A Theory of Substructural Types and Control

Jesse A. Tov Riccardo Pucella
Northeastern University, Boston, Massachusetts, USA
{tov,riccardo}@ccs.neu.edu

Abstract

Exceptions are invaluable for structured error handling in high-level languages, but they are at odds with linear types. More generally, control effects may delete or duplicate portions of the stack, which, if we are not careful, can invalidate all substructural usage guarantees for values on the stack. We have developed a type-and-effect system that tracks control effects and ensures that values on the stack are never wrongly duplicated or dropped. We present the system first with abstract control effects and prove its soundness. We then give examples of three instantiations with particular control effects, including exceptions and delimited continuations, and show that they meet the soundness criteria for specific control effects.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages

1. Substructural Types and Control

Consider, for example, a language like Scala (Odersky and Zenger 2005) with mutable references and arithmetic. Here is a method that takes two integers and divides each by the other, returning a pair of references to their quotients:

$$\begin{array}{l}\mathsf{def}\ \mathit{divRef}(z_1\colon \mathrm{Int},\ z_2\colon \mathrm{Int}) = \\ \quad (\mathsf{new}\ \mathrm{Ref}(z_1\ /\ z_2),\ \mathsf{new}\ \mathrm{Ref}(z_2\ /\ z_1))\end{array}$$

Suppose that references in this language are *linear*, meaning that they cannot be duplicated, and must be explicitly deallocated rather than implicitly dropped. In such a language, divRef has a memory leak. Most uses of divRef are harmless, but consider the expression $\mathit{divRef}(0, 5)$. The method will raise a division-by-zero exception, but (assuming one reasonable evaluation order) only *after* it has allocated a reference to hold the result of the first division. Because the method raises an exception but does not return the successfully allocated reference, there is no way for recovery code that catches the exception to free the reference.

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 types, however, divRef is not a problem, because such a type system does not require that references be freed explicitly. In a language with affine types, implicitly dropping a value is just fine—presumably there is a garbage collector—and only duplication is forbidden. Consider, however, adding delimited continuation operators such as shift and reset to a language with affine types. Assuming a method unref that dereferences and deallocates a reference, we might attempt to define a method $\mathit{squareRef}$ that takes a reference to an integer, frees it, and returns its contents, squared:

$$\begin{array}{l}\mathsf{def}\ \mathit{twiceTo}(x\colon \mathrm{Int}) = \\ \quad \mathsf{shift}\ \{\ (k\colon \mathrm{Int} \Rightarrow \mathrm{Int}) \Rightarrow k(k(x))\ \} \\ \mathsf{def}\ \mathit{squareRef}(r\colon \mathrm{Ref[Int]}) = \\ \quad \mathsf{reset}\ \{\ \mathit{twiceTo}(1) \times r.\mathit{unref}()\ \}\end{array}$$

Method $\mathit{twiceTo}$ uses shift to capture its continuation up to the nearest enclosing reset, and it then applies the captured continuation k twice to the parameter x. Method $\mathit{squareRef}$ provides the context for $\mathit{twiceTo}$ to capture, which is to free r and multiply by its contents:

$$[\,] \times r.\mathit{unref}()\,.$$

Since $\mathit{twiceTo}$ uses its continuation twice, the second use of the continuation will access a dangling pointer that the first use freed.

Typically, an affine type system works by imposing two syntactic requirements: a variable of affine type, such as r, 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 $\mathit{squareRef}$ example violates neither dictum. In the presence of delimited continuations, we need to add 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 would permit some limited uses of delimited continuations, such as

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.

coroutines, but we will show that this simple rule is overly restrictive.

Our solution. The memory leak and dangling pointer in the above examples can be fixed by small changes to the code. For $divRef$, it suffices to ensure that both divisions happen before both allocations:

```
def divRef(z1: Int, z2: Int) = {
  val z12 = z1 / z2
  val z21 = z2 / z1
  (new Ref(z12), new Ref(z21))
}
```

For $squareRef$, we need the dereferencing to happen once, outside the reset delimiter:

```
def squareRef(r: Ref[Int]) = {
  val z = r.unref()
  reset { twiceTo(1) × z }
}
```

Unfortunately, the conservative approximation suggested above, that all continuations be treated linearly, would still disallow these repaired examples. We have designed a type-and-effect system (Lucassen and Gifford 1988) that permits these two repaired versions of the methods while forbidding the original, erroneous versions. 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 paper, we

- exhibit a *generic* type system for substructural types and control defined in terms of an unspecified, abstract control effect (§4);
- give soundness criteria for the abstract control effect and prove type safety for the generic system, relying on the soundness of the abstract control effect (§5); and
- demonstrate three concrete instantiations of control effects and prove that they meet the soundness criteria (§6).

The generic type-and-effect system in §4 is defined as an extension to λ^{URAL} (Ahmed et al. 2005), a substructural λ calculus, which we review in §3, after discussing related work in §2.

2. Related Work and Comparison

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 ours. He uses substructural types in his object language to reason about how continuations will be used in a non-substructural source language, whereas we 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 we are 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 Lolliproc (2010) relates double negation elimination in classical linear logic to delimited control.

We draw 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 *escape* (1972), and Scheme's *call/cc* (Clinger 1985). However, for integration in a language with substructural types, control operators with delimited extent, originating with Felleisen's $\mathcal{F}$ (1988), 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 our calculus, we 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) has checked exceptions, an effect system for tracking the exceptions that a method may raise. Our version of exception effects is similar to Java's, except that we offer effect polymorphism, which makes higher-order programming with checked exceptions more convenient. Our type system for exceptions appears in §6.3.

Because Danvy and Filinski's *shift* (1989) 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. We give two substructural type systems with *shift* and *reset*. Section 6.1 presents a simpler version that severely limits the answer types of continuations that may be captured. Then, in §6.2, we 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.

Substructural type systems. Researchers have proposed a plethora of substructural type systems. These range from minimalistic models (Wadler 1992; Bierman 1993; Barber 1996; Morrisett et al. 2005) based on Girard's linear logic (1987), to real programming languages, which are often oriented toward specific problems such as safety in low-level languages (Grossman et al. 2002), typestate and protocol checking (Aldrich et al. 2009), or security (Swamy et al. 2010).

We translate our substructural type-and-effect system into Ahmed et al.'s λ^{URAL} (2005), which is a polymorphic λ calculus that supports a variety of substructural typing disciplines. We provide a primer on λ^{URAL} in §3.

Motivation. The software engineering case for structured exception handling is widely acknowledged and understood, but *shift* and *reset* (Danvy and Filinski 1989), the other control operators discussed in this paper, are more obscure. The essential idea is simple: whereas raising an exception discards the context up to some delimiter—the exception handler—*shift* captures and reifies the context up to its delimiter, *reset*, which allows reinstating the context later. These control operators may be used to implement exceptions, by capturing continuations but never reinvoking them, but they may also express other control structures, such as coroutines and cooperative multithreading, and they may be used to abstract non-determinism and search in an elegant way.

Our goal is to safely integrate control operators with substructural types. A substructural type system regulates the order and number of uses of data by statically ensuring that some values be used at most once, at least once, or exactly once (Walker 2005). Like *shift* and *reset*, substructural types are a general facility that can express a variety of specific language features, mostly for the purpose of managing stateful resources, such as typestate, region-based memory management, and session types.

The direct impetus for this work is the design of the programming language Alms (Tov and Pucella 2011), which provides both exceptions and affine types, a variety of substructural type that can prohibit reusing particular values. As demonstrated in §1, the combination of affine types and exceptions is not a problem. However, as we observe in that previous work, "we anticipate that safely combining linearity with exceptions requires a type-and-effect system to track when raising an exception would implicitly discard linear values." Our desire to add linear types to Alms motivates this development of a general theory of substructural types and control effects.

3. Syntax and Semantics of λ^{URAL}

In this paper, we add control effects to Ahmed et al.'s λ^{URAL} (2005), a substructural λ calculus. Our presentation of λ^{URAL} is heavily based on theirs, with a few small changes.

$$
\begin{array}{lll}
v & ::= & x \mid \lambda x.e \mid \Lambda.e \mid \langle\rangle \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v & \textit{(values)} \\
e & ::= & v \mid e_1\, e_2 \mid e\_ \mid \mathsf{let}\,\langle\rangle = e_1\ \mathsf{in}\ e_2 & \textit{(expressions)} \\
 & & \mid \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x_1 \to e_1;\ \mathsf{inr}\ x_2 \to e_2 & \\
\mathsf{q} & \in & \{\mathsf{U}, \mathsf{R}, \mathsf{A}, \mathsf{L}\} & \textit{(qualifier constants)} \\
\xi & ::= & \alpha \mid \mathsf{q} & \textit{(qualifiers)} \\
\overline{\tau} & ::= & \alpha \mid \tau_1 \multimap \tau_2 \mid \forall\alpha{:}\kappa.\tau \mid 1 \mid \tau_1 \oplus \tau_2 & \textit{(pretypes)} \\
\tau & ::= & \alpha \mid {}^{\xi}\overline{\tau} & \textit{(types)} \\
\iota & ::= & \xi \mid \overline{\tau} \mid \tau & \textit{(type-level terms)} \\
\kappa & ::= & \mathsf{QUAL} \mid \overline{\star} \mid \star & \textit{(kinds)}
\end{array}
$$

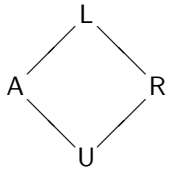
Figure 1. λ^{URAL} syntax

The syntax of λ^{URAL} appears in Figure 1.[1]

The expression level. Values include abstractions, type abstractions, the unit value, and injections into a sum. (This differs from Ahmed et al.'s presentation of λ^{URAL} by including sums—additive disjunctions, to be precise—rather than multiplicative conjunctions. Our theorems handle both, but we omit products in this paper for brevity. Sums are more useful for our purposes here.) Expressions include values, application, type application, unit elimination, and sum elimination. Following Ahmed et al., we 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), but the language at the type level is much richer. Four constant qualifiers (q) distinguish four substructural properties that may be enforced for values:

- L as in *linear*, for values that may be neither duplicated nor implicitly dropped;
- A as in *affine*, for values that may be dropped (weakening) but not duplicated;
- R as in *relevant*, for values that may be duplicated (contraction) but not dropped; and
- U as in *unlimited*, for ordinary values that allow both dropping and duplication.



The four constant qualifiers form a lattice, whereby it is always 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 ($\overline{\tau}$)

[1] This is the black-and-white version of this paper, suitable for printing; a version that uses colors to distinguish non-terminals from two different calculi is available online at www.ccs.neu.edu/~tov/pubs/substructural-control.

$$E ::= [] \mid E\,e_2 \mid v_1\,E \mid E\,\_ \quad \textit{(evaluation contexts)}$$
$$\mid \mathsf{let}\,\langle\rangle = E\ \mathsf{in}\ e_2$$
$$\mid \mathsf{case}\ E\ \mathsf{of}\ \mathsf{inl}\ x_1 \to e_1;\ \mathsf{inr}\ x_2 \to e_2$$

$\boxed{e \longmapsto e'}$ *(reduction)*

$$(\lambda x.e_1)\,v_2 \longmapsto \{v_2/x\}e_2$$
$$(\Lambda.e)\,\_ \longmapsto e$$
$$\mathsf{let}\,\langle\rangle = \langle\rangle\ \mathsf{in}\ e \longmapsto e$$
$$\mathsf{case}\ \mathsf{inl}\ v\ \mathsf{of}\ \mathsf{inl}\ x_1 \to e_1;\ \mathsf{inr}\ x_2 \to e_2 \longmapsto \{v/x_1\}e_1$$
$$\mathsf{case}\ \mathsf{inr}\ v\ \mathsf{of}\ \mathsf{inl}\ x_1 \to e_1;\ \mathsf{inr}\ x_2 \to e_2 \longmapsto \{v/x_2\}e_2$$
$$\frac{e \longmapsto e'}{E[e] \longmapsto E[e']}$$

Figure 2. λ^{URAL} operational semantics

$\boxed{\Delta \vdash \iota : \kappa}$ *(kinding type-level terms)*

K-VAR
$$\frac{\alpha{:}\kappa \in \Delta}{\Delta \vdash \alpha : \kappa}$$

K-QUAL
$$\frac{}{\Delta \vdash \mathsf{q} : \mathsf{QUAL}}$$

K-ARR
$$\frac{\Delta \vdash \tau_1 : \star \qquad \Delta \vdash \tau_2 : \star}{\Delta \vdash \tau_1 \multimap \tau_2 : \overline{\star}}$$

K-ALL
$$\frac{\Delta, \alpha{:}\kappa \vdash \tau : \star}{\Delta \vdash \forall\alpha{:}\kappa.\tau : \overline{\star}}$$

K-UNIT
$$\frac{}{\Delta \vdash 1 : \overline{\star}}$$

K-SUM
$$\frac{\Delta \vdash \tau_1 : \star \qquad \Delta \vdash \tau_2 : \star}{\Delta \vdash \tau_1 \oplus \tau_2 : \overline{\star}}$$

K-TYPE
$$\frac{\Delta \vdash \overline{\tau} : \overline{\star} \qquad \Delta \vdash \xi : \mathsf{QUAL}}{\Delta \vdash {}^{\xi}\overline{\tau} : \star}$$

Figure 3. λ^{URAL} statics (i): kinding

$\boxed{\Delta \vdash \xi_1 \preceq \xi_2}$ *(qualifier subsumption)*

QSUB-BOT
$$\frac{\Delta \vdash \xi : \mathsf{QUAL}}{\Delta \vdash \mathsf{U} \preceq \xi}$$

QSUB-TOP
$$\frac{\Delta \vdash \xi : \mathsf{QUAL}}{\Delta \vdash \xi \preceq \mathsf{L}}$$

QSUB-REFL
$$\frac{\Delta \vdash \xi : \mathsf{QUAL}}{\Delta \vdash \xi \preceq \xi}$$

$\boxed{\Delta \vdash \tau \preceq \xi}$ *(qualifier bound for types)*

B-VAR
$$\frac{\Delta \vdash \alpha : \star}{\Delta \vdash \alpha \preceq \mathsf{L}}$$

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)*

B-NIL
$$\frac{\Delta \vdash \xi : \mathsf{QUAL}}{\Delta \vdash \bullet \preceq \xi}$$

B-CONS
$$\frac{\Delta \vdash \Gamma \preceq \xi \qquad \Delta \vdash \tau \preceq \xi}{\Delta \vdash \Gamma, x{:}\tau \preceq \xi}$$

Figure 4. λ^{URAL} statics (ii): qualifiers

$\boxed{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2}$ *(type context splitting)*

S-NIL
$$\frac{}{\Delta \vdash \bullet \rightsquigarrow \bullet \boxplus \bullet}$$

S-CONSL
$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Delta \vdash \tau : \star}{\Delta \vdash \Gamma, x{:}\tau \rightsquigarrow (\Gamma_1, x{:}\tau) \boxplus \Gamma_2}$$

S-CONSR
$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Delta \vdash \tau : \star}{\Delta \vdash \Gamma, x{:}\tau \rightsquigarrow \Gamma_1 \boxplus (\Gamma_2, x{:}\tau)}$$

S-CONTRACT
$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Delta \vdash \tau \preceq \mathsf{R}}{\Delta \vdash \Gamma, x{:}\tau \rightsquigarrow (\Gamma_1, x{:}\tau) \boxplus (\Gamma_2, x{:}\tau)}$$

Figure 5. λ^{URAL} statics (iii): context splitting

specify the representation of a value, and its introduction and elimination rules. Pretypes include type variables, function types, universal quantification, the unit type, and additive disjunction. Types (τ) classify expressions. A type is either a pretype decorated with its qualifier (${}^{\xi}\overline{\tau}$) or a type variable. We use non-terminal ι to refer to the 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.

3.1 Operational Semantics

The operational semantics of λ^{URAL} is completely standard and appears in Figure 2. Reduction is call-by-value and evaluates operators before operands, which is important when we consider the sequencing of effects in §4.

3.2 Static Semantics

Type judgments for λ^{URAL} use two kinds of contexts:

$$\Delta ::= \bullet \mid \Delta, \alpha{:}\kappa \quad \textit{(kind contexts)}$$
$$\Gamma ::= \bullet \mid \Gamma, x{:}\tau \quad \textit{(type contexts)}$$

Figure 3 contains the kinding judgment ($\Delta \vdash \iota : \kappa$), which assigns kinds to type-level terms. This judgment

$\boxed{\Delta;\Gamma \vdash e : \tau}$ *(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-UNIT
$$\frac{\Delta \vdash \xi : \mathsf{QUAL}}{\Delta;\bullet \vdash \langle\rangle : {}^{\xi}1}$$

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-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-LETUNIT
$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Delta;\Gamma_1 \vdash e_1 : {}^{\xi}1 \qquad \Delta;\Gamma_2 \vdash e_2 : \tau}{\Delta;\Gamma \vdash \mathsf{let}\ \langle\rangle = e_1\ \mathsf{in}\ e_2 : \tau}$$

T-CASE
$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxplus \Gamma_2 \qquad \Delta;\Gamma_1 \vdash e : {}^{\xi}(\tau_1 \oplus \tau_2) \qquad \Delta;\Gamma_2, x_1{:}\tau_1 \vdash e_1 : \tau \qquad \Delta;\Gamma_2, x_2{:}\tau_2 \vdash e_2 : \tau}{\Delta;\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x_1 \to e_1;\ \mathsf{inr}\ x_2 \to e_2 : \tau}$$

Figure 6. λ^{URAL} statics (iv): typing

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 4, 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.

Figure 5 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, Figure 6 gives 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 and for unit and sum elimination, T-APP, T-LETUNIT, and T-CASE, split the context to distribute assumptions to subterms. Note, however, that both branches of a case expression share the same context.
- 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 §5, this property is key to our soundness theorem.

4. Generic Control Effects

Rather than add a specific control effect, such as exceptions or delimited continuations, to λ^{URAL}, we aim to design a substructural type system with a general notion of control effect. Thus, in this section, we define a new calculus, $\lambda^{\mathsf{URAL}}(\mathcal{C})$, parameterized by an unspecified control effect.

4.1 The Control Effect Parameter

In this subsection, we give the form of the parameter that stands for a particular control effect. Our definition of $\lambda^{\mathsf{URAL}}(\mathcal{C})$ relies only on this abstract specification of the formal parameter. In §5, we specify several properties of the parameter that are sufficient for a generic soundness theorem to hold, and in §6 we give three examples of actual control effect parameters.

Definition 4.1 (Control effect)**.**
A control effect instance is a triple $(\mathcal{C}, \bot_{\mathrm{e}}, \circledcirc)$ *where* $\mathcal{C}$ *is a set of control effects (c),* $\bot_{\mathrm{e}} \in \mathcal{C}$ *is a distinguished* pure *effect that denotes no actual control, and* $\circledcirc\colon \mathcal{C} \times \mathcal{C} \rightharpoonup \mathcal{C}$ *is an associative, partial, binary operation denoting effect sequencing.*

$\boxed{\mathrm{D} \vDash_{\mathrm{e}} i : k}$ *(kinding type-level terms)*

$$\textsc{C-K-Bot}\ \frac{}{\mathrm{D} \vDash_{\mathrm{e}} \bot_{\mathrm{e}} : \mathsf{CTL}}$$

$$\textsc{C-K-Arr}\ \frac{\mathrm{D} \vDash_{\mathrm{e}} t_1 : \star \qquad \mathrm{D} \vDash_{\mathrm{e}} t_2 : \star \qquad \mathrm{D} \vDash_{\mathrm{e}} c : \mathsf{CTL}}{\mathrm{D} \vDash_{\mathrm{e}} t_1 \xrightarrow{c}\!\circ\ t_2 : \overline{\star}}$$

$$\textsc{C-K-All}\ \frac{\mathrm{D}, \alpha{:}k \vDash_{\mathrm{e}} t : \star \qquad \mathrm{D} \vDash_{\mathrm{e}} c : \mathsf{CTL}}{\mathrm{D} \vDash_{\mathrm{e}} \forall^{c}\alpha{:}k.t : \overline{\star}}$$

Figure 7. $\lambda^{\mathsf{URAL}}(\mathcal{C})$ statics (i): updated kinding rules

$\boxed{\mathrm{D} \vDash_{\mathrm{e}} c \succeq \xi}$ *(qualifier bound for control effects)*

$$\textsc{C-B-Pure}\ \frac{\mathrm{D} \vDash_{\mathrm{e}} \xi : \mathsf{QUAL}}{\mathrm{D} \vDash_{\mathrm{e}} \bot_{\mathrm{e}} \succeq \xi} \qquad \textsc{C-B-Unl}\ \frac{\mathrm{D} \vDash_{\mathrm{e}} c : \mathsf{CTL}}{\mathrm{D} \vDash_{\mathrm{e}} c \succeq \mathsf{U}}$$

$\boxed{\mathrm{D} \vDash_{\mathrm{e}} c_1 \preceq c_2}$ *(control effect subsumption)*

$$\textsc{CSub-Refl}\ \frac{\mathrm{D} \vDash_{\mathrm{e}} c : \mathsf{CTL}}{\mathrm{D} \vDash_{\mathrm{e}} c \preceq c} \qquad \textsc{CSub-Trans}\ \frac{\mathrm{D} \vDash_{\mathrm{e}} c_1 \preceq c' \qquad \mathrm{D} \vDash_{\mathrm{e}} c' \preceq c_2}{\mathrm{D} \vDash_{\mathrm{e}} c_1 \preceq c_2}$$

Figure 8. $\lambda^{\mathsf{URAL}}(\mathcal{C})$ statics (ii): control effect judgments

For example, in §6.3 we add 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 $\bot_{\mathrm{e}}$ 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 (§6.2).

4.2 Updated Syntax

In $\lambda^{\mathsf{URAL}}(\mathcal{C})$, control effects constitute a fourth kind of type-level term, in addition to qualifiers, pretypes, and types. We add a new kind, CTL, and include abstract control effects ($c \in \mathcal{C}$) among the type-level terms:

$$\begin{array}{lll} k & ::= & \mathsf{QUAL} \mid \overline{\star} \mid \star \mid \mathsf{CTL} \qquad \textit{(kinds)} \\ i & ::= & \xi \mid \overline{t} \mid t \mid c \qquad \textit{(type-level terms)} \end{array}$$

Function and universal pretypes now have latent effects, which record the effect that will happen when an abstraction is applied. We update the definition of pretypes to include these latent effects:

$$\begin{array}{lll} \overline{t} & ::= & \cdots \mid t_1 \xrightarrow{c}\!\circ\ t_2 \mid \forall^{c}\alpha{:}k.t \qquad \textit{(pretypes)} \\ t & ::= & \alpha \mid {}^{\xi}\overline{t} \qquad \textit{(types)} \end{array}$$

The other pretype ($\overline{t}$) productions remain unchanged.

For non-terminal symbols that differ between λ^{URAL} and $\lambda^{\mathsf{URAL}}(\mathcal{C})$, we 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, ...).

4.3 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 ($\mathrm{D} \vDash_{\mathrm{e}} t \preceq \xi$), bounding type contexts $\mathrm{D} \vDash_{\mathrm{e}} \mathrm{G} \preceq \xi$), and splitting type contexts ($\mathrm{D} \vDash_{\mathrm{e}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_2$) are isomorphic to the λ^{URAL} versions of those judgments from Figures 4 and 5. 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. We identify control effects as the type-level terms (i) that are assigned kind CTL by the kinding judgment. Figure 7 shows one new kinding rule, C-K-Bot, which assigns kind CTL to the pure effect $\bot_{\mathrm{e}}$. We update rules C-K-Arr and C-K-All 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 (§6) must define additional kinding rules for their particular effects.

Control effect judgments. The first new judgment for control effects ($\mathrm{D} \vDash_{\mathrm{e}} c \succeq \xi$, Figure 8) relates control effects to qualifiers. This gives the meaning of a control effect in terms of a lower bound for how an expression with that effect may treat its own continuation. For example, if an expression e has some effect c such that $\mathrm{D} \vDash_{\mathrm{e}} c \succeq \mathsf{A}$, this indicates that e may drop but not duplicate its continuation. We give two rules 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.
- Rule C-B-Unl says that all control effects are bounded by U, which means that we 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 ($\mathrm{D} \vDash_{\mathrm{e}} c_1 \preceq c_2$) 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

$\boxed{\mathrm{D};\mathrm{G} \vdash_{\mathrm{e}} e : t \,;\, c}$ *(typing expressions)*

C-T-Subsume
$$\frac{\mathrm{D};\mathrm{G} \vdash_{\mathrm{e}} e : t \,;\, c' \qquad \mathrm{D} \vdash_{\mathrm{e}} c' \preceq c}{\mathrm{D};\mathrm{G} \vdash_{\mathrm{e}} e : t \,;\, c}$$

C-T-Weak
$$\frac{\mathrm{D} \vdash_{\mathrm{e}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_2 \qquad \mathrm{D};\mathrm{G}_1 \vdash_{\mathrm{e}} e : t \,;\, c \qquad \mathrm{D} \vdash_{\mathrm{e}} \mathrm{G}_2 \preceq \mathsf{A}}{\mathrm{D};\mathrm{G} \vdash_{\mathrm{e}} e : t \,;\, c}$$

C-T-Var
$$\frac{\mathrm{D} \vdash_{\mathrm{e}} t : \star}{\mathrm{D};\bullet, x{:}t \vdash_{\mathrm{e}} x : t \,;\, \bot_{\mathrm{e}}}$$

C-T-Abs
$$\frac{\mathrm{D} \vdash_{\mathrm{e}} \mathrm{G} \preceq \xi \qquad \mathrm{D};\mathrm{G}, x{:}t_1 \vdash_{\mathrm{e}} e : t_2 \,;\, c}{\mathrm{D};\mathrm{G} \vdash_{\mathrm{e}} \lambda x.e : {}^{\xi}(t_1 \xrightarrow{c} t_2) \,;\, \bot_{\mathrm{e}}}$$

C-T-TAbs
$$\frac{\mathrm{D} \vdash_{\mathrm{e}} \mathrm{G} \preceq \xi \qquad \mathrm{D}, \alpha{:}k;\mathrm{G} \vdash_{\mathrm{e}} e : t \,;\, c}{\mathrm{D};\mathrm{G} \vdash_{\mathrm{e}} \Lambda.e : {}^{\xi}\forall^{c}\alpha{:}k.t \,;\, \bot_{\mathrm{e}}}$$

C-T-Unit
$$\frac{\mathrm{D} \vdash_{\mathrm{e}} \xi : \mathsf{QUAL}}{\mathrm{D};\bullet \vdash_{\mathrm{e}} \langle\rangle : {}^{\xi}1 \,;\, \bot_{\mathrm{e}}}$$

C-T-Inl
$$\frac{\mathrm{D} \vdash_{\mathrm{e}} t_1 \preceq \xi \qquad \mathrm{D} \vdash_{\mathrm{e}} t_2 : \star \qquad \mathrm{D};\mathrm{G} \vdash_{\mathrm{e}} v_1 : t_1 \,;\, \bot_{\mathrm{e}}}{\mathrm{D};\mathrm{G} \vdash_{\mathrm{e}} \mathsf{inl}\ v_1 : {}^{\xi}(t_1 \oplus t_2) \,;\, \bot_{\mathrm{e}}}$$

C-T-Inr
$$\frac{\mathrm{D} \vdash_{\mathrm{e}} t_1 : \star \qquad \mathrm{D} \vdash_{\mathrm{e}} t_2 \preceq \xi \qquad \mathrm{D};\mathrm{G} \vdash_{\mathrm{e}} v_2 : t_2 \,;\, \bot_{\mathrm{e}}}{\mathrm{D};\mathrm{G} \vdash_{\mathrm{e}} \mathsf{inr}\ v_2 : {}^{\xi}(t_1 \oplus t_2) \,;\, \bot_{\mathrm{e}}}$$

C-T-App
$$\frac{\begin{array}{c}\mathrm{D};\mathrm{G}_1 \vdash_{\mathrm{e}} e_1 : {}^{\xi_1}(t_1 \xrightarrow{c} t_2) \,;\, c_1 \qquad \mathrm{D};\mathrm{G}_2 \vdash_{\mathrm{e}} e_2 : t_1 \,;\, c_2 \\ \mathrm{D} \vdash_{\mathrm{e}} \mathrm{G}_2 \preceq \xi_2 \qquad \mathrm{D} \vdash_{\mathrm{e}} c_1 \succeq \xi_2 \qquad \mathrm{D} \vdash_{\mathrm{e}} c_2 \succeq \xi_1 \\ \mathrm{D} \vdash_{\mathrm{e}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_2 \qquad \mathrm{D} \vdash_{\mathrm{e}} c_1 \otimes c_2 \otimes c : \mathsf{CTL}\end{array}}{\mathrm{D};\mathrm{G} \vdash_{\mathrm{e}} e_1\, e_2 : t_2 \,;\, c_1 \otimes c_2 \otimes c}$$

C-T-TApp
$$\frac{\mathrm{D};\mathrm{G} \vdash_{\mathrm{e}} e : {}^{\xi}\forall^{c'}\alpha{:}k.t \,;\, c \qquad \mathrm{D} \vdash_{\mathrm{e}} i : k \qquad \mathrm{D} \vdash_{\mathrm{e}} c \otimes c' : \mathsf{CTL}}{\mathrm{D};\mathrm{G} \vdash_{\mathrm{e}} e\,\_ : \{i/\alpha\}t \,;\, c \otimes c'}$$

C-T-LetUnit
$$\frac{\begin{array}{c}\mathrm{D};\mathrm{G}_1 \vdash_{\mathrm{e}} e_1 : {}^{\xi_1}1 \,;\, c_1 \qquad \mathrm{D};\mathrm{G}_2 \vdash_{\mathrm{e}} e_2 : t \,;\, c_2 \\ \mathrm{D} \vdash_{\mathrm{e}} \mathrm{G}_2 \preceq \xi_2 \qquad \mathrm{D} \vdash_{\mathrm{e}} c_1 \succeq \xi_2 \\ \mathrm{D} \vdash_{\mathrm{e}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_2 \qquad \mathrm{D} \vdash_{\mathrm{e}} c_1 \otimes c_2 : \mathsf{CTL}\end{array}}{\mathrm{D};\mathrm{G} \vdash_{\mathrm{e}} \mathsf{let}\,\langle\rangle = e_1\ \mathsf{in}\ e_2 : t \,;\, c_1 \otimes c_2}$$

C-T-Case
$$\frac{\begin{array}{c}\mathrm{D};\mathrm{G}_1 \vdash_{\mathrm{e}} e : {}^{\xi_1}(t_1 \oplus t_2) \,;\, c \\ \mathrm{D};\mathrm{G}_2, x_1{:}t_1 \vdash_{\mathrm{e}} e_1 : t \,;\, c' \qquad \mathrm{D};\mathrm{G}_2, x_2{:}t_2 \vdash_{\mathrm{e}} e_2 : t \,;\, c' \\ \mathrm{D} \vdash_{\mathrm{e}} \mathrm{G}_2 \preceq \xi_2 \qquad \mathrm{D} \vdash_{\mathrm{e}} c \succeq \xi_2 \qquad \mathrm{D} \vdash_{\mathrm{e}} c' \succeq \xi_1 \\ \mathrm{D} \vdash_{\mathrm{e}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_2 \qquad \mathrm{D} \vdash_{\mathrm{e}} c \otimes c' : \mathsf{CTL}\end{array}}{\mathrm{D};\mathrm{G} \vdash_{\mathrm{e}} \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x_1 \to e_1;\ \mathsf{inr}\ x_2 \to e_2 : t \,;\, c \otimes c'}$$

Figure 9. $\lambda^{\mathsf{URAL}}(\mathcal{C})$ statics (iii): typing

appear in Figure 8, 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 $\lambda^{\mathsf{URAL}}(\mathcal{C})$ (Figure 9) assigns not only a type t but an effect c to expressions: $\mathrm{D};\mathrm{G} \vdash_{\mathrm{e}} e : t \,;\, c$. 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:

$$\frac{\begin{array}{cc}(1) & \mathrm{D} \vdash_{\mathrm{e}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_2 \\ (2) & \mathrm{D};\mathrm{G}_1 \vdash_{\mathrm{e}} e_1 : {}^{\xi_1}(t_1 \xrightarrow{c} t_2) \,;\, c_1 \\ (3) & \mathrm{D};\mathrm{G}_2 \vdash_{\mathrm{e}} e_2 : t_1 \,;\, c_2 \\ (4) & \mathrm{D} \vdash_{\mathrm{e}} c_2 \succeq \xi_1 \\ (5) & \mathrm{D} \vdash_{\mathrm{e}} \mathrm{G}_2 \preceq \xi_2 \\ (6) & \mathrm{D} \vdash_{\mathrm{e}} c_1 \succeq \xi_2 \\ (7) & \mathrm{D} \vdash_{\mathrm{e}} c_1 \otimes c_2 \otimes c : \mathsf{CTL}\end{array}}{\mathrm{D};\mathrm{G} \vdash_{\mathrm{e}} e_1\, e_2 : t_2 \,;\, c_1 \otimes c_2 \otimes c}$$

We 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.

(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 we fix a 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, we require 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): $c_1 \otimes c_2 \otimes c$. 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-LETUNIT and C-T-CASE (unit and sum elimination) 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.

5. The Generic Theory

To prove type safety for $\lambda^{\mathsf{URAL}}(\mathcal{C})$, we define a type-preserving translation to λ^{URAL}. Rather than provide a reduction semantics for $\lambda^{\mathsf{URAL}}(\mathcal{C})$, we define its operational semantics in terms of the translation and the reduction semantics of λ^{URAL} (§3.1). Thus, if we can show that all well-typed $\lambda^{\mathsf{URAL}}(\mathcal{C})$ programs translate to well-typed λ^{URAL} programs, then λ^{URAL}'s type safety theorem applies 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 specified later in this section, we impose several more requirements on the abstract control effect parameter. As the semantics of $\lambda^{\mathsf{URAL}}(\mathcal{C})$ was parameterized by an abstract control effect, so is the theory of $\lambda^{\mathsf{URAL}}(\mathcal{C})$ parameterized by several definitions and properties that a control effect must satisfy.

The development of this section is constrained by several dependencies, so we provide an outline:

The Translation Parameter (§5.1). A control effect instance must supply a few definitions to fully specify its particular CCoS translation.

The Translation (§5.2). The definition of the CCoS translation relies on the definitions supplied by the control effect parameter.

Parameter Properties (§5.3). A control effect instance must satisfy several properties on which the generic type safety theorem relies.

Generic Type Safety (§5.4). The section culminates in a generic proof of type safety for $\lambda^{\mathsf{URAL}}(\mathcal{C})$.

5.1 The Translation Parameter

Definition 5.1 (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* $\lambda^{\mathsf{URAL}}(\mathcal{C})$ *effect to a* λ^{URAL} *type.*

Intuitively, we can 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, our translation 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, we 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*.

5.2 The Translation

In this subsection, we specify the CCoS translation from $\lambda^{\mathsf{URAL}}(\mathcal{C})$ to λ^{URAL}. In several places, we rely 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 10. 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.

$$
\begin{aligned}
\mathsf{QUAL}^* &= \mathsf{QUAL} && \textit{(kinds)}\\
\overline{\star}^* &= \overline{\star}\\
\star^* &= \star\\
\mathsf{CTL}^* &= \mathsf{QUAL}\\
\bullet^* &= \bullet && \textit{(kind contexts)}\\
(\mathrm{D}, \alpha{:}k)^* &= \mathrm{D}^*, \alpha{:}k^*
\end{aligned}
$$

Figure 10. CCoS translation (i): kinds and kind contexts

$$
\begin{aligned}
\alpha^* &= \alpha && \textit{(pretypes)}\\
1^* &= 1\\
(t_1 \oplus t_2)^* &= {t_1}^* \oplus {t_2}^*\\
(t_1 \xrightarrow{c} t_2)^* &=\\
&\quad \forall\alpha{:}\star.\,{}^{\mathsf{L}}({t_1}^* \multimap {}^{\mathsf{L}}({c^*}\,({t_2}^* \multimap \langle\!\langle \alpha, c \rangle\!\rangle_{\mathrm{e}}^{-}) \multimap \langle\!\langle \alpha, c \rangle\!\rangle_{\mathrm{e}}^{+}))\\
(\forall^{c}\beta{:}k.\,t)^* &=\\
&\quad \forall\alpha{:}\star.\,{}^{\mathsf{L}}\forall\beta{:}k^*.\,{}^{\mathsf{L}}({c^*}\,(t^* \multimap \langle\!\langle \alpha, c \rangle\!\rangle_{\mathrm{e}}^{-}) \multimap \langle\!\langle \alpha, c \rangle\!\rangle_{\mathrm{e}}^{+})\\
\alpha^* &= \alpha && \textit{(types)}\\
({}^{\xi}\overline{t})^* &= {}^{\xi}\overline{t}^*\\
\bullet^* &= \bullet && \textit{(type contexts)}\\
(\mathrm{G}, x{:}t)^* &= \mathrm{G}^*, x{:}t^*
\end{aligned}
$$

Figure 11. CCoS translation (ii): type-level terms and contexts

Figure 11 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 types translate both disjuncts, 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.
- 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 12. 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 $[\![e]\!]_{\mathrm{e}}$. 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 done explicitly using η expansion:

Lemma 5.2 (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 5.2 relies on another lemma:

Lemma 5.3 (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. By induction on the typing derivation for v. □

Proof of Lemma 5.2. Choose type contexts Γ_1 and Γ_2 according to Lemma 5.3. 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. □

Operational semantics of $\lambda^{\mathsf{URAL}}(\mathcal{C})$**.** Having defined the translation, we run a program e by applying the CCoS translation and passing it the initial continuation $\mathsf{done}_{\mathrm{e}}$. We define the operational semantics of $\lambda^{\mathsf{URAL}}(\mathcal{C})$ as a partial func-

$$
\begin{aligned}
x^* &= x && \textit{(values)}\\
(\lambda x.e)^* &= \Lambda.\lambda x.\ [\![e]\!]_{\mathrm{e}}\\
(\Lambda.e)^* &= \Lambda.\Lambda.\ [\![e]\!]_{\mathrm{e}}\\
(\mathsf{inl}\ v)^* &= \mathsf{inl}\ v^*\\
(\mathsf{inr}\ v)^* &= \mathsf{inr}\ v^*\\
\langle\rangle^* &= \langle\rangle
\end{aligned}
$$

$$
\begin{aligned}
[\![v]\!]_{\mathrm{e}} &= \lambda y.\, y\, v^* && \textit{(expressions)}\\
[\![e_1\, e_2]\!]_{\mathrm{e}} &= \lambda y.\ [\![e_1]\!]_{\mathrm{e}}\ (\lambda x_1.\ [\![e_2]\!]_{\mathrm{e}}\ (\lambda x_2.x_1\ \_\ x_2\ (\lambda x.\, y\, x)))\\
[\![e_1\ \_]\!]_{\mathrm{e}} &= \lambda y.\ [\![e_1]\!]_{\mathrm{e}}\ (\lambda x_1.x_1\ \_\ \_\ (\lambda x.\, y\, x))
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathsf{let}\ \langle\rangle = e_1\ \mathsf{in}\ e_2]\!]_{\mathrm{e}}\\
&\qquad = \lambda y.\ [\![e_1]\!]_{\mathrm{e}}\ (\lambda x_1.\ \mathsf{let}\ \langle\rangle = x_1\ \mathsf{in}\ [\![e_2]\!]_{\mathrm{e}}\ y)
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x_1 \to e_1;\ \mathsf{inr}\ x_2 \to e_2]\!]_{\mathrm{e}}\\
&\qquad = \lambda y.\ [\![e]\!]_{\mathrm{e}}\ (\lambda x.\ \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\ x_1\ \to\ [\![e_1]\!]_{\mathrm{e}}\ y;\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{inr}\ x_2\ \to\ [\![e_2]\!]_{\mathrm{e}}\ y)
\end{aligned}
$$

Figure 12. CCoS translation (iii): values and expressions

tion $eval : \mathit{Expressions} \rightharpoonup \mathit{Values} \cup \{\textsc{Wrong}\}$:

$$
eval(e) = \begin{cases} v & \text{if } [\![e]\!]_{\mathrm{e}}\ \mathsf{done}_{\mathrm{e}} \xmapsto{*} v;\\ \textsc{Wrong} & \text{if } [\![e]\!]_{\mathrm{e}}\ \mathsf{done}_{\mathrm{e}} \xmapsto{*} e' \\ & \quad \text{such that } e' \text{ is not a value} \\ & \quad \text{and } \neg\exists e''.e' \longmapsto e''. \end{cases}
$$

5.3 Parameter Properties

Having defined the CCoS translation, we are now ready to state the additional properties that the abstract control effect parameter must satisfy for the generic type safety theorem (§5.4) to hold:

Parameter Property 1 (Answer types)**.**

1. *For all* τ, $\langle\!\langle \tau, \bot_{\mathrm{e}} \rangle\!\rangle^-_{\mathrm{e}} = \langle\!\langle \tau, \bot_{\mathrm{e}} \rangle\!\rangle^+_{\mathrm{e}}$.
   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* $\langle\!\langle \tau \rangle\!\rangle_{\mathrm{e}} \triangleq \langle\!\langle \tau, \bot_{\mathrm{e}} \rangle\!\rangle^+_{\mathrm{e}}$.
2. *If* $\mathrm{D}^* \vdash \tau : \star$ *and* $\mathrm{D} \models_{\mathrm{e}} c : \mathsf{CTL}$ *then* $\mathrm{D}^* \vdash \langle\!\langle \tau, c \rangle\!\rangle^-_{\mathrm{e}} : \star$ *and* $\mathrm{D}^* \vdash \langle\!\langle \tau, c \rangle\!\rangle^+_{\mathrm{e}} : \star$.
   RATIONALE. For the translation to be well typed, well-kinded types and effects must become well-kinded answer types.
3. *For all* D, τ, $c_1 \neq \bot_{\mathrm{e}}$, *and* $c_2 \neq \bot_{\mathrm{e}}$ *such that* $\mathrm{D} \models_{\mathrm{e}} c_1 \otimes c_2 : \mathsf{CTL}$,
   (a) $\langle\!\langle \tau, c_1 \otimes c_2 \rangle\!\rangle^-_{\mathrm{e}} = \langle\!\langle \tau, c_2 \rangle\!\rangle^-_{\mathrm{e}}$,
   (b) $\langle\!\langle \tau, c_1 \otimes c_2 \rangle\!\rangle^+_{\mathrm{e}} = \langle\!\langle \tau, c_1 \rangle\!\rangle^+_{\mathrm{e}}$, *and*
   (c) $\langle\!\langle \tau, c_1 \rangle\!\rangle^-_{\mathrm{e}} = \langle\!\langle \tau, c_2 \rangle\!\rangle^+_{\mathrm{e}}$.
   RATIONALE. Effect sequencing must maintain answer types in order for the continuations of sequenced expressions to compose.
4. *If* $\mathrm{D} \models_{\mathrm{e}} c_1 \preceq c_2$, *then for every type* τ *there exists some type* τ' *such that* $\langle\!\langle \tau', c_1 \rangle\!\rangle^-_{\mathrm{e}} = \langle\!\langle \tau, c_2 \rangle\!\rangle^-_{\mathrm{e}}$ *and* $\langle\!\langle \tau', c_1 \rangle\!\rangle^+_{\mathrm{e}} = \langle\!\langle \tau, c_2 \rangle\!\rangle^+_{\mathrm{e}}$.
   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* $\Delta; \bullet \vdash \mathsf{done}_{\mathrm{e}} : {}^{\mathsf{L}}(\tau \multimap \langle\!\langle \tau \rangle\!\rangle_{\mathrm{e}})$.
RATIONALE. The $\mathsf{done}_{\mathrm{e}}$ value must be well typed for the translation of a whole program to be well typed.

Parameter Property 3 (Effect sequencing)**.**

If $\mathrm{D} \models_{\mathrm{e}} c_1 \otimes c_2 : \mathsf{CTL}$ *then* $\mathrm{D}^* \vdash (c_1 \otimes c_2)^* \preceq {c_1}^*$ *and* $\mathrm{D}^* \vdash (c_1 \otimes c_2)^* \preceq {c_2}^*$.
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. $c_1 \otimes c_2 = \bot_{\mathrm{e}}$ *if and only if* $c_1 = c_2 = \bot_{\mathrm{e}}$.
   RATIONALE. Sequencing impure expressions should not result in a pure expression.
2. *If* $\mathrm{D} \models_{\mathrm{e}} c_1 \otimes c_2 : \mathsf{CTL}$ *and* $c_1 \otimes c_2 \neq \bot_{\mathrm{e}}$, *then there exist some* $c_1' \neq \bot_{\mathrm{e}}$ *and* $c_2' \neq \bot_{\mathrm{e}}$ *such that*
   - $\mathrm{D} \models_{\mathrm{e}} c_1 \preceq c_1'$,
   - $\mathrm{D} \models_{\mathrm{e}} c_2 \preceq c_2'$,
   - $c_1' \otimes c_2' = c_1 \otimes c_2$, *and*
   - $\mathrm{D} \models_{\mathrm{e}} c_1' \otimes c_2' : \mathsf{CTL}$.

   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 5.4 (§5.4) must be extended, by induction on derivations, for any rules added to the kinding judgment* $\mathrm{D} \models_{\mathrm{e}} i : k$.
2. *Lemma 5.5 (§5.4) must be extended, by induction on derivations, for any rules added to the control effect bounding judgment* $\mathrm{D} \models_{\mathrm{e}} c \succeq \xi$.
3. *Lemma 5.6 (§5.4) must be extended, by induction on derivations, for any rules added to the control effect subsumption judgment* $\mathrm{D} \models_{\mathrm{e}} c_1 \preceq c_2$.
4. *Lemma 5.7 (§5.4) must be extended, by induction on derivations, for any rules added to the expression typing judgment* $\mathrm{D}; \mathrm{G} \models_{\mathrm{e}} e : t\ ; c$.

In §6, we give several example control effects and show that they satisfy the above properties.

5.4 Generic Type Safety

Assuming that the above properties hold of the control effect parameter, we can now prove a type safety theorem for $\lambda^{\mathsf{URAL}}(\mathcal{C})$ that leaves the control effect abstract. We sketch the proof here, but the full proof is available online at www.ccs.neu.edu/~tov/pubs/substructural-control.

We begin with a lemma that ensures that control effects translate to well-formed qualifiers:

Lemma 5.4 (Translation of kinding)**.**
For all D*, i, and k, if* $\mathrm{D} \vDash_{\mathrm{e}} i : k$ *then* $\mathrm{D}^* \vdash i^* : k^*$.

We continue with two lemmas concerning 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 5.5 (Translation of effect bounds)**.**
If $\mathrm{D} \vDash_{\mathrm{e}} c \succeq \xi$ *then* $\mathrm{D}^* \vdash \xi \preceq c^*$.

Lemma 5.6 (Translation of effect subsumption)**.**
If $\mathrm{D} \vDash_{\mathrm{e}} c_1 \preceq c_2$ *then* $\mathrm{D}^* \vdash {c_2}^* \preceq {c_1}^*$.

Proofs of Lemmas 5.4, 5.5, and 5.6. By induction on derivations. □

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 5.7 (Translation of term typing)**.**
If $\mathrm{D}; \mathrm{G} \vDash_{\mathrm{e}} e : t \,;\, c$ *then*

$$\mathrm{D}^*; \mathrm{G}^* \vdash [\![e]\!]_{\mathrm{e}} : {}^{\mathsf{L}}({}^{c^*}(t^* \multimap \langle\!\langle t^*, c \rangle\!\rangle_{\mathrm{e}}^{-}) \multimap \langle\!\langle t^*, c \rangle\!\rangle_{\mathrm{e}}^{+}).$$

Proof. By induction on the typing derivation, generalizing the induction hypothesis thus:

> If $\mathrm{D}; \mathrm{G} \vDash_{\mathrm{e}} e : t \,;\, c$, 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 $\mathrm{D}^*; \mathrm{G}^* \vdash [\![e]\!]_{\mathrm{e}} : {}^{\mathsf{L}}({}^{\xi_0}(t^* \multimap \langle\!\langle \tau_0, c \rangle\!\rangle_{\mathrm{e}}^{-}) \multimap \langle\!\langle \tau_0, c \rangle\!\rangle_{\mathrm{e}}^{+})$.

We consider two cases here:

Case $\dfrac{\mathrm{D}; \mathrm{G} \vDash_{\mathrm{e}} e : t \,;\, c' \qquad \mathrm{D} \vDash_{\mathrm{e}} c' \preceq c}{\mathrm{D}; \mathrm{G} \vDash_{\mathrm{e}} e : t \,;\, c}$**.**

By Property 5 (part 3), $\mathrm{D}^* \vdash {c}^* \preceq {c'}^*$, and thus by Property 1 (part 4), there exists some type τ_0' such that $\langle\!\langle \tau_0', c' \rangle\!\rangle_{\mathrm{e}}^{-} = \langle\!\langle \tau_0, c \rangle\!\rangle_{\mathrm{e}}^{-}$ and $\langle\!\langle \tau_0', c' \rangle\!\rangle_{\mathrm{e}}^{+} = \langle\!\langle \tau_0, c \rangle\!\rangle_{\mathrm{e}}^{+}$. 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 $\mathrm{D}; \mathrm{G} \vDash_{\mathrm{e}} e : t \,;\, c'$, using the same ξ_0 but with τ_0' for τ_0, yielding

$$\mathrm{D}^*; \mathrm{G}^* \vdash [\![e]\!]_{\mathrm{e}} : {}^{\mathsf{L}}({}^{\xi_0}(t^* \multimap \langle\!\langle \tau_0', c' \rangle\!\rangle_{\mathrm{e}}^{-}) \multimap \langle\!\langle \tau_0', c' \rangle\!\rangle_{\mathrm{e}}^{+}).$$

Then it suffices to substitute $\langle\!\langle \tau_0, c \rangle\!\rangle_{\mathrm{e}}^{-}$ for $\langle\!\langle \tau_0', c' \rangle\!\rangle_{\mathrm{e}}^{-}$ and $\langle\!\langle \tau_0, c \rangle\!\rangle_{\mathrm{e}}^{+}$ for $\langle\!\langle \tau_0', c' \rangle\!\rangle_{\mathrm{e}}^{+}$, which we know to be equal by Property 1 (part 4).

Case $\dfrac{\begin{array}{c}\mathrm{D} \vDash_{\mathrm{e}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_2 \qquad \mathrm{D} \vDash_{\mathrm{e}} \mathrm{G}_2 \preceq \xi_2 \\ \mathrm{D}; \mathrm{G}_1 \vDash_{\mathrm{e}} e_1 : {}^{\xi_1}(t_1 \xrightarrow{c} t_2) \,;\, c_1 \qquad \mathrm{D} \vDash_{\mathrm{e}} c_1 \succeq \xi_2 \\ \mathrm{D}; \mathrm{G}_2 \vDash_{\mathrm{e}} e_2 : t_1 \,;\, c_2 \qquad \mathrm{D} \vDash_{\mathrm{e}} c_2 \succeq \xi_1 \\ \mathrm{D} \vDash_{\mathrm{e}} c_1 \otimes c_2 \otimes c : \mathsf{CTL}\end{array}}{\mathrm{D}; \mathrm{G} \vDash_{\mathrm{e}} e_1\, e_2 : t_2 \,;\, c_1 \otimes c_2 \otimes c}$**.**

For rule C-T-APP, we want to show that $[\![e_1\, e_2]\!]_{\mathrm{e}}$ has type

$${}^{\mathsf{L}}({}^{\xi_0}(t_2^* \multimap \langle\!\langle \tau_0, c_1 \otimes c_2 \otimes c \rangle\!\rangle_{\mathrm{e}}^{-}) \multimap \langle\!\langle \tau_0, c_1 \otimes c_2 \otimes c \rangle\!\rangle_{\mathrm{e}}^{+}).$$

Consider the translation of $e_1\, e_2$,

$$\lambda y.\, [\![e_1]\!]_{\mathrm{e}}\, (\lambda x_1.\, [\![e_2]\!]_{\mathrm{e}}\, (\lambda x_2.\, x_1 \_ x_2\, (\lambda x.\, y\, x))).$$

The type derivation is too large to show here in detail, 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 $\mathrm{D}^* \vdash \xi_0 \preceq (c_1 \otimes c_2 \otimes c)^*$. By Property 3, each of ${c_1}^*$, ${c_2}^*$, and c^* is greater than $(c_1 \otimes c_2 \otimes c)^*$, 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 $[\![e_1]\!]_{\mathrm{e}}$ is

$$\lambda x_1.\, [\![e_2]\!]_{\mathrm{e}}\, (\lambda x_2.\, x_1 \_ x_2\, (\lambda x.\, y\, x)),$$

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 5 (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 ${}^{\xi_1}((t_1 \xrightarrow{c} t_2)^*)$, so it remains to show that $\mathrm{D}^* \vdash \xi_1 \preceq {c_2}^*$, by Property 5 (part 2) applied to the premise $\mathrm{D} \vDash_{\mathrm{e}} c_2 \succeq \xi_1$.

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 5.2 (Dereliction), the type of the η expansion $\lambda x.\, y\, x$ may be given qualifier c^*. □

Corollary 5.8 (Translation of program typing)**.**
If $\mathrm{D}; \mathrm{G} \vdash_{\mathrm{e}} e : t \,;\, \bot_{\mathrm{e}}$ *where* $\mathrm{D} \vdash_{\mathrm{e}} t \preceq \mathsf{A}$*, then*

$$\mathrm{D}^*; \mathrm{G}^* \vdash [\![e]\!]_{\mathrm{e}}\ \mathsf{done}_{\mathrm{e}} : \langle\!\langle t^* \rangle\!\rangle_{\mathrm{e}}.$$

Proof. By Lemma 5.4, Lemma 5.7, Property 2, and rules QSUB-REFL and T-APP. □

Lemma 5.9 (λ^{URAL} safety)**.**
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 5.10 ($\lambda^{\mathsf{URAL}}(\mathcal{C})$ safety)**.**
If $\bullet; \bullet \vdash_{\mathrm{e}} e : t \,;\, \bot_{\mathrm{e}}$*, and* $\bullet \vdash_{\mathrm{e}} t \preceq \mathsf{A}$ *then* $eval(e) \neq$ WRONG.

Proof. By Corollary 5.8, $\bullet; \bullet \vdash [\![e]\!]_{\mathrm{e}}\ \mathsf{done}_{\mathrm{e}} : \langle\!\langle t^* \rangle\!\rangle_{\mathrm{e}}$. Then by Lemma 5.9, either $[\![e]\!]_{\mathrm{e}}\ \mathsf{done}_{\mathrm{e}}$ reduces to a value v, in which case $eval(e) = v$, or $[\![e]\!]_{\mathrm{e}}\ \mathsf{done}_{\mathrm{e}}$ diverges, in which case $eval(e)$ is undefined. □

6. Example Control Effects

In the previous section, we proved type safety for $\lambda^{\mathsf{URAL}}(\mathcal{C})$, a substructural λ calculus parameterized by abstract control effects. In this section, we give three instances of control effects as described by Definition 4.1 and show that they satisfy the properties on which the generic type safety theorem depends.

6.1 Shift and Reset

We define here a control effect instance for delimited continuations. In this example, we restrict answer types to the unit type ${}^{\mathsf{U}}1$ in order to keep the effects simple. In §6.2, we 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, we extend the syntax:

$$e ::= \cdots \mid \mathsf{shift}\, x\, \mathsf{in}\, e \mid \mathsf{reset}\ e \qquad \textit{(expressions)}$$

We give the dynamics of the new expressions by defining their CCoS translations, which are standard:

$$\begin{aligned} [\![\mathsf{reset}\ e]\!]_{\mathcal{D}} &= \lambda y.\, y\, ([\![e]\!]_{\mathcal{D}}\ (\lambda x.\, x)) \\ [\![\mathsf{shift}\, x\, \mathsf{in}\, e]\!]_{\mathcal{D}} &= \lambda y.\, (\lambda x.\ [\![e]\!]_{\mathcal{D}}\ (\lambda x'.\, x')) \\ &\qquad (\Lambda.\, \lambda x.\, \lambda y'.\, y'\, (y\, x)) \end{aligned}$$

To type shift and reset, we define delimited continuation effects d as the dual lattice of the qualifier lattice ξ with a new point $\bot_{\mathcal{D}}$:

$$\begin{aligned} d ::= {} & \bot_{\mathcal{D}} && \textit{(no effect)} \\ \mid {} & \alpha && \textit{(an effect variable)} \\ \mid {} & \overline{\xi} && \textit{(treats continuation like } \xi\textit{)} \\ \mid {} & d_1 \sqcup d_2 && \textit{(effect join)} \end{aligned}$$

Let $\mathcal{D}$ be the set of delimited continuation effects (d) quotiented by the following equivalences:

$$\overline{\mathsf{L}} \sqcup \overline{\xi} = \overline{\xi} \sqcup \overline{\mathsf{L}} = \overline{\xi};$$
$$d \sqcup \bot_{\mathcal{D}} = \bot_{\mathcal{D}} \sqcup d = d \sqcup d = d.$$

(The quotient simplifies defining other functions and relations on delimited continuation effects.) Then we define delimited continuation effects as 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 13. 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 §6.2.) Then reset e is pure and also has type ${}^{\mathsf{U}}1$. To type $\mathsf{shift}\, x\, \mathsf{in}\, e$, we give x type ${}^{\xi}(t \xrightarrow{\bot_{\mathcal{D}}}\!\!\circ\ {}^{\mathsf{U}}1)$ 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 safety. To prove type safety for $\lambda^{\mathsf{URAL}}(\mathcal{C})$ extended with delimited continuation effects, we need to give the translation parameter as described by Definition 5.1. We define the translation parameter as follows:

$$\langle\!\langle \tau, d \rangle\!\rangle^{-}_{\mathcal{D}} = \langle\!\langle \tau, d \rangle\!\rangle^{+}_{\mathcal{D}} = {}^{\mathsf{U}}1$$

$$\mathsf{done}_{\mathcal{D}} = \lambda x.\, \langle\rangle$$

$$d^* = \begin{cases} \mathsf{L} & \text{if } d = \bot_{\mathcal{D}} \\ \alpha & \text{if } d = \alpha \\ \xi & \text{if } d = \overline{\xi} \\ \mathsf{U} & \text{otherwise} \end{cases}$$

Then, we must show that this definition satisfies the properties of §5.3:

Theorem 6.1 (Delimited continuation properties)**.**
Delimited continuation effects $(\mathcal{D}, \bot_{\mathcal{D}}, \sqcup)$ *satisfy Properties 1–5.*

Proof.

Property 1 (Answer types). We must show several equalities on answer types, such as $\langle\!\langle \tau, d_1 \rangle\!\rangle^{-}_{\mathcal{D}} = \langle\!\langle \tau, d_2 \rangle\!\rangle^{+}_{\mathcal{D}}$,

$\boxed{\mathrm{D} \vdash_{\mathcal{D}} i : k}$ *(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 13. Statics for delimited continuation effects

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 5.6.

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 5.7. There are two cases, for shift and reset, each of which requires a large type derivation. □

6.2 Shift and Reset with Answer-Type Modification

The type-and-effect system for shift and reset described in §6.1 requires that all *answer types*—the type of all reset expressions—be ${}^{\mathsf{U}}1$. 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 §6.1, but

$\boxed{\mathrm{D} \vdash_{\mathcal{A}} i : k}$ *(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)*

$$\textsc{ASub-Bot}\ \frac{\mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi}(t \rightarrowtail t) : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{A}} \bot_{\mathcal{A}} \preceq {}^{\Xi}(t \rightarrowtail t)}$$

$$\textsc{ASub-L}\ \frac{\mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi}(t_1 \rightarrowtail t_2) : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{A}} {}^{\mathsf{L}}(t_1 \rightarrowtail t_2) \preceq {}^{\Xi}(t_1 \rightarrowtail t_2)}$$

$$\textsc{ASub-Top}\ \frac{\mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi}(t_1 \rightarrowtail t_2) : \mathsf{CTL}}{\mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi}(t_1 \rightarrowtail t_2) \preceq {}^{\mathsf{U}}(t_1 \rightarrowtail t_2)}$$

$$\textsc{ASub-Join}\ \frac{\mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi_1}(t_1 \rightarrowtail t_2) \preceq {}^{\Xi'_1}(t_1 \rightarrowtail t_2) \qquad \mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi_2}(t_1 \rightarrowtail t_2) \preceq {}^{\Xi'_2}(t_1 \rightarrowtail t_2)}{\mathrm{D} \vdash_{\mathcal{A}} {}^{\Xi_1,\Xi_2}(t_1 \rightarrowtail t_2) \preceq {}^{\Xi'_1,\Xi'_2}(t_1 \rightarrowtail t_2)}$$

$\boxed{\mathrm{D}; \mathrm{G} \vdash_{\mathcal{A}} e : t \,;\, a}$ *(answer-type effect expression typing)*

$$\textsc{A-T-Reset}\ \frac{\mathrm{D}; \mathrm{G} \vdash_{\mathcal{A}} e : t_0 \,;\, {}^{\Xi}(t_0 \rightarrowtail t)}{\mathrm{D}; \mathrm{G} \vdash_{\mathcal{A}} \mathsf{reset}\ e : t \,;\, \bot_{\mathcal{A}}}$$

$$\textsc{A-T-Shift}\ \frac{\mathrm{D}; \mathrm{G}, x{:}^{\xi}(t_1 \xrightarrow{\bot_{\mathcal{A}}\circ} t_2) \vdash_{\mathcal{A}} e : t_0 \,;\, {}^{\Xi}(t_0 \rightarrowtail t)}{\mathrm{D}; \mathrm{G} \vdash_{\mathcal{A}} \mathsf{shift}\, x\, \mathsf{in}\, e : t_1 \,;\, {}^{\Xi,\xi}(t_2 \rightarrowtail t)}$$

Figure 14. Statics for answer-type effects

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:

$$\begin{array}{rll} a & ::= \bot_{\mathcal{A}} & \textit{(pure)} \\ & \mid\ {}^{\Xi}(t_1 \rightarrowtail t_2) & \textit{(a control effect)} \\ \text{where } \Xi & ::= \xi_1, \ldots, \xi_j & \end{array}$$

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:

$$\begin{gathered} \bot_{\mathcal{A}} \circ a = a \\ a \circ \bot_{\mathcal{A}} = a \\ {}^{\Xi}(t' \rightarrowtail t_2) \circ {}^{\Xi'}(t_1 \rightarrowtail t') = {}^{\Xi,\Xi'}(t_1 \rightarrowtail t_2). \end{gathered}$$

Any other cases are undefined. 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 14. 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 §6.1. 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 §6.1, which they follow for the qualifier portion, and the rules from Danvy and Filinski (1989), which they follow for maintaining answer types.

Type safety. To prove type safety for $\lambda^{\mathsf{URAL}}(\mathcal{C})$ extended with answer-type modification, we define the translation parameter as follows:

$$\begin{aligned}
\langle\!\langle \tau, \bot_{\mathcal{A}} \rangle\!\rangle^{-}_{\mathcal{A}} &= \tau \\
\langle\!\langle \tau, {}^{\Xi}(t_1 \rightarrowtail t_2) \rangle\!\rangle^{-}_{\mathcal{A}} &= {t_1}^* \\
\langle\!\langle \tau, \bot_{\mathcal{A}} \rangle\!\rangle^{+}_{\mathcal{A}} &= \tau \\
\langle\!\langle \tau, {}^{\Xi}(t_1 \rightarrowtail t_2) \rangle\!\rangle^{+}_{\mathcal{A}} &= {t_2}^* \\
\mathsf{done}_{\mathcal{A}} &= \lambda x.x \\
a^* &= \begin{cases} \mathsf{L} & \text{if } a = \bot_{\mathcal{A}} \\ \xi & \text{if } a = {}^{\xi}(t_1 \rightarrowtail t_2) \\ \mathsf{U} & \text{otherwise} \end{cases}
\end{aligned}$$

Theorem 6.2 (Answer-type effect properties)**.**
Answer-type modification effects $(\mathcal{A}, \bot_{\mathcal{A}}, \circ)$ *satisfy Properties 1–5.*

6.3 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:

$$\begin{array}{lll} \psi \in \mathit{Exn} & & \textit{(exceptions)} \\ e ::= \cdots \mid \mathsf{raise}\ \psi \mid e_1\ \mathsf{handle}\ \psi \to e_2 & & \textit{(expressions)} \end{array}$$

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:

$$\begin{array}{lll} \Psi ::= \varnothing & & \textit{(pure)} \\ \mid\ \alpha & & \textit{(an effect variable)} \\ \mid\ \{\psi\} & & \textit{(a single exception)} \\ \mid\ \Psi_1 \cup \Psi_2 & & \textit{(exception set union)} \end{array}$$

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:

$\boxed{\mathrm{D} \models_{\overline{\mathrm{x}}} \Psi_1 \preceq \Psi_2}$ *(exception effect subsumption)*

$$\textsc{XSub-Subset}\ \frac{\Psi_1 \subseteq \Psi_2 \qquad \mathrm{D} \models_{\overline{\mathrm{x}}} \Psi_1 : \mathsf{CTL} \qquad \mathrm{D} \models_{\overline{\mathrm{x}}} \Psi_2 : \mathsf{CTL}}{\mathrm{D} \models_{\overline{\mathrm{x}}} \Psi_1 \preceq \Psi_2}$$

The other new type rules for exception effects appear in Figure 15. 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

$\boxed{\mathrm{D} \models_{\overline{\mathrm{x}}} i : k}$ *(kinding exception effects)*

$$\textsc{X-K-Sing}\ \frac{}{\mathrm{D} \models_{\overline{\mathrm{x}}} \{\psi\} : \mathsf{CTL}} \qquad \textsc{X-K-Union}\ \frac{\mathrm{D} \models_{\overline{\mathrm{x}}} \Psi_1 : \mathsf{CTL} \qquad \mathrm{D} \models_{\overline{\mathrm{x}}} \Psi_2 : \mathsf{CTL}}{\mathrm{D} \models_{\overline{\mathrm{x}}} \Psi_1 \cup \Psi_2 : \mathsf{CTL}}$$

$\boxed{\mathrm{D} \models_{\overline{\mathrm{x}}} \Psi \succeq \xi}$ *(qualifier bound for exception effects)*

$$\textsc{X-B-Raise}\ \frac{\mathrm{D} \models_{\overline{\mathrm{x}}} \Psi : \mathsf{CTL}}{\mathrm{D} \models_{\overline{\mathrm{x}}} \Psi \succeq \mathsf{A}}$$

$\boxed{\mathrm{D}; \mathrm{G} \models_{\overline{\mathrm{x}}} e : t \,;\, \Psi}$ *(exception effect expression typing)*

$$\textsc{X-T-Raise}\ \frac{\mathrm{D} \models_{\overline{\mathrm{x}}} t : \star}{\mathrm{D}; \bullet \models_{\overline{\mathrm{x}}} \mathsf{raise}\ \psi : t \,;\, \{\psi\}}$$

$$\textsc{X-T-Handle}\ \frac{\begin{array}{c}\mathrm{D} \models_{\overline{\mathrm{x}}} \mathrm{G} \rightsquigarrow \mathrm{G}_1 \boxplus \mathrm{G}_2 \\ \mathrm{D}; \mathrm{G}_1 \models_{\overline{\mathrm{x}}} e_1 : t \,;\, \{\psi\} \cup \Psi \\ \mathrm{D}; \mathrm{G}_2 \models_{\overline{\mathrm{x}}} e_2 : t \,;\, \Psi \qquad \mathrm{D} \models_{\overline{\mathrm{x}}} \mathrm{G}_2 \preceq \mathsf{A}\end{array}}{\mathrm{D}; \mathrm{G} \models_{\overline{\mathrm{x}}} e_1\ \mathsf{handle}\ \psi \to e_2 : t \,;\, \Psi}$$

Figure 15. Statics for exception effects

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

$$\frac{}{[v_1, v_2]_\psi\ \psi^* \longmapsto v_1\ \psi^*} \qquad \frac{\psi \neq \psi'}{[v_1, v_2]_\psi\ \psi'^* \longmapsto v_2\ \psi'^*}.$$

$$\frac{\begin{array}{ll}\Delta; \Gamma \vdash v_1 : {}^{\xi_1}({}^{\mathsf{U}}\mathsf{exn} \multimap \tau) & \Delta \vdash \xi_1 \preceq \xi \\ \Delta; \Gamma \vdash v_2 : {}^{\xi_2}({}^{\mathsf{U}}\mathsf{exn} \multimap \tau) & \Delta \vdash \xi_2 \preceq \xi\end{array}}{\Delta; \Gamma \vdash [v_1, v_2]_\psi : {}^{\xi}({}^{\mathsf{U}}\mathsf{exn} \multimap \tau)}$$

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:

$$[\![\mathsf{raise}\ \psi]\!]^{\Psi}_{x} = \lambda\_.\ \mathsf{inl}\ \psi^{*}$$
$$[\![e_1\ \mathsf{handle}\ \psi \to e_2]\!]^{\Psi}_{x} = \lambda y.\,[v, y]\,([\![e_1]\!]^{\{\psi\}\cup\Psi}_{x}\ (\lambda x.\ \mathsf{inr}\ x))$$

$$\text{where } v = \begin{cases} \lambda\_.\,[\![e_2]\!]^{\varnothing}_{x}\ y & \text{if } \Psi = \varnothing; \\ [\lambda\_.\,[\![e_2]\!]^{\Psi}_{x}\ y, \lambda x.\ \mathsf{inl}\ x]_{\psi} & \text{if } \Psi \neq \varnothing. \end{cases}$$

Example. The first Scala example from §1 may be recast in λ^{URAL} (with references, pairs, and integer division) as follows:

$$\lambda z_1\, z_2.\,\mathsf{pair}\,(\mathsf{ref}\,(z_1 \,/\, z_2))\,(\mathsf{ref}\,(z_2 \,/\, z_1))$$

Let us assume the following (monomorphic, for brevity) types for the operations:

$$\cdot\,/\,\cdot : {}^{\mathsf{U}}(\mathsf{int} \xrightarrow{\varnothing}_{\circ} {}^{\mathsf{U}}(\mathsf{int} \xrightarrow{\{\mathsf{DivBy0}\}}_{\circ} {}^{\mathsf{U}}\mathsf{int}))$$
$$\mathsf{ref} : {}^{\mathsf{U}}(\mathsf{int} \xrightarrow{\varnothing}_{\circ} {}^{\mathsf{L}}\mathsf{intref})$$
$$\mathsf{pair} : {}^{\mathsf{U}}({}^{\mathsf{L}}\mathsf{intref} \xrightarrow{\varnothing}_{\circ} {}^{\mathsf{L}}({}^{\mathsf{L}}\mathsf{intref} \xrightarrow{\varnothing}_{\circ} {}^{\mathsf{L}}({}^{\mathsf{L}}\mathsf{intref} \otimes {}^{\mathsf{L}}\mathsf{intref})))$$

To type the application of term $\mathsf{pair}\,(\mathsf{ref}\,(z_1 \,/\, z_2))$ to term $\mathsf{ref}\,(z_2 \,/\, z_1)$, according to premise (6) of rule C-T-App, the effect of the operator must be bounded by the qualifier of the type of the operand. The effect of the operator, $\mathsf{pair}\,(\mathsf{ref}\,(z_1 \,/\, z_2))$, is $\{\mathsf{DivBy0}\}$, based on the type of $/$; the type of the operand, $\mathsf{ref}\,(z_2 \,/\, z_1)$, is ${}^{\mathsf{L}}\mathsf{intref}$. But $\bullet \vdash_{\overline{\chi}} \{\mathsf{DivBy0}\} \succeq \mathsf{L}$ is not derivable—a term that can raise an exception does not necessarily treat it context linearly—so the original code has a type error in $\lambda^{\mathsf{URAL}}(\mathcal{C})$.

We can repair the example, as we did in §1, by explicitly ordering the effects so that both divisions happen before any references are allocated:

$$\lambda z_1\, z_2.\,(\lambda x_1\, x_2.\,\mathsf{pair}\,(\mathsf{ref}\,x_1)\,(\mathsf{ref}\,x_2))\,(z_1 \,/\, z_2)\,(z_2 \,/\, z_1)$$

Term $\lambda x_1\, x_2.\,\mathsf{pair}\,(\mathsf{ref}\,x_1)\,(\mathsf{ref}\,x_2)$ has an unlimited type:

$${}^{\mathsf{U}}(\mathsf{int} \xrightarrow{\varnothing}_{\circ} {}^{\mathsf{U}}(\mathsf{int} \xrightarrow{\varnothing}_{\circ} {}^{\mathsf{L}}({}^{\mathsf{L}}\mathsf{intref} \otimes {}^{\mathsf{L}}\mathsf{intref})))$$

Thus, it does not matter that its argument, $z_1 \,/\, z_2$, has non-trivial effect. Similarly, because the codomain of that type is unlimited, it is permissible that the second argument, $z_2 \,/\, z_1$, has non-trivial effect as well. Thus, the repaired example is typeable in $\lambda^{\mathsf{URAL}}(\mathcal{C})$.

Type safety. To prove type safety for $\lambda^{\mathsf{URAL}}(\mathcal{C})$ extended with exceptions, we define the translation parameter as follows:

$$\langle\!\langle \tau, \Psi \rangle\!\rangle^{-}_{x} = \langle\!\langle \tau, \Psi \rangle\!\rangle^{+}_{x} = {}^{\mathsf{L}}({}^{\mathsf{U}}\mathsf{exn} \oplus \tau)$$
$$\mathsf{done}_{x} = \lambda x.\ \mathsf{inr}\ x$$
$$\Psi^{*} = \begin{cases} \mathsf{L} & \text{if } \Psi = \varnothing \\ \mathsf{A} & \text{if } \Psi \neq \varnothing \end{cases}$$

Theorem 6.3 (Exception effect properties)**.**

Exception effects $(\mathcal{X}, \varnothing, \cup)$ *satisfy Properties 1–5.*

The proofs of theorems in this section appear in the extended version of this paper, available at www.ccs.neu.edu/~tov/pubs/substructural-control.

7. Conclusion

We began this study with the desire to add linear types to Alms, a general-purpose programming language with affine types and exceptions. The treatment of exceptions in §6.3 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. We believe that with appropriate defaults most function arrows will not require annotation, but more research is required in that direction.

Another potential direction for future research is to consider how other control effects fit into our general framework. We 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 have not considered them. Others, such as Felleisen's *prompt* and *control* (1988) are probably tractable with a more expressive version of our 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.

For the cases we consider, however, $\lambda^{\mathsf{URAL}}(\mathcal{C})$ provides a simple and generic framework for integrating substructural types and control effects. We have shown that our type system for $\lambda^{\mathsf{URAL}}(\mathcal{C})$ is sound provided that the particular instantiation of control effects meets several criteria, and we have exhibited three instances of control effects that meet these criteria. We contend that this provides a solid grounding for the extension of realistic substructural programming languages with control effects.

Acknowledgments

We wish to thank Vincent St-Amour, Sam Tobin-Hochstadt, Aaron Turon, and the anonymous referees for their helpful comments, discussion, and corrections. This research was supported in part by AFOSR grant FA9550-09-1-0110.

References

A. Ahmed, M. Fluet, and G. 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.

J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In *Proc. Onward!*, pages 1015–1022, Orlando, FL, USA, October 2009.

K. Asai and Y. Kameyama. Polymorphic delimited continuations. In *Programming Languages and Systems*, volume 4807 of *Lecture Notes in Computer Science*, pages 239–254. Springer, 2007.

A. Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-960347, Laboratory for Foundations of Computer Science, University of Edinburgh, September 1996.

G. M. Bierman. *On Intuitionistic Linear Logic*. PhD thesis, University of Cambridge, August 1993.

W. Clinger, ed. The revised revised report on Scheme or an UnCommon Lisp. AI Memo No. 848, MIT AI Lab, Cambridge, MA, USA, August 1985.

O. Danvy and A. Filinski. A functional abstraction of typed contexts. Technical Report DIKU Rapport 89/12, Computer Science Department, University of Copenhagen, Denmark, 1989.

M. Felleisen. The theory and practice of first-class prompts. In J. Ferrante and P. Mager, editors, *Proc. 15th Annual ACM Symposium on Principles of Programming Languages (POPL'88)*, pages 180–190, San Diego, CA, USA, January 1988.

J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

J. B. Goodenough. Structured exception handling. In *Proc. 2th Annual ACM Symposium on Principles of Programming Languages (POPL'75)*, pages 204–224, Palo Alto, CA, USA, January 1975.

J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.

D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. 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.

O. Kiselyov and C. 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.

P. J. Landin. A generalization of jumps and labels. Technical report, UNIVAC Systems Programming Research, 1965.

J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In J. Ferrante and P. Mager, editors, *Proc. 15th Annual ACM Symposium on Principles of Programming Languages (POPL'88)*, pages 47–57, San Diego, CA, USA, January 1988.

K. Mazurak and S. 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, USA, September 2010.

G. Morrisett, A. Ahmed, and M. 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.

M. Odersky and M. Zenger. Scalable component abstractions. In *Proc 20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 41–57, San Diego, CA, USA, October 2005.

J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proc. ACM Annual Conference*, volume 2, pages 717–470, Boston, MA, USA, August 1972.

N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in Fine. In A. D. Gordon, editor, *Proc. 19th European Symposium on Programming (ESOP'10)*, volume 6012 of *Lecture Notes in Computer Science*, pages 529–549, Paphos, Cyprus, March 2010.

H. 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, USA, January 2003.

J. A. Tov and R. Pucella. Practical affine types. In *Proc. 38th Annual ACM Symposium on Principles of Programming Languages (POPL'11)*, pages 447–458, Austin, TX, USA, January 2011.

P. 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.

D. Walker. Substructural type systems. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pages 3–44. MIT Press, Cambridge, 2005.