

2. We must also show that if $\mathrm{D} \vdash_{\mathcal{D}} d_1 \sqcup d_2 : \mathsf{CTL}$ and $d_1 \sqcup d_2 \neq \bot_{\mathcal{D}}$, then there exist some $d_1' \neq \bot_{\mathcal{D}}$ and $d_2' \neq \bot_{\mathcal{D}}$ with particular properties. For each ${d_i}^{(i \in \{1,2\})}$, if $d_i = \bot_{\mathcal{D}}$ then let $d_i' = \overline{\mathsf{L}}$; otherwise, let $d_i' = d_i$. This ensures that 1–2) each $\mathrm{D} \vdash_{\mathcal{D}} d_i \preceq d_i'$, 3) $d_1 \sqcup d_2 = d_1' \sqcup d_2'$, and 4) $d_1' \sqcup d_2'$ is well formed.

Property 5 (New rules).

1. For translation of effect bounds, let $\mathrm{D} \vdash_{\mathcal{D}} d \succeq \xi$; we need to show that $\mathrm{D}^* \vdash \xi \preceq d^*$. We proceed by induction on the derivation of $\mathrm{D} \vdash_{\mathcal{D}} d \succeq \xi$, which has two new cases to consider:

   Case $\dfrac{\mathrm{D} \vdash_{\mathcal{C}} \xi \preceq \xi'}{\mathrm{D} \vdash_{\mathcal{D}} \overline{\xi'} \succeq \xi}$.

   Then $\overline{\xi'}^* = \xi'$, and by lemma C.6, $\mathrm{D}^* \vdash \xi \preceq \xi'$.

   Case $\dfrac{\mathrm{D} \vdash_{\mathcal{D}} d_1 \succeq \xi \qquad \mathrm{D} \vdash_{\mathcal{D}} d_2 \succeq \xi}{\mathrm{D} \vdash_{\mathcal{D}} d_1 \sqcup d_2 \succeq \xi}$.

   By the induction hypothesis,

   (1) $\mathrm{D}^* \vdash \xi \preceq {d_1}^*$ and

   (2) $\mathrm{D}^* \vdash \xi \preceq {d_2}^*$.

   Now we consider several possibilities for $d_1 \sqcup d_2$ in light of the quotienting of $\mathcal{D}$:

   - If $d_1 = d_2$ then $d_1 \sqcup d_2 = d_1$, so we have $\mathrm{D}^* \vdash \xi \preceq (d_1 \sqcup d_2)^*$ by (1).
   - If $d_1 = \bot_{\mathcal{D}}$, then $d_1 \sqcup d_2 = d_2$, and thus $\mathrm{D}^* \vdash \xi \preceq (d_1 \sqcup d_2)^*$ by (2).
   - By symmetry with the previous case if $d_2 = \bot_{\mathcal{D}}$.

- If $d_1 = \overline{\xi_1}$ and $d_2 = \overline{\xi_2}$ where $\xi_1 \sqcap \xi_2$ is defined, then

  (3) $d_1 \sqcup d_2 = \overline{(\xi_1 \sqcap \xi_2)}$.

  Then:

  (4) $\mathrm{D}^* \vdash \xi \preceq \xi_1$ by (1), def. $\overline{\xi_1}^*$

  (5) $\mathrm{D}^* \vdash \xi \preceq \xi_2$ by (2), def. $\overline{\xi_2}^*$

  (6) $\mathrm{D}^* \vdash \xi \preceq \xi_1 \sqcap \xi_2$ by (4–5), lemma C.2

  (7) $\mathrm{D}^* \vdash \xi \preceq (d_1 \sqcup d_2)^*$ by (3, 6), def. $\overline{\xi}^*$

- If $d_1 = \overline{\xi_1}$ and $d_2 = \overline{\xi_2}$ where $\xi_1 \sqcap \xi_2$ is not defined, then by lemma C.3 and (1–2), $\xi = \mathsf{U}$. Then by rule QSUB-BOT.
- If $d_1 = \alpha$, then since $\mathrm{D} \vdash_{\mathcal{D}} \alpha \succeq \xi$, we know by inversion that $\xi = \mathsf{U}$. Then by rule QSUB-BOT.
- If $d_2 = \alpha$, then by symmetry with the previous case.

2. For translation of effect subsumption, let $\mathrm{D} \vdash_{\mathcal{D}} d_1 \preceq d_2$; we must show that $\mathrm{D}^* \vdash {d_2}^* \preceq {d_1}^*$. We proceed by induction on the derivation of $\mathrm{D} \vdash_{\mathcal{D}} d_1 \preceq d_2$, which has several cases to consider:

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{D}} d : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{D}} \bot_{\mathcal{D}} \preceq d}$.

Then $\mathrm{D}^* \vdash d^* \preceq \mathsf{L}$ by rule QSUB-TOP.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{C}} \xi : \mathsf{QUAL}}{\mathrm{D} \vdash_{\mathcal{D}} \overline{\mathsf{L}} \preceq \overline{\xi}}$.

Then $\mathrm{D}^* \vdash \xi \preceq \mathsf{L}$ by rule QSUB-TOP.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{D}} d : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{D}} d \preceq \overline{\mathsf{U}}}$.

Then $\mathrm{D}^* \vdash \mathsf{U} \preceq d^*$ by rule QSUB-BOT.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{D}} d_1 \preceq d_1' \qquad \mathrm{D} \vdash_{\mathcal{D}} d_2 \preceq d_2' \qquad \mathrm{D} \vdash_{\mathcal{D}} d_1 \sqcup d_2 : \mathsf{CTL} \qquad \mathrm{D} \vdash_{\mathcal{D}} d_1' \sqcup d_2' : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{D}} d_1 \sqcup d_2 \preceq d_1' \sqcup d_2'}$.

Without loss of generality, let

(1) $\mathrm{D} = \bullet, \alpha{:}\mathsf{QUAL}, \beta{:}\mathsf{QUAL}, \alpha'{:}\mathsf{CTL}, \beta'{:}\mathsf{CTL}$.

(i) \ (ii)	$\bot_{\mathcal{D}} \preceq d$	$\alpha' \preceq \alpha'$	$\overline{\mathsf{R}} \preceq \overline{\mathsf{R}}$	$\overline{\mathsf{A}} \preceq \overline{\mathsf{A}}$	$\overline{\mathsf{L}} \preceq \overline{\mathsf{R}}$	$\overline{\mathsf{L}} \preceq \overline{\mathsf{A}}$	$\overline{\mathsf{L}} \preceq \overline{\mathsf{L}}$	$\overline{\mathsf{L}} \preceq \overline{\alpha}$	$\overline{\alpha} \preceq \overline{\alpha}$	$\overline{\beta} \preceq \overline{\beta}$	$d \preceq \overline{\mathsf{U}}$
$\bot_{\mathcal{D}} \preceq d$	$\mathsf{U} \preceq \mathsf{L}$	$\mathsf{U} \preceq \alpha'$	$\mathsf{U} \preceq \mathsf{R}$	$\mathsf{U} \preceq \mathsf{A}$	$\mathsf{U} \preceq \mathsf{L}$	$\mathsf{U} \preceq \mathsf{L}$	$\mathsf{U} \preceq \mathsf{L}$	$\mathsf{U} \preceq \mathsf{L}$	$\mathsf{U} \preceq \alpha$	$\mathsf{U} \preceq \beta$	$\mathsf{U} \preceq \mathsf{U}$
$\alpha' \preceq \alpha'$	$\mathsf{U} \preceq \alpha'$	$\alpha' \preceq \alpha'$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$
$\beta' \preceq \beta'$	$\mathsf{U} \preceq \beta'$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$
$\overline{\mathsf{R}} \preceq \overline{\mathsf{R}}$	$\mathsf{U} \preceq \mathsf{R}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{R} \preceq \mathsf{R}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{R} \preceq \mathsf{R}$	$\mathsf{U} \preceq \mathsf{R}$	$\mathsf{R} \preceq \mathsf{R}$	$\mathsf{U} \preceq \mathsf{R}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$
$\overline{\mathsf{A}} \preceq \overline{\mathsf{A}}$	$\mathsf{U} \preceq \mathsf{A}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{A} \preceq \mathsf{A}$	$\mathsf{U} \preceq \mathsf{A}$	$\mathsf{A} \preceq \mathsf{A}$	$\mathsf{A} \preceq \mathsf{A}$	$\mathsf{U} \preceq \mathsf{A}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$
$\overline{\mathsf{L}} \preceq \overline{\mathsf{R}}$	$\mathsf{U} \preceq \mathsf{L}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{R} \preceq \mathsf{R}$	$\mathsf{U} \preceq \mathsf{A}$	$\mathsf{R} \preceq \mathsf{L}$	$\mathsf{U} \preceq \mathsf{L}$	$\mathsf{R} \preceq \mathsf{L}$	$\mathsf{U} \preceq \mathsf{L}$	$\mathsf{U} \preceq \alpha$	$\mathsf{U} \preceq \beta$	$\mathsf{U} \preceq \mathsf{U}$
$\overline{\mathsf{L}} \preceq \overline{\mathsf{A}}$	$\mathsf{U} \preceq \mathsf{L}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{R}$	$\mathsf{A} \preceq \mathsf{A}$	$\mathsf{U} \preceq \mathsf{L}$	$\mathsf{A} \preceq \mathsf{L}$	$\mathsf{A} \preceq \mathsf{L}$	$\mathsf{U} \preceq \mathsf{L}$	$\mathsf{U} \preceq \alpha$	$\mathsf{U} \preceq \beta$	$\mathsf{U} \preceq \mathsf{U}$
$\overline{\mathsf{L}} \preceq \overline{\mathsf{L}}$	$\mathsf{U} \preceq \mathsf{L}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{R} \preceq \mathsf{R}$	$\mathsf{A} \preceq \mathsf{A}$	$\mathsf{R} \preceq \mathsf{L}$	$\mathsf{A} \preceq \mathsf{L}$	$\mathsf{L} \preceq \mathsf{L}$	$\alpha \preceq \mathsf{L}$	$\alpha \preceq \alpha$	$\beta \preceq \beta$	$\mathsf{U} \preceq \mathsf{U}$
$\overline{\mathsf{L}} \preceq \overline{\alpha}$	$\mathsf{U} \preceq \mathsf{L}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{R}$	$\mathsf{U} \preceq \mathsf{A}$	$\mathsf{U} \preceq \mathsf{L}$	$\mathsf{U} \preceq \mathsf{L}$	$\alpha \preceq \mathsf{L}$	$\alpha \preceq \mathsf{L}$	$\alpha \preceq \alpha$	$\mathsf{U} \preceq \beta$	$\mathsf{U} \preceq \mathsf{U}$
$\overline{\mathsf{L}} \preceq \overline{\beta}$	$\mathsf{U} \preceq \mathsf{L}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{R}$	$\mathsf{U} \preceq \mathsf{A}$	$\mathsf{U} \preceq \mathsf{L}$	$\mathsf{U} \preceq \mathsf{L}$	$\beta \preceq \mathsf{L}$	$\mathsf{U} \preceq \mathsf{L}$	$\mathsf{U} \preceq \alpha$	$\beta \preceq \beta$	$\mathsf{U} \preceq \mathsf{U}$
$\overline{\alpha} \preceq \overline{\alpha}$	$\mathsf{U} \preceq \alpha$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \alpha$	$\mathsf{U} \preceq \alpha$	$\alpha \preceq \alpha$	$\alpha \preceq \alpha$	$\alpha \preceq \alpha$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$
$\overline{\beta} \preceq \overline{\beta}$	$\mathsf{U} \preceq \beta$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \beta$	$\mathsf{U} \preceq \beta$	$\beta \preceq \beta$	$\mathsf{U} \preceq \beta$	$\mathsf{U} \preceq \mathsf{U}$	$\beta \preceq \beta$	$\mathsf{U} \preceq \mathsf{U}$
$d \preceq \overline{\mathsf{U}}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$	$\mathsf{U} \preceq \mathsf{U}$

Figure C.1: Exhaustive proof for final case in translation subsumption

By cases on the possibilities for d_1, d_2, d_1', and d_2', we construct a table showing all the cases in figure C.1. We label the rows with instances of the judgment (i) $\mathrm{D} \vdash_{\mathcal{D}} d_1 \preceq d_1'$ and the columns with instances of the judgment (ii) $\mathrm{D} \vdash_{\mathcal{D}} d_2 \preceq d_2'$. Using lemma C.7, we fill the cells with instances of $\mathrm{D}^* \vdash (d_1' \sqcup d_2')^* \preceq (d_1 \sqcup d_2)^*$.

3. For translation of kinding, let $\mathrm{D} \vdash_{\mathcal{C}} d : \mathsf{CTL}$. We must show that $\mathrm{D}^* \vdash d^* : \mathsf{QUAL}$. We proceed by induction on the derivation, considering the two new kinding rules:

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{C}} \xi : \mathsf{QUAL}}{\mathrm{D} \vdash_{\mathcal{D}} \overline{\xi} : \mathsf{CTL}}$.

By the induction hypothesis, $\mathrm{D}^* \vdash \xi : \mathsf{QUAL}$, noting that $(\overline{\xi})^* = \xi$ and $\mathsf{CTL}^* = \mathsf{QUAL}$.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{C}} d_1 : \mathsf{CTL} \qquad \mathrm{D} \vdash_{\mathcal{C}} d_2 : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{D}} d_1 \sqcup d_2 : \mathsf{CTL}}$.

By cases on $d_1 \sqcup d_2$:

Case α**.**

Then $(d_1 \sqcup d_2)^* = \alpha$. By the quotienting of d, $d_1 \sqcup d_2 = \alpha$ only if at least one of d_1 or d_2 is α. From the premises, we know that

$\mathrm{D} \vdash_{\mathcal{C}} \alpha : \mathsf{CTL}$, which means that $\alpha{:}\mathsf{CTL} \in \mathrm{D}$. This means that $\alpha{:}\mathsf{QUAL} \in \mathrm{D}^*$, so $\mathrm{D}^* \vdash \alpha : \mathsf{QUAL}$.

Case $\bot_{\mathcal{D}}$.

Then $(d_1 \sqcup d_2)^* = \mathsf{L}$, so $\mathrm{D}^* \vdash \mathsf{L} : \mathsf{QUAL}$ by rule K-QUAL.

Case $\overline{\xi}$.

Then $(d_1 \sqcup d_2)^* = \xi$. By the quotienting of d, $d_1 \sqcup d_2 = \overline{\xi}$ in one of two ways:

Case $d_1 = \overline{\xi_1}$ and $d_2 = \overline{\xi_2}$ and $\xi_1 \sqcap \xi_2 = \xi$.

That is,

(1) $(d_1 \sqcup d_2)^* = \xi_1 \sqcap \xi_2$,

(2) ${d_1}^* = \xi_1$, and

(3) ${d_2}^* = \xi_2$.

By the induction hypothesis, twice, we have that

(4) $\mathrm{D}^* \vdash {d_1}^* : \mathsf{QUAL}$ and

(5) $\mathrm{D}^* \vdash {d_2}^* : \mathsf{QUAL}$,

and by substitution (2–3).

(6) $\mathrm{D}^* \vdash \xi_1 : \mathsf{QUAL}$ and

(7) $\mathrm{D}^* \vdash \xi_2 : \mathsf{QUAL}$,

Since $\xi_1 \sqcap \xi_2 = \xi$, the meet is defined. By the definition of meet, $\xi_1 \sqcap \xi_2$ is either a constant qualifier $\mathfrak{q}$, which has a kind by rule K-QUAL, or it is a identical to at least one of ξ_1 or ξ_2, both of which are well-kinded under D^*.

Case $d_1 = d_2 = \overline{\xi}$.

This is subsumed by the previous case.

Otherwise.

Then $(d_1 \sqcup d_2)^* = \mathsf{U}$, so $\mathrm{D}^* \vdash \mathsf{U} : \mathsf{QUAL}$ by rule K-QUAL.

4. For translation of typing, let

- $\mathrm{D}; \mathrm{G} \vdash_{\mathcal{D}} e : t \,; d$.
- $\mathrm{D}^* \vdash \xi_0 \preceq d^*$, and
- $\mathrm{D}^* \vdash \tau' : \star$.

We must show that $\mathrm{D}^*;\mathrm{G}^* \vdash [\![e]\!]_{\mathcal{D}} : {}^{\mathsf{L}}(^{\xi_0}(t^* \multimap {}^{\mathsf{U}}1) \multimap {}^{\mathsf{U}}1)$. We proceed by induction on the typing derivation, with two cases to consider:

Case $\dfrac{\mathrm{D};\mathrm{G} \vdash_{\mathcal{D}} e' : {}^{\mathsf{U}}1 ; d'}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{D}} \mathsf{reset}\, e' : {}^{\mathsf{U}}1 ; \bot_{\mathcal{D}}}$.

We must show that $\mathrm{D}^*;\mathrm{G}^* \vdash [\![e']\!]_{\mathcal{D}} : {}^{\mathsf{L}}(^{\xi_0}({}^{\mathsf{U}}1 \multimap {}^{\mathsf{U}}1) \multimap {}^{\mathsf{U}}1)$. Then,

(1) $[\![\mathsf{reset}\, e']\!]_{\mathcal{D}} = \lambda y.\, y([\![e']\!]_{\mathcal{D}}(\lambda x.x))$ by def. $[\![e]\!]_{\mathcal{D}}$

(2) $\mathrm{D}^*;\mathrm{G}^* \vdash [\![e']\!]_{\mathcal{D}} : {}^{\mathsf{L}}(^{d'^*}({}^{\mathsf{U}}1 \multimap {}^{\mathsf{U}}1) \multimap {}^{\mathsf{U}}1)$ by IH

(3) $\mathrm{D}^*;\bullet, x{:}{}^{\mathsf{U}}1 \vdash x : {}^{\mathsf{U}}1$ by rule T-Var

(4) $\mathrm{D}^*;\bullet \vdash \lambda x.x : {}^{d'^*}({}^{\mathsf{U}}1 \multimap {}^{\mathsf{U}}1)$ by (3), rule T-Abs

(5) $\mathrm{D}^*;\mathrm{G}^* \vdash [\![e']\!]_{\mathcal{D}}(\lambda x.x) : {}^{\mathsf{U}}1$ by (2, 4), rule T-App

(6) $\mathrm{D}^*;\bullet, y{:}{}^{\xi_0}({}^{\mathsf{U}}1 \multimap {}^{\mathsf{U}}1) \vdash y : {}^{\xi_0}({}^{\mathsf{U}}1 \multimap {}^{\mathsf{U}}1)$ by rule T-Var

(7) $\mathrm{D}^*;\mathrm{G}^*, y{:}{}^{\xi_0}({}^{\mathsf{U}}1 \multimap {}^{\mathsf{U}}1) \vdash y([\![e']\!]_{\mathcal{D}}(\lambda x.x)) : {}^{\mathsf{U}}1$ by (5–6), rule T-App

(8) $\mathrm{D}^*;\mathrm{G}^* \vdash \lambda y.\, y([\![e']\!]_{\mathcal{D}}(\lambda x.x)) : {}^{\mathsf{L}}(^{\xi_0}({}^{\mathsf{U}}1 \multimap {}^{\mathsf{U}}1) \multimap {}^{\mathsf{U}}1)$ by (7),rule T-Abs

(9) $\mathrm{D}^*;\mathrm{G}^* \vdash [\![\mathsf{reset}\, e']\!]_{\mathcal{D}} : {}^{\mathsf{L}}(^{\xi_0}({}^{\mathsf{U}}1 \multimap {}^{\mathsf{U}}1) \multimap {}^{\mathsf{U}}1)$ by (1, 8).

Case $\dfrac{\mathrm{D};\mathrm{G}, y'{:}{}^{\xi}(t \xrightarrow{\bot_{\mathcal{D}}} {}^{\mathsf{U}}1) \vdash_{\mathcal{D}} e' : {}^{\mathsf{U}}1 ; d'}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{D}} \mathsf{shift}\, y' \,\mathsf{in}\, e' : t ; d' \sqcup \overline{\xi}}$.

We must show that $\mathrm{D}^*;\mathrm{G}^* \vdash [\![\mathsf{shift}\, y' \,\mathsf{in}\, e']\!]_{\mathcal{D}} : {}^{\mathsf{L}}(^{\xi_0}(t^* \multimap {}^{\mathsf{U}}1) \multimap {}^{\mathsf{U}}1)$. By our premises, we know that

(1) $\mathrm{D}^* \vdash \xi_0 \preceq (d' \sqcup \overline{\xi})^*$

Now, consider whether $d'^* \sqcap \xi$ is defined: If so, then $\mathrm{D}^* \vdash \xi_0 \preceq d'^* \sqcap \xi$, which means that $\mathrm{D}^* \vdash \xi_0 \preceq \xi$ by lemma C.2. If the meet is not defined, then $(d' \sqcup \overline{\xi})^* = \mathsf{U}$, which means that $\mathrm{D}^* \vdash \xi_0 \preceq \mathsf{U}$. In either case, we have that

(2) $\mathrm{D}^* \vdash \xi_0 \preceq \xi$.

Then,

(3) $[\![\mathsf{shift}\, y' \,\mathsf{in}\, e']\!]_{\mathcal{D}} = \lambda y.(\lambda y'.[\![e']\!]_{\mathcal{D}}(\lambda x.x))(\Lambda.\lambda x.\lambda y''.y''(yx))$ by def. $[\![e]\!]_{\mathcal{D}}$

(4) $\mathrm{D}^*;\mathrm{G}^*, y':(^{\xi}(t \xrightarrow{\perp_{\mathcal{D}}}_{\circ} {}^{\cup}1))^* \vdash [\![e']\!]_{\mathcal{D}} : {}^{\mathsf{L}}({}^{d'^*}({}^{\cup}1 \multimap {}^{\cup}1) \multimap {}^{\cup}1)$ by IH

(5) $\mathrm{D}^*;\bullet \vdash \lambda x.x : {}^{d'^*}({}^{\cup}1 \multimap {}^{\cup}1)$ by rules T-Var and T-Abs

(6) $\mathrm{D}^*;\mathrm{G}^*, y':(^{\xi}(t \xrightarrow{\perp_{\mathcal{D}}}_{\circ} {}^{\cup}1))^* \vdash [\![e']\!]_{\mathcal{D}}(\lambda x.x) : {}^{\cup}1$ by (4–5), rule T-App

(7) $\mathrm{D}^*;\mathrm{G}^* \vdash \lambda y'.[\![e']\!]_{\mathcal{D}}(\lambda x.x) : {}^{\mathsf{L}}((^{\xi}(t \xrightarrow{\perp_{\mathcal{D}}}_{\circ} {}^{\cup}1))^* \multimap {}^{\cup}1)$ by (6), rule T-Abs

(8) $\mathrm{D}^*, \alpha{:}\star;\bullet, y'':{}^{\mathsf{L}}({}^{\cup}1 \multimap {}^{\cup}1) \vdash y'' : {}^{\mathsf{L}}({}^{\cup}1 \multimap {}^{\cup}1)$ by rule T-Var

(9) $\mathrm{D}^*, \alpha{:}\star;\bullet, y:{}^{\xi_0}(t^* \multimap {}^{\cup}1) \vdash y : {}^{\xi_0}(t^* \multimap {}^{\cup}1)$ by rule T-Var

(10) $\mathrm{D}^*, \alpha{:}\star;\bullet, x{:}t^* \vdash x : t^*$ by rule T-Var

(11) $\mathrm{D}^*, \alpha{:}\star;\bullet, y:{}^{\xi_0}(t^* \multimap {}^{\cup}1), x{:}t^* \vdash yx : {}^{\cup}1$ by (9–10), rule T-App

(12) $\mathrm{D}^*, \alpha{:}\star;\bullet, y:{}^{\xi_0}(t^* \multimap {}^{\cup}1), x{:}t^*, y'':{}^{\mathsf{L}}({}^{\cup}1 \multimap {}^{\cup}1) \vdash y''(yx) : {}^{\cup}1$ by (8, 11), rule T-App

(13) $\mathrm{D}^*, \alpha{:}\star;\bullet, y:{}^{\xi_0}(t^* \multimap {}^{\cup}1), x{:}t^* \vdash \lambda y''.y''(yx) : {}^{\mathsf{L}}({}^{\mathsf{L}}({}^{\cup}1 \multimap {}^{\cup}1) \multimap {}^{\cup}1)$ by (12), rule T-Abs

(14) $\mathrm{D}^*, \alpha{:}\star;\bullet, y:{}^{\xi_0}(t^* \multimap {}^{\cup}1) \vdash \lambda x.\lambda y''.y''(yx) : {}^{\mathsf{L}}(t^* \multimap {}^{\mathsf{L}}({}^{\mathsf{L}}({}^{\cup}1 \multimap {}^{\cup}1) \multimap {}^{\cup}1))$ by (13), rule T-Abs

(15) $\mathrm{D}^* \vdash \bullet, y:{}^{\xi_0}(t^* \multimap {}^{\cup}1) \preceq \xi$ by (2)

(16) $\mathrm{D}^*;\bullet, y:{}^{\xi_0}(t^* \multimap {}^{\cup}1) \vdash \Lambda.\lambda x.\lambda y''.y''(yx) : {}^{\xi}\forall\alpha{:}\star.{}^{\mathsf{L}}(t^* \multimap {}^{\mathsf{L}}({}^{\mathsf{L}}({}^{\cup}1 \multimap {}^{\cup}1) \multimap {}^{\cup}1))$ by (14–15), rule T-TAbs

(17) $\mathrm{D}^*;\bullet, y:{}^{\xi_0}(t^* \multimap {}^{\cup}1) \vdash \Lambda.\lambda x.\lambda y''.y''(yx) : (^{\xi}(t \xrightarrow{\perp_{\mathcal{D}}}_{\circ} {}^{\cup}1))^*$ by (16), def. $\overline{t}^*$

(18) $\mathrm{D}^*;\mathrm{G}^*, y:{}^{\xi_0}(t^* \multimap {}^{\cup}1) \vdash (\lambda y'.[\![e']\!]_{\mathcal{D}}(\lambda x.x))(\Lambda.\lambda x.\lambda y''.y''(yx)) : {}^{\cup}1$ by (7, 17), rule T-App

(19) $\mathrm{D}^*;\mathrm{G}^* \vdash \lambda y.(\lambda y'.[\![e']\!]_{\mathcal{D}}(\lambda x.x))(\Lambda.\lambda x.\lambda y''.y''(yx)) : {}^{\mathsf{L}}({}^{\xi_0}(t^* \multimap {}^{\mathsf{U}}1) \multimap {}^{\mathsf{U}}1) \multimap {}^{\mathsf{U}}1)$ by (18), rule T-Abs

(20) $\mathrm{D}^*;\mathrm{G}^* \vdash [\![\mathsf{shift}\, y' \,\mathsf{in}\, e']\!]_{\mathcal{D}} : {}^{\mathsf{L}}({}^{\xi_0}(t^* \multimap {}^{\mathsf{U}}1) \multimap {}^{\mathsf{U}}1)$ by (3, 19). □

C.3.2 Answer-Type Modification Effects

In this section, we consider the delimited continuations with answer-type modification effects from §8.5.2.

Definition C.8 (Answer-type effects to simple shift/reset effects)**.**

We define a translation from answer-type modification effects to fixed-answer type effects:

$$\mathscr{D}(\bot_{\mathcal{A}}) = \bot_{\mathcal{D}}$$
$$\mathscr{D}({}^{\xi_1,\ldots,\xi_j}(t_1 \rightarrowtail t_2)) = \overline{\xi_1} \sqcup \ldots \sqcup \overline{\xi_j}$$

Lemma C.9 (Effect translation)**.**

For all a and a':

1. $a^* = \mathscr{D}(a)^*$.
2. $\mathscr{D}(a \oslash a') = \mathscr{D}(a) \sqcup \mathscr{D}(a')$ *when* $a \oslash a'$ *is defined.*
3. *If* $\mathrm{D} \vdash_{\mathcal{A}} a : \mathsf{CTL}$ *then* $\mathrm{D} \vdash_{\mathcal{D}} \mathscr{D}(a) : \mathsf{CTL}$.
4. *If* $\mathrm{D} \vdash_{\mathcal{A}} a_1 \preceq a_2$ *then* $\mathrm{D} \vdash_{\mathcal{D}} \mathscr{D}(a_1) \preceq \mathscr{D}(a_2)$.

Proof.

1. By cases on a:

   Case $\bot_{\mathcal{A}}$.

   Then ${\bot_{\mathcal{A}}}^* = \mathsf{L} = {\bot_{\mathcal{D}}}^* = \mathscr{D}(\bot_{\mathcal{A}})^*$.

   Case ${}^{\Xi}(t_1 \rightarrowtail t_2)$.

   By cases on Ξ:

Case ξ**.**

Then ${}^{\xi}(t_1 \rightarrowtail t_2)^* = \xi = \overline{\xi}^* = \mathscr{D}({}^{\xi}(t_1 \rightarrowtail t_2))^*$.

Case $\xi_1, \ldots, \xi_j$**.**

Then $({}^{\xi_1,\ldots,\xi_j}(t_1 \rightarrowtail t_2))^* = \mathsf{U} = (\overline{\xi_1} \sqcup \ldots \sqcup \overline{\xi_j})^* = \mathscr{D}({}^{\xi_1,\ldots,\xi_j}(t_1 \rightarrowtail t_2))^*$.

2. By cases on a:

Case $\bot_{\mathcal{A}}$**.**

Then $\mathscr{D}(\bot_{\mathcal{A}} \oslash a') = \mathscr{D}(a_2) = \bot_{\mathcal{D}} \sqcup \mathscr{D}(a') = \mathscr{D}(\bot_{\mathcal{A}}) \sqcup \mathscr{D}(a')$.

Case ${}^{\Xi}(t_1 \rightarrowtail t_2)$**.**

By cases on a':

Case $\bot_{\mathcal{A}}$**.**

By symmetry with the $a = \bot_{\mathcal{A}}$ case above.

Case ${}^{\Xi'}(t'_1 \rightarrowtail t'_2)$**.**

Then ${}^{\Xi}(t_1 \rightarrowtail t_2) \oslash {}^{\Xi'}(t'_1 \rightarrowtail t'_2)$ is well formed only if $t_1 = t'_2$. Let $\xi_1, \ldots, \xi_j = \Xi$ and $\xi'_1, \ldots, \xi'_k = \Xi'$. Then,

(1) $\quad \mathscr{D}({}^{\xi_1,\ldots,\xi_j}(t_1 \rightarrowtail t_2) \oslash {}^{\xi'_1,\ldots,\xi'_k}(t'_1 \rightarrowtail t'_2))$

(2) $\quad = \mathscr{D}({}^{\xi_1,\ldots,\xi_j,\xi'_1,\ldots,\xi'_k}(t'_1 \rightarrowtail t_2))$

(3) $\quad = \overline{\xi_1} \sqcup \ldots \sqcup \overline{\xi_j} \sqcup \overline{\xi'_1} \sqcup \ldots \sqcup \overline{\xi'_k}$

(4) $\quad = \mathscr{D}({}^{\xi_1,\ldots,\xi_j}(t_1 \rightarrowtail t_2)) \sqcup \mathscr{D}({}^{\xi'_1,\ldots,\xi'_k}(t'_1 \rightarrowtail t'_2))$.

3. By induction on the derivation of $\mathrm{D} \vdash_{\mathcal{A}} a : \mathsf{CTL}$:

Case $\dfrac{}{\mathrm{D} \vdash_{\mathcal{A}} \bot_{\mathcal{A}} : \mathsf{CTL}}$**.**

Then by rule C-K-Bot, $\mathrm{D} \vdash_{\mathcal{D}} \bot_{\mathcal{D}} : \mathsf{CTL}$.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{A}} \xi : \mathsf{QUAL} \qquad \mathrm{D} \vdash_{\mathcal{A}} t_1 : \star \qquad \mathrm{D} \vdash_{\mathcal{A}} t_2 : \star}{\mathrm{D} \vdash_{\mathcal{A}} {}^{\xi}(t_1 \rightarrowtail t_2) : \mathsf{CTL}}$**.**

Then by rule D-K-Qual, $\mathrm{D} \vdash_{\mathcal{D}} \overline{\xi} : \mathsf{CTL}$.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{A}} {}^{\xi_1,\ldots,\xi_j}(t_1 \rightarrowtail t_2) : \mathsf{CTL} \qquad \mathrm{D} \vdash_{\mathcal{A}} {}^{\xi'_1,\ldots,\xi'_k}(t_1 \rightarrowtail t_2) : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{A}} {}^{\xi_1,\ldots,\xi_j,\xi'_1,\ldots,\xi'_k}(t_1 \rightarrowtail t_2) : \mathsf{CTL}}$**.**

By the induction hypothesis, twice,

(1) $\mathrm{D} \vdash_{\mathcal{D}} \overline{\xi_1} \sqcup \ldots \sqcup \overline{\xi_j} : \mathsf{CTL}$ and

(2) $\mathrm{D} \vdash_{\mathcal{D}} \overline{\xi'_1} \sqcup \ldots \sqcup \overline{\xi'_k} : \mathsf{CTL}$.

Then by rule D-K-JOIN,

(3) $\mathrm{D} \vdash_{\mathcal{D}} \overline{\xi_1} \sqcup \ldots \sqcup \overline{\xi_j} \sqcup \overline{\xi'_1} \sqcup \ldots \sqcup \overline{\xi'_k} : \mathsf{CTL}$,

or equivalently,

(4) $\mathrm{D} \vdash_{\mathcal{D}} \mathscr{D}(^{\xi_1,\ldots,\xi_j}(t_1 \rightarrowtail t_2)) \otimes \mathscr{D}(^{\xi'_1,\ldots,\xi'_k}(t_1 \rightarrowtail t_2)) : \mathsf{CTL}$.

Otherwise.

No other cases assign kind CTL to a type.

4. By induction on the derivation of $\mathrm{D} \vdash_{\mathcal{A}} a_1 \preceq a_2$:

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{A}} a : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{A}} a \preceq a}$.

Then $\mathrm{D} \vdash_{\mathcal{D}} \mathscr{D}(a) \preceq \mathscr{D}(a)$ by rule CSUB-REFL.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{A}} a_1 \preceq a' \qquad \mathrm{D} \vdash_{\mathcal{A}} a' \preceq a_2}{\mathrm{D} \vdash_{\mathcal{A}} a_1 \preceq a_2}$.

By the induction hypothesis twice and rule CSUB-TRANS.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi}(t \rightarrowtail t) : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{A}} \bot_{\mathcal{A}} \preceq {}^{\Xi}(t \rightarrowtail t)}$.

Then $\mathrm{D} \vdash_{\mathcal{D}} \bot_{\mathcal{D}} \preceq \mathscr{D}({}^{\Xi}(t \rightarrowtail t))$ by rule DSUB-BOT.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{A}} {}^{\xi_1,\ldots,\xi_j}(t_1 \rightarrowtail t_2) : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{A}} {}^{\mathsf{L}}(t_1 \rightarrowtail t_2) \preceq {}^{\xi_1,\ldots,\xi_j}(t_1 \rightarrowtail t_2)}$.

For each ξ_k in $\xi_1, \ldots, \xi_j$, by rule DSUB-LIN, we have that $\mathrm{D} \vdash_{\mathcal{D}} \overline{\mathsf{L}} \preceq \overline{\xi_k}$. By induction on the length of $\overline{\xi_1} \sqcup \ldots \sqcup \overline{\xi_j}$ and repeated application of rule DSUB-JOIN, we have that $\mathrm{D} \vdash_{\mathcal{D}} \overline{\mathsf{L}} \preceq \overline{\xi_1} \sqcup \ldots \sqcup \overline{\xi_j}$. Then note that ${}^{\mathsf{L}}(t_1 \rightarrowtail t_2) = {}^{\mathsf{L}}(t_1 \rightarrowtail t_2)$ by the quotienting of Ξ.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{A}} {}^{\xi_1,\ldots,\xi_j}(t_1 \rightarrowtail t_2) : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{A}} {}^{\xi_1,\ldots,\xi_j}(t_1 \rightarrowtail t_2) \preceq {}^{\mathsf{U}}(t_1 \rightarrowtail t_2)}$.

As in the previous case, but using rule DSUB-TOP for each $\mathrm{D} \vdash_{\mathcal{D}} \overline{\xi_j} \preceq \overline{\mathsf{U}}$.

$$\textbf{Case}\ \frac{\mathrm{D} \vdash_{\mathcal{A}} {}^{\xi_1,\ldots,\xi_j}(t_1 \rightarrowtail t_2) \preceq {}^{\xi''_1,\ldots,\xi''_k}(t_1 \rightarrowtail t_2) \quad \mathrm{D} \vdash_{\mathcal{A}} {}^{\xi'_1,\ldots,\xi'_m}(t_1 \rightarrowtail t_2) \preceq {}^{\xi'''_1,\ldots,\xi'''_n}(t_1 \rightarrowtail t_2)}{\mathrm{D} \vdash_{\mathcal{A}} {}^{\xi_1,\ldots,\xi_j,\xi''_1,\ldots,\xi''_k}(t_1 \rightarrowtail t_2) \preceq {}^{\xi'_1,\ldots,\xi'_m,\xi'''_1,\ldots,\xi'''_n}(t_1 \rightarrowtail t_2)}.$$

By the induction hypothesis,

(1) $\mathrm{D} \vdash_{\mathcal{D}} \overline{\xi_1} \sqcup \ldots \sqcup \overline{\xi_j} \preceq \overline{\xi''_1} \sqcup \ldots \sqcup \overline{\xi''_k}$ and

(2) $\mathrm{D} \vdash_{\mathcal{D}} \overline{\xi'_1} \sqcup \ldots \sqcup \overline{\xi'_m} \preceq \overline{\xi'''_1} \sqcup \ldots \sqcup \overline{\xi'''_n}$.

Then by rule DSUB-JOIN,

(3) $\mathrm{D} \vdash_{\mathcal{D}} \overline{\xi_1} \sqcup \ldots \sqcup \overline{\xi_j} \sqcup \overline{\xi'_1} \sqcup \ldots \sqcup \overline{\xi'_m} \preceq \overline{\xi''_1} \sqcup \ldots \sqcup \overline{\xi''_k} \sqcup \overline{\xi'''_1} \sqcup \ldots \sqcup \overline{\xi'''_n}$. □

LEMMA C.10 (No information)**.**

If $a^* = \mathsf{U}$ *then* $(a \oslash a')^* = \mathsf{U}$.

Proof. Then:

(1)	$a^* = \mathsf{U}$	by antecedent
(2)	$\mathscr{D}(a)^* = \mathsf{U}$	by (1), lemma C.2.1
(3)	$(a \oslash a')^* = \mathscr{D}(a \oslash a')^*$	by lemma C.2.1
(4)	$= (\mathscr{D}(a) \sqcup \mathscr{D}(a'))^*$	by lemma C.2.2
(5)	$= \mathsf{U}$	by (2), lemma C.7. □

THEOREM 8.14 (Answer-type effect properties, restated from p. 239)**.**

Answer-type modification effects $(\mathcal{A}, \bot_{\mathcal{A}}, \circ)$ *satisfy properties 1–5.*

Proof.

Property 1 (Answer types).

1. By the definition, $\langle\!\langle \tau, \bot_{\mathcal{A}} \rangle\!\rangle^-_{\mathcal{A}} = \tau = \langle\!\langle \tau, \bot_{\mathcal{A}} \rangle\!\rangle^+_{\mathcal{A}}$. Thus, $\langle\!\langle \tau \rangle\!\rangle_{\mathcal{A}} = \tau$.

2. Let $\mathrm{D}^* \vdash \tau : \star$ and $\mathrm{D} \vdash_{\mathcal{A}} a : \mathsf{CTL}$. We need to show that $\mathrm{D}^* \vdash \langle\!\langle \tau, a \rangle\!\rangle^-_{\mathcal{A}} : \star$ and $\mathrm{D}^* \vdash \langle\!\langle \tau, a \rangle\!\rangle^+_{\mathcal{A}} : \star$. By induction on the latter derivation:

Case $\dfrac{}{\mathrm{D} \vdash_{\mathcal{A}} \bot_{\mathcal{A}} : \mathsf{CTL}}$.

Then $\langle\!\langle \tau, a \rangle\!\rangle^-_{\mathcal{A}} = \langle\!\langle \tau, a \rangle\!\rangle^+_{\mathcal{A}} = \langle\!\langle \tau \rangle\!\rangle_{\mathcal{A}} = \tau$, and $\mathrm{D}^* \vdash \tau : \star$ by our assumption.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{A}} \xi : \mathsf{QUAL} \qquad \mathrm{D} \vdash_{\mathcal{A}} t_1 : \star \qquad \mathrm{D} \vdash_{\mathcal{A}} t_2 : \star}{\mathrm{D} \vdash_{\mathcal{A}} {}^{\xi}(t_1 \rightarrowtail t_2) : \mathsf{CTL}}$.

Then $\langle\!\langle \tau, a \rangle\!\rangle^-_{\mathcal{A}} = {t_1}^*$ and $\langle\!\langle \tau, a \rangle\!\rangle^+_{\mathcal{A}} = {t_2}^*$. By lemma 8.5, $\mathrm{D}^* \vdash {t_1}^* : \star$ and $\mathrm{D}^* \vdash {t_1}^* : \star$.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi_1}(t_1 \rightarrowtail t_2) : \mathsf{CTL} \qquad \mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi_2}(t_1 \rightarrowtail t_2) : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi_1, \Xi_2}(t_1 \rightarrowtail t_2) : \mathsf{CTL}}$.

Then we have that $\langle\!\langle \tau, a \rangle\!\rangle^-_{\mathcal{A}} = {t_1}^* = \langle\!\langle \tau, {}^{\Xi_1}(t_1 \rightarrowtail t_2) \rangle\!\rangle^-_{\mathcal{A}}$ and $\langle\!\langle \tau, a \rangle\!\rangle^+_{\mathcal{A}} = {t_2}^* = \langle\!\langle \tau, {}^{\Xi_1}(t_1 \rightarrowtail t_2) \rangle\!\rangle^+_{\mathcal{A}}$. Then by the induction hypothesis.

Otherwise.

No other cases assign kind CTL to a type.

3. Let $c_1 \neq \bot_{\mathcal{A}}$, $c_2 \neq \bot_{\mathcal{A}}$, and $\mathrm{D} \vdash_{\mathcal{A}} a_1 \otimes a_2 : \mathsf{CTL}$. By the definition of effect sequencing, $a_1 \otimes a_2$ is defined only if one of the effects is $\bot_{\mathcal{A}}$, which is rules out by the assumption, or if the sequenced effect is of the form

$$a_1 \otimes a_2 = {}^{\Xi}(t' \rightarrowtail t_2) \otimes {}^{\Xi'}(t_1 \rightarrowtail t') = {}^{\Xi, \Xi'}(t_1 \rightarrowtail t_2).$$

Then:

a) $\langle\!\langle \tau, a_1 \otimes a_2 \rangle\!\rangle^-_{\mathcal{A}} = \langle\!\langle \tau, {}^{\Xi, \Xi'}(t_1 \rightarrowtail t_2) \rangle\!\rangle^-_{\mathcal{A}} = {t_1}^* = \langle\!\langle \tau, {}^{\Xi'}(t_1 \rightarrowtail t') \rangle\!\rangle^-_{\mathcal{A}} = \langle\!\langle \tau, a_2 \rangle\!\rangle^-_{\mathcal{A}}$.

b) $\langle\!\langle \tau, a_1 \otimes a_2 \rangle\!\rangle^+_{\mathcal{A}} = \langle\!\langle \tau, {}^{\Xi, \Xi'}(t_1 \rightarrowtail t_2) \rangle\!\rangle^+_{\mathcal{A}} = {t_2}^* = \langle\!\langle \tau, {}^{\Xi}(t' \rightarrowtail t_2) \rangle\!\rangle^+_{\mathcal{A}} = \langle\!\langle \tau, a_1 \rangle\!\rangle^+_{\mathcal{A}}$.

c) $\langle\!\langle \tau, a_1 \rangle\!\rangle^-_{\mathcal{A}} = \langle\!\langle \tau, {}^{\Xi}(t' \rightarrowtail t_2) \rangle\!\rangle^-_{\mathcal{A}} = {t'}^* = \langle\!\langle \tau, {}^{\Xi'}(t_1 \rightarrowtail t') \rangle\!\rangle^+_{\mathcal{A}} = \langle\!\langle \tau, a_2 \rangle\!\rangle^+_{\mathcal{A}}$.

4. Let $\mathrm{D} \vdash_{\mathcal{A}} a_1 \preceq a_2$. Given an arbitrary type τ, we must find a type τ' such that $\langle\!\langle \tau', a_1 \rangle\!\rangle^-_{\mathcal{A}} = \langle\!\langle \tau, a_2 \rangle\!\rangle^-_{\mathcal{A}}$ and $\langle\!\langle \tau', a_1 \rangle\!\rangle^+_{\mathcal{A}} = \langle\!\langle \tau, a_2 \rangle\!\rangle^+_{\mathcal{A}}$. By induction on the effect subsumption derivation:

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{A}} a : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{A}} a \preceq a}$.

Then $a_1 = a_2$, so by substitution of one for the other.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{A}} a_1 \preceq a' \qquad \mathrm{D} \vdash_{\mathcal{A}} a' \preceq a_2}{\mathrm{D} \vdash_{\mathcal{A}} a_1 \preceq a_2}$.

By the induction hypothesis, twice, and transitivity of equality.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi}(t \rightarrowtail t) : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{A}} \bot_{\mathcal{A}} \preceq {}^{\Xi}(t \rightarrowtail t)}$.

Let $\tau' = t^*$. Then $\langle\!\langle t^*, a_1 \rangle\!\rangle^-_{\mathcal{A}} = \langle\!\langle t^* \rangle\!\rangle_{\mathcal{A}} = t^* = \langle\!\langle \tau, {}^{\Xi}(t \rightarrowtail t) \rangle\!\rangle^-_{\mathcal{A}}$, and likewise $\langle\!\langle t^*, a_1 \rangle\!\rangle^+_{\mathcal{A}} = \langle\!\langle t^* \rangle\!\rangle_{\mathcal{A}} = t^* = \langle\!\langle \tau, {}^{\Xi}(t \rightarrowtail t) \rangle\!\rangle^+_{\mathcal{A}}$.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi}(t_1 \rightarrowtail t_2) : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{A}} {}^{\mathsf{L}}(t_1 \rightarrowtail t_2) \preceq {}^{\Xi}(t_1 \rightarrowtail t_2)}$.

It does not matter what τ' we choose, so let $\tau' = \tau$. Then $\langle\!\langle \tau, a_1 \rangle\!\rangle^-_{\mathcal{A}} = {t_1}^* = \langle\!\langle \tau, a_2 \rangle\!\rangle^-_{\mathcal{A}}$ and $\langle\!\langle \tau, a_1 \rangle\!\rangle^+_{\mathcal{A}} = {t_2}^* = \langle\!\langle \tau, a_2 \rangle\!\rangle^+_{\mathcal{A}}$.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi}(t_1 \rightarrowtail t_2) : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi}(t_1 \rightarrowtail t_2) \preceq {}^{\mathsf{U}}(t_1 \rightarrowtail t_2)}$.

As in the previous case, let $\tau' = \tau$.

Case $\dfrac{\mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi_1}(t_1 \rightarrowtail t_2) \preceq {}^{\Xi'_1}(t_1 \rightarrowtail t_2) \qquad \mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi_2}(t_1 \rightarrowtail t_2) \preceq {}^{\Xi'_2}(t_1 \rightarrowtail t_2)}{\mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi_1,\Xi_2}(t_1 \rightarrowtail t_2) \preceq {}^{\Xi'_1,\Xi'_2}(t_1 \rightarrowtail t_2)}$.

As in the previous case, let $\tau' = \tau$.

Property 2 (Done).

(1) $\Delta; \bullet, x{:}\tau \vdash x : \tau$ — by rule T-VAR

(2) $\Delta; \bullet \vdash \lambda x.x : {}^{\mathsf{L}}(\tau \multimap \tau)$ — by (1), rule T-ABS

(3) $\langle\!\langle \tau \rangle\!\rangle_{\mathcal{A}} = \tau$ — by def. $\langle\!\langle \tau \rangle\!\rangle_{\mathcal{A}}$

(4) $\Delta; \bullet \vdash \lambda x.x : {}^{\mathsf{L}}(\tau \multimap \langle\!\langle \tau \rangle\!\rangle_{\mathcal{A}})$ — by (2–3).

Property 3 (Effect sequencing). Let $\mathrm{D} \vdash_{\mathcal{A}} a_1 \oslash a_2 : \mathsf{CTL}$. By lemma C.9,

(1) $(a_1 \oslash a_2)^* = \mathscr{D}(a_1 \oslash a_2)^* = \mathscr{D}(a_1)^* \sqcup \mathscr{D}(a_2)^*$,

(2) ${a_1}^* = \mathscr{D}(a_1)^*$,

(3) ${a_2}^* = \mathscr{D}(a_2)^*$,

(4) $\mathrm{D} \vdash_{\mathcal{D}} \mathscr{D}(a_1) \sqcup \mathscr{D}(a_2) : \mathsf{CTL}$.

By this same property for d effects, we have that

(5) $\mathrm{D}^* \vdash \mathscr{D}(a_1)^* \sqcup \mathscr{D}(a_2)^* \preceq \mathscr{D}(a_1)^*$ and

(6) $\mathrm{D}^* \vdash \mathscr{D}(a_1)^* \sqcup \mathscr{D}(a_2)^* \preceq \mathscr{D}(a_2)^*$.

Property 4 (Bottom and lifting).

1. By the definition of $a_1 \oslash a_2$.

2. By cases on a_1 and a_2:

   Case $\bot_{\mathcal{A}}$ and $\bot_{\mathcal{A}}$**.**

   Contradicts the assumption that $a_1 \oslash a_2 \neq \bot_{\mathcal{A}}$.

   Case $\bot_{\mathcal{A}}$ and ${}^{\Xi_2}(t_1 \rightarrowtail t')$**.**

   Let $c_1' = {}^{\mathsf{L}}(t' \rightarrowtail t')$ and $c_2' = c_2$.

   Case ${}^{\Xi_1}(t' \rightarrowtail t_2)$ and $\bot_{\mathcal{A}}$**.**

   Let $c_1' = c_1$ and $c_2' = {}^{\mathsf{L}}(t' \rightarrowtail t')$.

   Case ${}^{\Xi_1}(t' \rightarrowtail t_2)$ and ${}^{\Xi_2}(t_1 \rightarrowtail t')$**.**

   Let $c_1' = c_1$ and $c_2' = c_2$.

Property 5 (New rules).

1. For translation of effect bounds, let $\mathrm{D} \vdash_{\mathcal{A}} a \succeq \xi$; we need to show that $\mathrm{D}^* \vdash \xi \preceq a^*$. We proceed by induction on the derivation of $\mathrm{D} \vdash_{\mathcal{A}} a \succeq \xi$, which has but one new case to consider:

   Case $\dfrac{\mathrm{D} \vdash_{\mathcal{A}} \xi \preceq \xi_1 \quad \cdots \quad \mathrm{D} \vdash_{\mathcal{A}} \xi \preceq \xi_j \qquad \mathrm{D} \vdash_{\mathcal{A}} t_1 : \star \qquad \mathrm{D} \vdash_{\mathcal{A}} t_2 : \star}{\mathrm{D} \vdash_{\mathcal{A}} {}^{\xi_1,\ldots,\xi_j}(t_1 \rightarrowtail t_2) \succeq \xi}$**.**

   If $\xi_1, \ldots, \xi_j$ is equivalent to a single qualifier ξ', then ${}^{\xi'}(t_1 \rightarrowtail t_2)^* = \xi'$. By the premises, $\mathrm{D} \vdash_{\mathcal{A}} \xi \preceq \xi'$, and by lemma C.6, $\mathrm{D}^* \vdash \xi \preceq \xi'$.

   Otherwise, $\xi_1, \ldots, \xi_j$ is not equivalent to a single qualifier. Based on the quotienting of $\mathcal{A}$, means that there are some qualifiers ξ_i and ξ_i' in the collection of qualifiers such that $\xi_i \sqcap \xi_i'$ is undefined. By lemma C.3, we know that $\xi = \mathsf{U}$. Then by rule QSUB-BOT.

2. For translation of effect subsumption, let $\mathrm{D} \vdash_{\mathcal{A}} a_1 \preceq a_2$; we must show that $\mathrm{D}^* \vdash {a_2}^* \preceq {a_1}^*$.

   (1) $\mathrm{D} \vdash_{\mathcal{D}} \mathscr{D}(a_1) \preceq \mathscr{D}(a_2)$ by lemma C.2.4

   (2) $\mathrm{D}^* \vdash \mathscr{D}(a_2)^* \preceq \mathscr{D}(a_1)^*$ by property 5 for d

   (3) $\mathrm{D}^* \vdash {a_2}^* \preceq {a_1}^*$ by lemma C.2.1.

3. For translation of kinding, let $\mathrm{D} \vdash_{\mathcal{A}} a : \mathsf{CTL}$. By lemma C.2.3, $\mathrm{D} \vdash_{\mathcal{D}} \mathscr{D}(a) : \mathsf{CTL}$. Then by lemma 8.5, $\mathrm{D}^* \vdash \mathscr{D}(a)^* : \mathsf{QUAL}$. Note, finally, that $\mathscr{D}(a)^* = a^*$ by lemma C.2.1.

4. For translation of typing, let

   - $\mathrm{D};\mathrm{G} \vdash_{\mathcal{A}} e : t ; a$.
   - $\mathrm{D}^* \vdash \xi_0 \preceq a^*$, and
   - $\mathrm{D}^* \vdash \tau' : \star$.

   We must show that $\mathrm{D}^*;\mathrm{G}^* \vdash [\![e]\!]_{\mathcal{A}} : {}^{\mathsf{L}}({}^{\xi_0}(t^* \multimap \langle\!\langle \tau', a \rangle\!\rangle^-_{\mathcal{A}}) \multimap \langle\!\langle \tau', a \rangle\!\rangle^-_{\mathcal{A}})$. We proceed by induction on the typing derivation, with two cases to consider:

   Case $\dfrac{\mathrm{D};\mathrm{G} \vdash_{\mathcal{A}} e' : t_0 ; {}^{\Xi}(t_0 \rightarrowtail t)}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{A}} \mathsf{reset}\, e' : t ; \bot_{\mathcal{A}}}$.

   We must show that $\mathrm{D}^*;\mathrm{G}^* \vdash [\![e']\!]_{\mathcal{A}} : {}^{\mathsf{L}}({}^{\xi_0}(t^* \multimap \tau') \multimap \tau')$. Let $a' = {}^{\Xi}(t_0 \rightarrowtail t)$. Then,

   (1) $[\![\mathsf{reset}\, e']\!]_{\mathcal{A}} = \lambda y.\, y([\![e']\!]_{\mathcal{D}}(\lambda x.x))$ by def. $[\![e]\!]_{\mathcal{A}}$

   (2) $\mathrm{D}^*;\mathrm{G}^* \vdash [\![e']\!]_{\mathcal{A}} : {}^{\mathsf{L}}({}^{{a'}^*}({t_0}^* \multimap {t_0}^*) \multimap t^*)$ by IH

   (3) $\mathrm{D}^*;\bullet, x{:}{t_0}^* \vdash x : {t_0}^*$ by rule T-VAR

   (4) $\mathrm{D}^*;\bullet \vdash \lambda x.x : {}^{{a'}^*}({t_0}^* \multimap {t_0}^*)$ by (3), rule T-ABS

   (5) $\mathrm{D}^*;\mathrm{G}^* \vdash [\![e']\!]_{\mathcal{A}}(\lambda x.x) : t^*$ by (2, 4), rule T-APP

   (6) $\mathrm{D}^*;\bullet, y{:}{}^{\xi_0}(t^* \multimap \tau') \vdash y : {}^{\xi_0}(t^* \multimap \tau')$ by rule T-VAR

   (7) $\mathrm{D}^*;\mathrm{G}^*, y{:}{}^{\xi_0}(t^* \multimap \tau') \vdash y([\![e']\!]_{\mathcal{A}}(\lambda x.x)) : \tau'$ by (5–6), rule T-APP

(8) $\mathrm{D}^*;\mathrm{G}^* \vdash \lambda y.\, y([\![e']\!]_{\mathcal{A}}(\lambda x.x)) : {}^{\mathsf{L}}({}^{\xi_0}(t^* \multimap \tau') \multimap \tau')$
by (7), rule T-ABS

(9) $\mathrm{D}^*;\mathrm{G}^* \vdash [\![\mathsf{reset}\, e']\!]_{\mathcal{A}} : {}^{\mathsf{L}}({}^{\xi_0}(t^* \multimap \tau') \multimap \tau')$
by (1, 8).

Case $\dfrac{\mathrm{D};\mathrm{G}, y' {:}^{\xi}(t_1 \xrightarrow{\perp_{\mathcal{A}}\circ} t_2) \vdash_{\mathcal{A}} e' : t_0 ;\, {}^{\Xi}(t_0 \rightarrowtail t)}{\mathrm{D};\mathrm{G} \vdash_{\mathcal{A}} \mathsf{shift}\, y' \,\mathsf{in}\, e' : t_1 ;\, {}^{\Xi,\xi}(t_2 \rightarrowtail t)}$.

We must show that $\mathrm{D}^*;\mathrm{G}^* \vdash [\![\mathsf{shift}\, y' \,\mathsf{in}\, e']\!]_{\mathcal{A}} : {}^{\mathsf{L}}({}^{\xi_0}({t_1}^* \multimap {t_2}^*) \multimap t^*)$.
Let $a' = {}^{\Xi}(t_0 \rightarrowtail t)$.

(1) $[\![\mathsf{shift}\, y' \,\mathsf{in}\, e']\!]_{\mathcal{A}} = \lambda y.(\lambda y'.[\![e']\!]_{\mathcal{A}}(\lambda x.x))(\Lambda.\lambda x.\lambda y''.y''(yx))$
by def. $[\![e]\!]_{\mathcal{A}}$

(2) $\mathrm{D}^* \vdash \xi_0 \preceq ({}^{\Xi,\xi}(t_2 \rightarrowtail t))^*$ by lem. assumption

(3) $\mathrm{D}^* \vdash ({}^{\Xi}(t_0 \rightarrowtail t) \otimes {}^{\xi}(t_2 \rightarrowtail t_0))^* \preceq {}^{\xi}(t_2 \rightarrowtail t_0)^*$
by property 3

(4) $\mathrm{D}^* \vdash ({}^{\Xi,\xi}(t_2 \rightarrowtail t))^* \preceq \xi$ by (3), def a^*

(5) $\mathrm{D}^* \vdash \xi_0 \preceq \xi$ by (2, 4), trans.

(6) $\mathrm{D}^* \vdash \bullet, y{:}^{\xi_0}({t_1}^* \multimap {t_2}^*) \preceq \xi$ by (5)

(7) $\mathrm{D}^*;\mathrm{G}^*, y' {:} ({}^{\xi}(t_1 \xrightarrow{\perp_{\mathcal{A}}\circ} t_2))^* \vdash [\![e']\!]_{\mathcal{A}} : {}^{\mathsf{L}}({}^{a'^*}({t_0}^* \multimap {t_0}^*) \multimap t^*)$
by IH

(8) $\mathrm{D}^*;\bullet \vdash \lambda x.x : {}^{a'^*}({t_0}^* \multimap {t_0}^*)$ by rules T-VAR and T-ABS

(9) $\mathrm{D}^*;\mathrm{G}^*, y' {:} ({}^{\xi}(t_1 \xrightarrow{\perp_{\mathcal{A}}\circ} t_2))^* \vdash [\![e']\!]_{\mathcal{A}}(\lambda x.x) : t^*$
by (7–8), rule T-APP

(10) $\mathrm{D}^*;\mathrm{G}^* \vdash \lambda y'.[\![e']\!]_{\mathcal{A}}(\lambda x.x) : {}^{\mathsf{L}}(({}^{\xi}(t_1 \xrightarrow{\perp_{\mathcal{A}}\circ} t_2))^* \multimap t^*)$
by (9), rule T-ABS

(11) $\mathrm{D}^*, \alpha{:}\star;\bullet, y''{:}{}^{\mathsf{L}}({t_2}^* \multimap \alpha) \vdash y'' : {}^{\mathsf{L}}({t_2}^* \multimap \alpha)$
by rule T-VAR

(12) $\mathrm{D}^*, \alpha{:}\star;\bullet, y{:}^{\xi_0}({t_1}^* \multimap {t_2}^*) \vdash y : {}^{\xi_0}({t_1}^* \multimap {t_2}^*)$
by rule T-VAR

(13) $\mathrm{D}^*, \alpha{:}\star;\bullet, x{:}{t_1}^* \vdash x : {t_1}^*$ by rule T-VAR

(14) $\mathrm{D}^*, \alpha{:}\star;\bullet, y{:}^{\xi_0}({t_1}^* \multimap {t_2}^*), x{:}{t_1}^* \vdash yx : {t_2}^*$
by (12–13), rule T-APP

(15) $\mathrm{D}^*, \alpha{:}\star; \bullet, y{:}^{\xi_0}(t_1{}^* \multimap t_2{}^*), x{:}t_1{}^*, y''{:}^{\mathsf{L}}(t_2{}^* \multimap \alpha) \vdash y''(yx) : \alpha$
by (11, 14), rule T-App

(16) $\mathrm{D}^*, \alpha{:}\star; \bullet, y{:}^{\xi_0}(t_1{}^* \multimap t_2{}^*), x{:}t_1{}^* \vdash \lambda y''.\, y''(yx) : {}^{\mathsf{L}}({}^{\mathsf{L}}(t_2{}^* \multimap \alpha) \multimap \alpha)$
by (15), rule T-Abs

(17) $\mathrm{D}^*, \alpha{:}\star; \bullet, y{:}^{\xi_0}(t_1{}^* \multimap t_2{}^*) \vdash \lambda x.\, \lambda y''.\, y''(yx) : {}^{\mathsf{L}}(t_1{}^* \multimap {}^{\mathsf{L}}({}^{\mathsf{L}}(t_2{}^* \multimap \alpha) \multimap \alpha))$
by (16), rule T-Abs

(18) $\mathrm{D}^*; \bullet, y{:}^{\xi_0}(t_1{}^* \multimap t_2{}^*) \vdash \Lambda.\, \lambda x.\, \lambda y''.\, y''(yx) : {}^{\xi}\forall \alpha{:}\star.\, {}^{\mathsf{L}}(t_1{}^* \multimap {}^{\mathsf{L}}({}^{\mathsf{L}}(t_2{}^* \multimap \alpha) \multimap \alpha))$
by (6, 17), rule T-TAbs

(19) $\mathrm{D}^*; \bullet, y{:}^{\xi_0}(t_1{}^* \multimap t_2{}^*) \vdash \Lambda.\, \lambda x.\, \lambda y''.\, y''(yx) : ({}^{\xi}(t_1 \xrightarrow{\perp_{\mathcal{A}}} t_2))^*$
by (18), def. $\overline{t}^*$

(20) $\mathrm{D}^*; \mathrm{G}^*, y{:}^{\xi_0}(t_1{}^* \multimap t_2{}^*) \vdash (\lambda y'.\, [\![e']\!]_{\mathcal{A}}(\lambda x.x))(\Lambda.\, \lambda x.\, \lambda y''.\, y''(yx)) : t^*$
by (10, 19), rule T-App

(21) $\mathrm{D}^*; \mathrm{G}^* \vdash \lambda y.(\lambda y'.\, [\![e']\!]_{\mathcal{A}}(\lambda x.x))(\Lambda.\, \lambda x.\, \lambda y''.\, y''(yx)) : {}^{\mathsf{L}}({}^{\xi_0}(t_1{}^* \multimap t_2{}^*) \multimap t^*)$
by (20), rule T-Abs

(22) $\mathrm{D}^*; \mathrm{G}^* \vdash [\![\mathsf{shift}\, y' \,\mathsf{in}\, e']\!]_{\mathcal{A}} : {}^{\mathsf{L}}({}^{\xi_0}(t_1{}^* \multimap t_2{}^*) \multimap t^*)$
by (1, 21). □

C.3.3 Exception Effects

In this section, we consider the exception effects from §8.5.3.

Theorem 8.15 (Exception effect properties, restated from p. 242)**.**

Exception effects $(\mathcal{X}, \varnothing, \cup)$ *satisfy properties 1–5.*

Proof.

Property 1 (Answer types).

1. Trivial, because $\langle\!\langle \tau, \Psi \rangle\!\rangle^-_{\mathcal{X}} = {}^{\mathsf{L}}({}^{\bigcup}\mathsf{exn} \oplus \tau) = \langle\!\langle \tau, \Psi \rangle\!\rangle^+_{\mathcal{X}}$ for all τ and Ψ.

2. Then,

   (1) $\mathrm{D}^* \vdash \tau : \star$ by lemma assumption

(2) $\mathrm{D}^* \vdash {}^{\mathsf{U}}\mathsf{exn} : \star$ — by def. exn, rule K-TYPE

(3) $\mathrm{D}^* \vdash {}^{\mathsf{U}}\mathsf{exn} \oplus \tau : \overline{\star}$ — by (1–2), rule K-SUM

(4) $\mathrm{D}^* \vdash {}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{exn} \oplus \tau) : \star$ — by (3), rule K-TYPE

(5) $\mathrm{D}^* \vdash \langle\!\langle \tau, \Psi \rangle\!\rangle^-_{\mathcal{X}} : \star$ and $\mathrm{D}^* \vdash \langle\!\langle \tau, \Psi \rangle\!\rangle^+_{\mathcal{X}} : \star$ — by (4), defs.

3. Trivial, as in the first case.

4. Let $\tau' = \tau$.

Property 2 (Done).

(1) $\Delta \vdash \tau \preceq \mathsf{L}$ — by rule B-VAR or rules B-TYPE and QSUB-TOP

(2) $\Delta \vdash {}^{\mathsf{U}}\mathsf{exn} : \star$ — by rule K-TYPE

(3) $\Delta; \bullet, x{:}\tau \vdash x : \tau$ — by rule T-VAR

(4) $\Delta; \bullet, x{:}\tau \vdash \mathsf{inr}\, x : {}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{exn} \oplus \tau)$ — by (1–3), rule T-INR

(5) $\Delta; \bullet \vdash \lambda x.\, \mathsf{inr}\, x : {}^{\mathsf{L}}(\tau \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{exn} \oplus \tau))$ — by (4), rule T-ABS

(6) $\Delta; \bullet \vdash \mathsf{done}_{\mathcal{X}} : {}^{\mathsf{L}}(\tau \multimap \langle\!\langle \tau \rangle\!\rangle_{\mathcal{X}})$ — by (5), def. $\mathsf{done}_{\mathcal{X}}$, def. $\langle\!\langle \tau \rangle\!\rangle_{\mathcal{X}}$.

Property 3 (Effect sequencing). Let $\mathrm{D} \vdash_{\mathcal{A}} \Psi_1 \cup \Psi_2 : \mathsf{CTL}$. We need that $\mathrm{D}^* \vdash (\Psi_1 \cup \Psi_2)^* \preceq {\Psi_1}^*$ and $\mathrm{D}^* \vdash (\Psi_1 \cup \Psi_2)^* \preceq {\Psi_2}^*$. By cases on Ψ_1 and Ψ_2:

Case $\Psi_1 = \varnothing$ and $\Psi_2 = \varnothing$.

Then $(\Psi_1 \cup \Psi_2)^* = \mathsf{L}$, ${\Psi_1}^* = \mathsf{L}$, and ${\Psi_2}^* = \mathsf{L}$, so by rule QSUB-TOP.

Case $\Psi_1 = \varnothing$ and $\Psi_2 \neq \varnothing$.

Then $(\Psi_1 \cup \Psi_2)^* = \mathsf{A}$, ${\Psi_1}^* = \mathsf{L}$, and ${\Psi_2}^* = \mathsf{A}$, so by rules QSUB-TOP and QSUB-REFL.

Case $\Psi_1 \neq \varnothing$ and $\Psi_2 = \varnothing$.

By symmetry with the previous case.

Case $\Psi_1 \neq \varnothing$ **and** $\Psi_2 \neq \varnothing$**.**

Then $(\Psi_1 \cup \Psi_2)^* = \mathsf{A}$, ${\Psi_1}^* = \mathsf{A}$, and ${\Psi_2}^* = \mathsf{A}$, so by rule QSUB-REFL.

Property 4 (Bottom and lifting).

1. By the definition of set union.
2. Let $c'_1 = c_1$ and $c'_2 = c_2$. □

Property 5 (New rules).

1. For translation of effect bounds, let $\mathrm{D} \vdash_{\mathfrak{X}} \Psi \succeq \xi$; we need to show that $\mathrm{D}^* \vdash \xi \preceq \Psi^*$. We proceed by induction on the derivation of $\mathrm{D} \vdash_{\mathfrak{X}} \Psi \succeq \xi$, which has but one new case to consider:

   Case $\dfrac{\mathrm{D} \vdash_{\mathfrak{X}} \Psi : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathfrak{X}} \Psi \succeq \mathsf{A}}$**.**

   By cases on Ψ:

   Case $\varnothing$**.**

   Then $\Psi^* = \mathsf{L}$, so by rule QSUB-TOP.

   Otherwise.

   Then $\Psi^* = \mathsf{A}$, so by rule QSUB-REFL.

2. For translation of effect subsumption, let $\mathrm{D} \vdash_{\mathfrak{X}} \Psi_1 \preceq \Psi_2$; we must show that $\mathrm{D}^* \vdash {\Psi_2}^* \preceq {\Psi_1}^*$. By cases on Ψ_1 and Ψ_2:

   Case $\Psi_1 = \varnothing$ **and** $\Psi_2 = \varnothing$**.**

   Then ${\Psi_1}^* = \mathsf{L}$ and ${\Psi_2}^* = \mathsf{L}$, so by rule QSUB-TOP.

   Case $\Psi_1 = \varnothing$ **and** $\Psi_2 \neq \varnothing$**.**

   Then ${\Psi_1}^* = \mathsf{L}$ and ${\Psi_2}^* = \mathsf{A}$, so by rule QSUB-TOP.

   Case $\Psi_1 \neq \varnothing$ **and** $\Psi_2 = \varnothing$**.**

   Vacuous, because $\Psi_1 \not\subseteq \Psi_2$.

   Case $\Psi_1 \neq \varnothing$ **and** $\Psi_2 \neq \varnothing$**.**

   Then ${\Psi_1}^* = \mathsf{A}$ and ${\Psi_2}^* = \mathsf{A}$, so by rule QSUB-REFL.

3. By definition $\Psi^* = \mathsf{L}$ or $\Psi^* = \mathsf{A}$, so $\mathrm{D}^* \vdash \Psi^* : \mathsf{QUAL}$ by rule K-QUAL.

4. For translation of typing, let

   - $\mathrm{D};\mathrm{G} \vdash_{\mathfrak{X}} e : t ; \Psi$.
   - $\mathrm{D}^* \vdash \xi_0 \preceq \Psi^*$, and
   - $\mathrm{D}^* \vdash \tau' : \star$.

   We must show that

   $$\mathrm{D}^*;\mathrm{G}^* \vdash [\![e]\!]^{\Psi}_{\mathfrak{X}} : \mathsf{L}(^{\xi_0}(t^* \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau')) \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau')).$$

   We proceed by induction on the typing derivation, with two cases to consider:

   Case $\dfrac{\mathrm{D} \vdash_{\mathfrak{C}} t : \star}{\mathrm{D};\bullet \vdash_{\mathfrak{X}} \mathsf{raise}\ \psi : t ; \{\psi\}}$.

   (1) $[\![\mathsf{raise}\ \psi]\!]^{\{\psi\}}_{\mathfrak{X}} = \lambda x.\ \mathsf{inl}\ \psi^*$ — by def. $[\![e]\!]^{\Psi}_{\mathfrak{X}}$

   (2) $\mathrm{D}^* \vdash {}^{\mathsf{U}}\mathsf{exn} \preceq \mathsf{L}$ — by rules B-Type and QSub-Top

   (3) $\mathrm{D}^* \vdash \bullet, x{:}^{\xi_0}(t^* \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau')) \rightsquigarrow \bullet \boxplus \bullet, x{:}^{\xi_0}(t^* \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau'))$ — by rules S-Nil and S-ConsR

   (4) $\mathrm{D}^*;\bullet \vdash \psi^* : {}^{\mathsf{U}}\mathsf{exn}$ — by def. exn

   (5) $\Psi^* = \mathsf{A}$ — by def. $\{\psi\}^*$

   (6) $\mathrm{D}^* \vdash \xi_0 \preceq \mathsf{A}$ — by lemma assumption, (5)

   (7) $\mathrm{D}^* \vdash \bullet, x{:}^{\xi_0}(t^* \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau')) \preceq \mathsf{A}$ — by (6), rules B-Type and B-Cons

   (8) $\mathrm{D}^*;\bullet, x{:}^{\xi_0}(t^* \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau')) \vdash \psi^* : {}^{\mathsf{U}}\mathsf{exn}$ — by (3–4, 7), rule T-Weak

   (9) $\mathrm{D}^*;\bullet, x{:}^{\xi_0}(t^* \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau')) \vdash \mathsf{inl}\ \psi^* : \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau')$ — by (2, 8), rule T-Inl

   (10) $\mathrm{D}^*;\bullet \vdash \lambda x.\ \mathsf{inl}\ \psi^* : \mathsf{L}(^{\xi_0}(t^* \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau')) \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau'))$

   (11) $\mathrm{D}^*;\bullet \vdash [\![\mathsf{raise}\ \psi]\!]^{\{\psi\}}_{\mathfrak{X}} : \mathsf{L}(^{\xi_0}(t^* \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau')) \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau'))$ — by (1, 10).

$$\textbf{Case}\ \frac{\mathrm{D} \vdash_{\mathbb{C}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_2 \qquad \mathrm{D};\mathrm{G}_1 \vdash_{\mathbb{X}} e_1 : t ; \{\psi\} \cup \Psi' \qquad \mathrm{D};\mathrm{G}_2 \vdash_{\mathbb{X}} e_2 : t ; \Psi' \qquad \mathrm{D} \vdash_{\mathbb{C}} \mathrm{G}_2 \preceq \mathsf{A}}{\mathrm{D};\mathrm{G} \vdash_{\mathbb{X}} e_1 \,\mathsf{handle}\, \psi \to e_2 : t ; \Psi'}.$$

By cases on Ψ':

Case $\Psi' = \varnothing$**.**

(1) $[\![e_1 \,\mathsf{handle}\, \psi \to e_2]\!]^{\varnothing}_{\mathbb{X}} = \lambda y.[\lambda\_.[\![e_2]\!]^{\varnothing}_{\mathbb{X}}\, y, y]([\![e_1]\!]^{\{\psi\}}_{\mathbb{X}}(\lambda x.\,\mathsf{inr}\, x))$ by def. $[\![e]\!]^{\Psi}_{\mathbb{X}}$

(2) $\mathrm{D}^*;{\mathrm{G}_2}^* \vdash [\![e_2]\!]^{\varnothing}_{\mathbb{X}} : \mathsf{L}(^{\xi_0}(t^* \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau')) \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau'))$ by IH, $\xi_0 \preceq \varnothing^*$

(3) $\mathrm{D}^*;\bullet, y{:}^{\xi_0}(t^* \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau')) \vdash y : {}^{\xi_0}(t^* \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau'))$ by rule T-Var

(4) $\mathrm{D}^*;{\mathrm{G}_2}^*, y{:}^{\xi_0}(t^* \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau')) \vdash [\![e_2]\!]^{\varnothing}_{\mathbb{X}}\, y : \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau')$ by (2–3), rule T-App

(5) $\mathrm{D}^*;{\mathrm{G}_2}^*, y{:}^{\xi_0}(t^* \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau')), x'{:}^{\mathsf{U}}\mathsf{exn} \vdash [\![e_2]\!]^{\varnothing}_{\mathbb{X}}\, y : \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau')$ by (4), rule T-Weak

(6) $\mathrm{D}^*;{\mathrm{G}_2}^*, y{:}^{\xi_0}(t^* \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau')) \vdash \lambda\_.[\![e_2]\!]^{\varnothing}_{\mathbb{X}}\, y : \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau'))$ by (5), rule T-Abs

(7) $\mathrm{D}^*;\mathrm{G}^*, y{:}^{\xi_0}(t^* \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau')) \vdash y : {}^{\xi_0}(t^* \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau'))$ by (3), rule T-Weak

(8) $\mathrm{D}^*;\mathrm{G}^*, y{:}^{\xi_0}(t^* \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau')) \vdash [\lambda\_.[\![e_2]\!]^{\varnothing}_{\mathbb{X}}\, y, y] : \mathsf{L}(^{\mathsf{L}}(^{\mathsf{U}}\mathsf{exn} \oplus t^*) \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus \tau'))$ by (6–7), rule T-Sum

(9) $\mathrm{D}^*;{\mathrm{G}_1}^* \vdash [\![e_1]\!]^{\{\psi\}}_{\mathbb{X}} : \mathsf{L}(^{\mathsf{A}}(t^* \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus t^*)) \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus t^*))$ by IH, $\mathsf{A} \preceq \{\psi\}^*$

(10) $\mathrm{D}^*;\bullet, x{:}t^* \vdash x : t^*$ by rule T-Var

(11) $\mathrm{D}^*;\bullet, x{:}t^* \vdash \mathsf{inr}\, x : \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus t^*)$ by (10), rule T-Inr

(12) $\mathrm{D}^*;\bullet \vdash \lambda x.\,\mathsf{inr}\, x : {}^{\mathsf{A}}(t^* \multimap \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus t^*))$ by (11), rule T-Abs

(13) $\mathrm{D}^*;{\mathrm{G}_1}^* \vdash [\![e_1]\!]^{\{\psi\}}_{\mathbb{X}}(\lambda x.\,\mathsf{inr}\, x) : \mathsf{L}(^{\mathsf{U}}\mathsf{exn} \oplus t^*)$ by (9, 12), rule T-App

(14) $\mathrm{D}^*;\mathrm{G}^*, y{:}^{\xi_0}(t^* \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{exn} \oplus \tau')) \vdash$
$[\lambda\_.[\![e_2]\!]_{\mathcal{X}}^{\varnothing}\, y, y]([\![e_1]\!]_{\mathcal{X}}^{\{\psi\}}(\lambda x.\,\mathsf{inr}\,x)) : {}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{exn} \oplus \tau')$
by (8, 13), rule T-App

(15) $\mathrm{D}^*;\mathrm{G}^* \vdash \lambda y.[\lambda\_.[\![e_2]\!]_{\mathcal{X}}^{\varnothing}\, y, y]([\![e_1]\!]_{\mathcal{X}}^{\{\psi\}}(\lambda x.\,\mathsf{inr}\,x)) : {}^{\mathsf{L}}(\xi_0(t^* \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{exn} \oplus \tau')) \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{exn} \oplus \tau'))$
by (14), rule T-Abs

(16) $\mathrm{D}^*;\mathrm{G}^* \vdash [\![e_1\ \mathsf{handle}\ \psi \to e_2]\!]_{\mathcal{X}}^{\varnothing} : {}^{\mathsf{L}}(\xi_0(t^* \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{exn} \oplus \tau')) \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{exn} \oplus \tau'))$
by (1, 15).

Case $\Psi' \neq \varnothing$**.**

This means that $\Psi'^* = \mathsf{A}$, so we know that

(1) $\mathrm{D}^* \vdash \xi_0 \preceq \mathsf{A}$.

Then:

(2) $[\![e_1\ \mathsf{handle}\ \psi \to e_2]\!]_{\mathcal{X}}^{\Psi'} =$
$\lambda y.[[\lambda\_.[\![e_2]\!]_{\mathcal{X}}^{\Psi'}\, y, \lambda x.\,\mathsf{inl}\,x]_{\psi}, y]([\![e_1]\!]_{\mathcal{X}}^{\{\psi\}\cup\Psi'}(\lambda x.\,\mathsf{inr}\,x))$
by def. $[\![e_1]\!]_{\mathcal{X}}^{\Psi}$

(3) $\mathrm{D}^*;\mathrm{G}_2{}^* \vdash [\![e_2]\!]_{\mathcal{X}}^{\Psi'} : {}^{\mathsf{L}}(\xi_0(t^* \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{exn} \oplus \tau')) \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{exn} \oplus \tau'))$
by IH, $\xi_0 \preceq \varnothing^*$

(4) $\mathrm{D}^*;\bullet, y{:}^{\xi_0}(t^* \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{exn} \oplus \tau')) \vdash y : {}^{\xi_0}(t^* \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{exn} \oplus \tau'))$
by rule T-Var

(5) $\mathrm{D}^*;\mathrm{G}_2{}^*, y{:}^{\xi_0}(t^* \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{exn} \oplus \tau')) \vdash [\![e_2]\!]_{\mathcal{X}}^{\Psi'}\, y : {}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{exn} \oplus \tau')$
by (3–4), rule T-App

(6) $\mathrm{D}^*;\mathrm{G}_2{}^*, y{:}^{\xi_0}(t^* \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{exn} \oplus \tau')), x'{:}^{\mathsf{U}}\mathsf{exn} \vdash [\![e_2]\!]_{\mathcal{X}}^{\Psi'}\, y :$
${}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{exn} \oplus \tau')$
by (5), rule T-Weak

(7) $\mathrm{D}^*;\mathrm{G}_2{}^*, y{:}^{\xi_0}(t^* \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{exn} \oplus \tau')) \vdash \lambda\_.[\![e_2]\!]_{\mathcal{X}}^{\Psi'}\, y : {}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{exn} \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{exn} \oplus \tau'))$
by (6), rule T-Abs

(8) $\mathrm{D}^*;\bullet \vdash \lambda x.\,\mathsf{inl}\,x : {}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{exn} \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{exn} \oplus \tau'))$
by rules T-Var, T-Inl, and T-Abs

(9) $\mathrm{D}^* \vdash \mathrm{G}_2{}^* \preceq \mathsf{A}$
by assumption, lemma C.6

(10) $\mathrm{D}^* \vdash \mathrm{G}_2{}^*, y{:}^{\xi_0}(t^* \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{exn} \oplus \tau')) \preceq \mathsf{A}$
by (1, 9)

(11) $\mathrm{D}^*;\mathrm{G}_2{}^*, y{:}^{\xi_0}(t^* \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\,\mathsf{exn} \oplus \tau')) \vdash \lambda x.\,\mathsf{inl}\,x : {}^{\mathsf{L}}({}^{\mathsf{U}}\,\mathsf{exn} \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\,\mathsf{exn} \oplus \tau'))$ by (8, 10), rule T-Weak

(12) $\mathrm{D}^*;\mathrm{G}_2{}^*, y{:}^{\xi_0}(t^* \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\,\mathsf{exn} \oplus \tau')) \vdash [\lambda\_.\,[\![e_2]\!]_{\mathrm{X}}^{\Psi'}\, y, \lambda x.\,\mathsf{inl}\,x]_{\psi} : {}^{\mathsf{L}}({}^{\mathsf{U}}\,\mathsf{exn} \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\,\mathsf{exn} \oplus \tau'))$ by (7, 11), def. $[v_1, v_2]_{\psi}$

(13) $\mathrm{D}^*;\mathrm{G}_2{}^*, y{:}^{\xi_0}(t^* \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\,\mathsf{exn} \oplus \tau')) \vdash y : {}^{\xi_0}(t^* \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\,\mathsf{exn} \oplus \tau'))$ by (4), rule T-Weak

(14) $\mathrm{D}^*;\mathrm{G}_2{}^*, y{:}^{\xi_0}(t^* \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\,\mathsf{exn} \oplus \tau')) \vdash [[\lambda\_.\,[\![e_2]\!]_{\mathrm{X}}^{\Psi'}\, y, \lambda x.\,\mathsf{inl}\,x]_{\psi}, y] : {}^{\mathsf{L}}({}^{\mathsf{L}}({}^{\mathsf{U}}\,\mathsf{exn} \oplus t^*) \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\,\mathsf{exn} \oplus \tau'))$ by (12–13), rule T-SumE

(15) $\mathrm{D}^*;\mathrm{G}_1{}^* \vdash [\![e_1]\!]_{\mathrm{X}}^{\{\psi\}\cup\Psi'} : {}^{\mathsf{L}}({}^{\mathsf{A}}(t^* \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\,\mathsf{exn} \oplus t^*)) \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\,\mathsf{exn} \oplus t^*))$ by IH, $\mathsf{A} \preceq (\{\psi\} \cup \Psi')^*$

(16) $\mathrm{D}^*;\bullet \vdash \lambda x.\,\mathsf{inr}\,x : {}^{\mathsf{A}}(t^* \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\,\mathsf{exn} \oplus t^*))$ by rules T-Var, T-Inr, and T-Abs

(17) $\mathrm{D}^*;\mathrm{G}_1{}^* \vdash [\![e_1]\!]_{\mathrm{X}}^{\{\psi\}\cup\Psi'}(\lambda x.\,\mathsf{inr}\,x) : {}^{\mathsf{L}}({}^{\mathsf{U}}\,\mathsf{exn} \oplus t^*)$ by (15–16), rule T-App

(18) $\mathrm{D}^*;\mathrm{G}^*, y{:}^{\xi_0}(t^* \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\,\mathsf{exn} \oplus \tau')) \vdash [[\lambda\_.\,[\![e_2]\!]_{\mathrm{X}}^{\Psi'}\, y, \lambda x.\,\mathsf{inl}\,x]_{\psi}, y]([\![e_1]\!]_{\mathrm{X}}^{\{\psi\}\cup\Psi'}(\lambda x.\,\mathsf{inr}\,x)) : {}^{\mathsf{L}}({}^{\mathsf{U}}\,\mathsf{exn} \oplus \tau')$ by (14, 17), rule T-App

(19) $\mathrm{D}^*;\mathrm{G}^* \vdash \lambda y.[[\lambda\_.\,[\![e_2]\!]_{\mathrm{X}}^{\Psi'}\, y, \lambda x.\,\mathsf{inl}\,x]_{\psi}, y]([\![e_1]\!]_{\mathrm{X}}^{\{\psi\}\cup\Psi'}(\lambda x.\,\mathsf{inr}\,x)) : {}^{\mathsf{L}}({}^{\xi_0}(t^* \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\,\mathsf{exn} \oplus \tau')) \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\,\mathsf{exn} \oplus \tau'))$ by (18), rule T-Abs

(20) $\mathrm{D}^*;\mathrm{G}^* \vdash [\![e_1\ \mathsf{handle}\ \psi \to e_2]\!]_{\mathrm{X}}^{\Psi'} : {}^{\mathsf{L}}({}^{\xi_0}(t^* \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\,\mathsf{exn} \oplus \tau')) \multimap {}^{\mathsf{L}}({}^{\mathsf{U}}\,\mathsf{exn} \oplus \tau'))$ by (2, 19).

List of Definitions and Propositions

Bibliography

Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(1–2):3–57, 1993.

Amal Ahmed, Matthew Fluet, and Greg Morrisett. A step-indexed model of substructural state. In *Proc. 10th ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, pages 78–91, Tallinn, Estonia, September 2005.

Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In *Proc. 3rd International Symposium on Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 1991.

Kenichi Asai and Yukiyoshi Kameyama. Polymorphic delimited continuations. In *Programming Languages and Systems*, volume 4807 of *Lecture Notes in Computer Science*, pages 239–254. Springer, 2007.

Andrew Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-960347, Laboratory for Foundations of Computer Science, University of Edinburgh, September 1996.

Paul S. Barth, Rishiyur S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In *Proc. 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'91)*, pages 538–568, Cambridge, Mass., August 1991.

P. N[ick] Benton. A mixed linear and non-linear logic: Proofs, terms and models. In *Computer Science Logic*, volume 933 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 1995.

G[avin] M. Bierman. *On Intuitionistic Linear Logic*. PhD thesis, University of Cambridge, August 1993.

Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. The ML Kit (version 1). DIKU-report 93/14, Department of Computer Science, University of Copenhagen, March 1993.

John Boyland. Checking interference with fractional permissions. In *Proc. 10th International Symposium on Static Analysis (SAS'03)*, pages 55–72, San Diego, Calif., June 2003.

T[om] Brus, M[arko] C. J. D. van Eekelen, M[aarten] van Leer, M[arinus] J. Plasmeijer, and H[enk] P. Barendregt. Clean: A language for functional graph rewriting. In *Proc. Conference on Functional Programming Languages and Computer Architecture (FPCA'87)*, Portland, Ore., September 1987.

Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proc. 21th International Conference on Concurrency Theory (CONCUR'10)*, pages 222–236, Paris, France, August 2010.

Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *Proc. 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pages 213–224, Victoria, B.C., Canada, September 2008.

William Clinger, ed. The revised revised report on Scheme or an UnCommon Lisp. AI Memo No. 848, MIT AI Lab, Cambridge, Mass., August 1985.

Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proc. 9th Annual ACM Symposium on Principles of Programming Languages (POPL'82)*, pages 207–212, Albuquerque, N.M., January 1982.

Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. Technical Report DIKU Rapport 89/12, Computer Science Department, University of Copenhagen, Denmark, 1989.

Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*, pages 59–69, Snowbird, Utah, May 2001.

Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proc. 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, pages 13–24, Berlin, Germany, June 2002.

Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proc. 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'06)*, pages 177–190, Leuven, Belgium, April 2006.

Matthias Felleisen. The theory and practice of first-class prompts. In *Proc. 15th Annual ACM Symposium on Principles of Programming Languages (POPL'88)*, pages 180–190, San Diego, Calif., January 1988.

Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proc. 7th ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, pages 48–59, Pittsburgh, Pa., September 2002.

Matthew Fluet, Greg Morrisett, and Amal Ahmed. Linear regions are all you need. In *Proc. 15th European Symposium on Programming (ESOP'06)*, pages 7–21, Vienna, Austria, March 2006.

Matthew Thomas Fluet. *Monadic and Substructural Type Systems for Region-Based Memory Management*. PhD thesis, Cornell University, January 2007.

Jean-Yves Girard. *Interprétation Fonctionelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieur*. Thèse de doctorat d'état, Université Paris VII, 1972.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

John B. Goodenough. Structured exception handling. In *Proc. 2th Annual ACM Symposium on Principles of Programming Languages (POPL'75)*, pages 204–224, Palo Alto, Calif., January 1975.

James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proc. 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, pages 282–293, Berlin, Germany, June 2002.

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Proc. 7th European Symposium on Programming (ESOP'98)*, pages 122–138, Lisbon, Portugal, March 1998.

Mark P. Jones. First-class polymorphism with type inference. In *Proc. 24th Annual ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 483–496, Paris, France, January 1997.

Oleg Kiselyov and Chung-chieh Shan. A substructural type system for delimited continuations. In *Proc. 8th International Conference on Typed Lambda Calculi and Applications (TLCA'07)*, pages 223–239, Paris, France, June 2007.

Peter J. Landin. A generalization of jumps and labels. Technical report, UNIVAC Systems Programming Research, 1965.

Didier Le Botlan and Didier Rémy. ML^F: Raising ML to the power of System F. In *Proc. 8th ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, pages 27–38, Uppsala, Sweden, September 2003.

Daan Leijen. First-class polymorphism with existential types. Unpublished manuscript, August 2006.

Daan Leijen. HMF: Simple type inference for first-class polymorphism. In *Proc. 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pages 283–294, Victoria, B.C., Canada, September 2008.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system, release 3.12*. Institut National de Recherche en Informatique et en Automatique, July 2011.

John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proc. 15th Annual ACM Symposium on Principles of Programming Languages (POPL'88)*, pages 47–57, San Diego, Calif., January 1988.

Jacob Matthews and Amal Ahmed. Parametric polymorphism through run-time sealing or, theorems for low, low prices! In *Proc. 17th European Symposium on Programming (ESOP'08)*, pages 16–31, Budapest, Hungary, March 2008.

Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *Proc. 34th Annual ACM Symposium on Principles of Programming Languages (POPL'07)*, pages 3–10, Nice, France, January 2007.

Karl Mazurak and Steve Zdancewic. Lolliproc: To concurrency from classical linear logic via Curry-Howard and control. In *Proc. 15th ACM SIGPLAN International Conference on Functional Programming (ICFP'10)*, pages 39–50, Baltimore, Md., September 2010.

Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. Lightweight linear types in System F°. In *Proc. 5th ACM SIGPLAN Workshop on Types in Language*

Design and Implementation (TLDI'10), pages 77–88, Madrid, Spain, January 2010.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML*. MIT Press, Cambridge, Mass., revised edition, 1997.

Greg Morrisett, Amal Ahmed, and Matthew Fluet. L^3: A linear language with locations. In *Proc. 7th International Conference on Typed Lambda Calculi and Applications (TLCA'05)*, pages 293–307, Nara, Japan, April 2005.

Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. In *Proc. 6th Workshop on Foundations of Object-Oriented Languages (FOOL'99)*, pages 35–55, San Antonio, Texas, January 1999.

Simon Peyton Jones, ed. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, 2003.

Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, Mass., 2002.

François Pottier. Wandering through linear types, capabilities, and regions. Survey talk given at INRIA, Rocquencourt, France, May 2007.

François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pages 387–490. MIT Press, Cambridge, 2005.

Riccardo Pucella and Alec Heller. Capability-based calculi for session types. Unpublished manuscript, 2008.

Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *Proc. 1st ACM SIGPLAN Symposium on Haskell (Haskell'08)*, pages 25–36, Victoria, B.C., Canada, September 2008.

John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proc. ACM Annual Conference*, volume 2, pages 717–470, Boston, Mass., August 1972.

Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. In *Proc. 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'10)*, pages 89–102, Madrid, Spain, January 2010.

Claudio V. Russo and Dimitrios Vytiniotis. QML: Explicit first-class polymorphism for ML. In *Proc. 2009 ACM SIGPLAN workshop on ML (ML'09)*, pages 3–14, Edinburgh, UK, August 2009.

Vincent Simonet. Type inference with structural subtyping: A faithful formalization of an efficient constraint solver. In *Proc. Asian Symposium on Programming Languages and Systems (APLAS'03)*, pages 283–302, Beijing, China, November 2003.

Guy Lewis Steele Jr. and Gerald Jay Sussman. Scheme: An interpreter for extended lambda calculus. AI Memo 349, MIT, 1975.

Martin Steffen. *Polarized Higher-Order Subtyping*. Dissertation zur Erlangung des Grades Doktor-Ingenieur, Technische Fakultät, Friedrich-Alexander-Universität Erlangen-Nürnberg, 1997.

W. Richard Stevens. *UNIX Network Programming*. Prentice-Hall, New Jersey, 1990.

Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, January 1986.

Ivan E. Sutherland and Gary W. Hodgman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, January 1974.

Nikhil Swamy, Juan Chen, and Ravi Chugh. Enforcing stateful authorization and information flow policies in Fine. In *Proc. 19th European Symposium on Programming (ESOP'10)*, pages 529–549, Paphos, Cyprus, March 2010.

Hayo Thielecke. From control effects to typed continuation passing. In *Proc. 30th Annual ACM Symposium on Principles of Programming Languages (POPL'03)*, pages 139–149, New Orleans, La., January 2003.

Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *Proc 21th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*, pages 964–974, Portland, Ore., October 2006.

Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *Proc. 35th Annual ACM Symposium on Principles of Programming Languages (POPL'08)*, pages 395–406, San Francisco, Calif., January 2008.

Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, February 1997.

Bernardo Toninho, Luís Caires, and Frank Pfenning. Dependent session types via intuitionistic linear type theory. In *Proc. 13th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'11)*, pages 161–172, Odense, Denmark, July 2011.

Jesse A. Tov and Riccardo Pucella. Stateful contracts for affine types. In *Proc. 19th European Symposium on Programming (ESOP'10)*, pages 550–569, Paphos, Cyprus, March 2010.

Jesse A. Tov and Riccardo Pucella. A theory of substructural types and control. In *Proc 26th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'11)*, pages 625–642, Portland, Ore., October 2011a.

Jesse A. Tov and Riccardo Pucella. Practical affine types. In *Proc. 38th Annual ACM Symposium on Principles of Programming Languages (POPL'11)*, pages 447–458, Austin, Texas, January 2011b.

Vasco Vasconcelos, Simon Gay, and António Ravara. Session types for functional multithreading. In *Proc. 15th International Conference on*

Concurrency Theory (CONCUR'04), pages 497–511, London, UK, August 2004.

Edsko de Vries, Rinus Plasmeijer, and David M Abrahamson. Uniqueness typing simplified. In *Proc. 19th Annual Workshop on Implementation and Application of Functional Languages (IFL'07)*, pages 201–218, Freiberg, Germany, September 2007.

Philip Wadler. Is there a use for linear logic? In *Proc. Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'91)*, pages 255–273, New Haven, CT, USA, June 1991.

Philip Wadler. There's no substitute for linear logic. In *Proc. 8th International Workshop on the Mathematical Foundations of Programming Semantics (MFPS'92)*, Oxford, UK, April 1992.

Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *Proc. 16th Annual ACM Symposium on Principles of Programming Languages (POPL'89)*, pages 60–76, Austin, Texas, January 1989.

David Walker, Karl Crary, and Greg Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, July 2000.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

Dengping Zhu and Hongwei Xi. Safe programming with pointers through stateful views. In *Proc. 8th International Symposium on Practical Aspects of Declarative Languages (PADL'05)*, pages 83–97, Long Beach, Calif., January 2005.