PC: Amal Ahmed
Robby Findler (chair)
Fritz Henglein
Gavin Bierman
Gilad Bracha
Jeff Foster
Peter Thiemann
Sam Tobin-Hochstadt

SC: Matthias Felleisen
Cormac Flanagan
Nate Nystrom
Jan Vitek
Philip Wadler
Tobias Wrigstad

Organizers: Tobias Wrigstad and Jan Vitek

Schedule

Invited Talk: Scripting in a Concurrent World

John Field

IBM Research

As scripting languages are used to build increasingly complex systems, they must eventually confront concurrency. Concurrency typically arises from two distinct needs: handling "naturally" concurrent external (human- or software-generated) events, and enhancing application performance. Concurrent applications are difficult to program in general; these difficulties are multiplied in a distributed setting, where partial failures are common and where resources cannot be centrally managed. The distributed systems community has made enormous progress over the past few decades designing specialized systems that scale to handle millions of users and petabytes of data. However, combining individual systems into composite applications that are scalable—not to mention reliable, secure, and easy to develop maintain—remains an enormous challenge. This is where programming languages should be able to help: good languages are designed to facilitate composing large applications from smaller components and for reasoning about the behavior of applications modularly. In this talk, I will discuss some of the challenges inherent in building distributed and concurrent applications, and discuss how scripting languages could evolve to address these challenges. I will illustrate some of the ideas using examples from the Thorn programming language, which is being developed jointly by IBM and Purdue University.

The work on Thorn is joint with B. Bloom, B. Burg, J. Dolby, N. Nystrom, J. Östlund, G. Richards, R. Strnisa, E. Torlak, J. Vitek, and T. Wrigstad.

Bio: John Field is a Research Staff Member at IBM's T.J. Watson Research Center and manager of the Advanced Programming Tools Group. He received a Ph.D. from Cornell University in 1991, where he was a member of the team that built the Synthesizer Generator, a multi-lingual IDE that was subsequently developed into a suite of programming productivity tools by Grammatech, Inc. At IBM, his research has centered on the design of tools and programming models for large software systems, with the aim of increasing programmer productivity and software quality. Most recently, his work has focused on tools and programming models for distributed and concurrent applications. Tools and techniques developed by the Advanced Programming Tools Group have been incorporated into a number of IBM application development products, including the VisualAge Cobol suite, the VisualAge Y2K Analysis and Remediation Toolkit (which received an Outstanding Technical Achievement Award from IBM), IBM DB2's Database Migration Toolkit, and IBM Rational's AppScan DE web security tool.

Language with a Pluggable Type System and Optional Runtime Monitoring of Type Errors

Jukka Lehtosalo and David J. Greaves

University of Cambridge Computer Laboratory
firstname.lastname@cl.cam.ac.uk

Abstract. Adding a static type system to a dynamically-typed language can be an invasive change that requires coordinated modification of existing programs, virtual machines and development tools. Optional pluggable type systems do not affect runtime semantics of programs, and thus they can be added to a language without affecting existing code and tools. However, in programs mixing dynamic and static types, pluggable type systems do not allow reporting runtime type errors precisely. We present *optional runtime monitoring of type errors* for tracking these errors without affecting execution semantics. Our Python-like target language Alore has a nominal optional type system with *bindable interfaces* that can be bound to existing classes by clients to help the safe evolution of programs and scripts to static typing.

1 Introduction

Dynamic typing enables high productivity for scripting, but it does not scale well to large-scale software development. Adding an optional static type system that allows gradually evolving a dynamically-typed program to a statically-typed one has been proposed as a solution to this problem [15–18].

Several factors make adding static type checking to a mature dynamically-typed language such as Python challenging. Adding the type system is an invasive change that affects the language in fundamental ways. All the tooling from virtual machines, compilers, debuggers to integrated debugging environments needs to be updated to be aware of the static type system.

This objection can be dealt with, in part, by using an *optional pluggable type system* that does not affect the runtime semantics of the language: existing tooling that is not aware of the type system can still be used, although with potentially limited effectiveness. Thus there is a migration path to static typing that does not require drastic changes to the infrastructure. This is analogous to GJ [5] and Java generics, which augmented the Java type system while retaining runtime semantics that are compatible with previous Java and JVM versions.

Gradual types [15, 16, 19] and contracts [7] enable type errors and blame to be tracked in the boundary between static and dynamic typing, but since an error causes the program to be terminated, this affects program semantics. A common objection against previous pluggable type systems [3, 4] was that they do not define how to detect these kinds of errors. We propose using *optional runtime monitoring of type errors* to find

runtime type violations automatically, without having to modify the language semantics. A runtime debugger detects and reports type errors, but it allows continuing the program execution after runtime type errors.

Python is a complex language and has extensive dynamic features such as *eval* that make it difficult to retrofit it with static typing. As a result, either the type system will fail to properly support some language functionality or the type system has to be very sophisticated to deal with the language complexity. A complicated type system increases the effort of updating and maintaining all the tooling, and acts as a barrier of entry for existing programmers.

As a step towards solving the above objection, we have decided to use as our vehicle for exploration a language that is semantically easier to deal with than Python, but which shares many interesting properties with Python. Although in many ways similar to Python, our target language Alore also diverges from it in several ways to enable designing a simple but useful type system. An implemention of the dynamically-typed subset of Alore with extensive documentation is available for download[1].

Finally, existing libraries and frameworks for a mature language are often difficult to retrofit with static typing, due to heavy reliance on dynamic language features. We have also implemented a core standard library for our language, inspired by the Python standard library, to enable experimenting with realistic programs. In contrast to Python, a relatively simple type system is sufficient for adding static types to the library.

In Section 2, we formalise the core language FJ$^?$ that is semantically equivalent to a subset of Alore, our target language. It is very similar to Featherweight Java (FJ) [10], but it supports mixing dynamically-typed and statically-typed code. The formalisation includes a very simple pluggable type system.

Section 3 describes optional runtime monitoring for reporting runtime type errors in FJ$^?$ programs and discusses its properties. Section 4 indicates how the type system and runtime monitoring system for the core calculus can be extended to support Alore.

Alore includes many features of Python while supporting optional static typing. In Section 5 we compare our language with Python, focusing on features that affect the type system. Finally, section 6 discusses related work, and Section 7 presents conclusions and directions for future work.

2 Core Language

A program in our core language FJ$^?$ consists of a sequence of mutually recursive class definitions `L` and a single expression `e`.[2] It supports classes with fields (`f` or `g`), a constructor (`K`) and methods (`M`), recursion through the `this` object, casts, inheritance and method overriding. Figure 1 defines the syntax of FJ$^?$. The use of Java-like syntax throughout this paper highlights the similarity to FJ. All FJ$^?$ programs can be trivially translated to the Alore syntax.

`C` and `D` range over class names, and a type (`T`, `S` or `U`) is either a class name or the *dynamic type* '?'. The set of variables (`x`) also includes `this`. The class `Object` is the top

[1] `http://www.alorelang.org/`

[2] The name FJ$^?$ was also used by Ina and Igarashi [11] for FJ extended with gradual typing. Our approach is similar to theirs, but it uses semantics-preserving tracking of runtime type errors.

```
L ::= class C extends C { T̄ f̄; K M̄; }
K ::= C(T̄ f̄) { super(f̄); this.f̄=f̄; }
M ::= T m(T̄ x̄) { return e; }
e ::= x | e.f | e.m(ē) | new C(ē) | (T)e
```

Fig. 1. Syntax of FJ$^?$.

of the inheritance hierarchy. Several notational conventions simplify the presentation. $\overline{\mathtt{x}}$ stands for the sequence $\mathtt{x}_1, \ldots, \mathtt{x}_n$. The comma operator is also used for sequence concatenation. $\overline{\mathtt{C}}\ \overline{\mathtt{f}}$ is shorthand for $\mathtt{C}_1\ \mathtt{f}_1; \ldots; \mathtt{C}_n\ \mathtt{f}_n$. Similarly, we use $\mathtt{this.}\overline{\mathtt{f}}\mathtt{=}\overline{\mathtt{f}}$ to mean $\mathtt{this.f}_1\mathtt{=f}_1; \ldots; \mathtt{this.f}_n\mathtt{=f}_n$. Note that we omit several basic consistency assumptions in the rules for simplicity (fields cannot be overridden, inheritance hierarchies must not form cycles, all class and variables names must be bound, method and field names must be distinct, etc.). Any type declarations can be omitted, and these types are implicitly defined as '?'. The formalisation, however, assumes that all types are explicitly given.

Subtyping is based on inheritance. Like Siek and Taha [15] in their gradual type system, we use a *consistency relation* to determine the compatibility of types in addition to ordinary subtyping. The consistency relation $\sim$ is used for defining type compatibility. The ? type is consistent with every other type. The $\lesssim$ relation models consistency *or* subtyping. Rules for subtyping ($<:$), consistency ($\sim$) and consistent-or-subtype ($\lesssim$) are given below:

$$\mathtt{T} <: \mathtt{T} \qquad \frac{\mathtt{S} <: \mathtt{T} \qquad \mathtt{T} <: \mathtt{U}}{\mathtt{S} <: \mathtt{U}} \qquad \frac{\mathtt{class\ C\ extends\ D\ \{\ \ldots\ \}}}{\mathtt{C} <: \mathtt{D}}$$

$$\mathtt{T} \sim \mathtt{T} \qquad \mathtt{T} \sim ? \qquad ? \sim \mathtt{T} \qquad \frac{\mathtt{S} <: \mathtt{T}}{\mathtt{S} \lesssim \mathtt{T}} \qquad \frac{\mathtt{S} \sim \mathtt{T}}{\mathtt{S} \lesssim \mathtt{T}}$$

The auxiliary function $\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ gives the field names and their types of class C. $\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{T}} \to \mathtt{T}$ gives the signature of method m of class C. $\mathit{mbody}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{x}}.\,\mathtt{e}$ gives the body e and arguments x of a method. We omit the definitions of these functions; they are identical to FJ.

The type system is nominal, class-based with single inheritance. It is similar to Java in the hope of making it easy to adopt by programmers familiar with Java. Unlike Python, the language has cast expressions. The type ? enables statically-typed and dynamically-typed code to be mixed.

Figure 2 defines the evaluation rules for the language. These are equivalent to Featherweight Java except for the rule DYCAST. It causes *dynamic casts* to be ignored during evaluation – they only affect type checking. It is notable that the evaluation rules never refer to the statically declared types of fields or methods.

Figure 3 presents selected typing rules of the core language that differ from FJ. All operations are permitted on values of type ?. As result, every valid FJ$^?$ program typechecks even when it is type-erased by replacing all declared types with the *?* type. Fully typed FJ$^?$ programs with no *?* types inherit the type safety property of FJ [10].

Like FJ, the T-METHOD rule requires the signature of a method that overrides a superclass method to be *equal* to the superclass method signature. We could relax this

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\mathtt{new\ C}(\overline{\mathtt{e}}).\mathtt{f}_i \longrightarrow \mathtt{e}_i}\ \text{(R-FIELD)} \qquad \frac{\mathit{mbody}(\mathtt{m},\mathtt{C}) = \overline{\mathtt{x}}.\,\mathtt{e}_0}{\mathtt{new\ C}(\overline{\mathtt{e}}).\mathtt{m}(\overline{\mathtt{d}}) \longrightarrow [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{C}(\overline{\mathtt{e}})/\mathtt{this}]\mathtt{e}_0}\ \text{(R-INVK)}$$

$$\frac{\mathtt{C} <: \mathtt{T}}{(\mathtt{T})\mathtt{new\ C}(\overline{\mathtt{e}}) \longrightarrow\ \mathtt{new\ C}(\overline{\mathtt{e}})}\ \text{(R-CAST)} \qquad (?)\mathtt{new\ C}(\overline{\mathtt{e}}) \longrightarrow\ \mathtt{new\ C}(\overline{\mathtt{e}})\ \text{(R-DYCAST)}$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0'}{\mathtt{e}_0.\mathtt{f} \longrightarrow \mathtt{e}_0'.\mathtt{f}}\ \text{(RC-FIELD)} \qquad \frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0'}{\mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \longrightarrow \mathtt{e}_0'.\mathtt{m}(\overline{\mathtt{e}})}\ \text{(RC-INVK-RECV)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i'}{\mathtt{e}_0.\mathtt{m}(\ldots,\mathtt{e}_i,\ \ldots) \longrightarrow \mathtt{e}_0.\mathtt{m}(\ldots,\mathtt{e}_i',\ \ldots)}\ \text{(RC-INVK-ARG)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i'}{\mathtt{new\ C}(\ldots,\mathtt{e}_i,\ \ldots) \longrightarrow\ \mathtt{new\ C}(\ldots,\mathtt{e}_i',\ \ldots)}\ \text{(RC-NEW-ARG)} \qquad \frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0'}{(\mathtt{T})\mathtt{e}_0 \longrightarrow (\mathtt{T})\mathtt{e}_0'}\ \text{(RC-CAST)}$$

Fig. 2. Reduction rules for FJ$^?$.

Expression typing:

$$\frac{\Gamma \vdash \mathtt{e} : ?}{\Gamma \vdash \mathtt{e.f} : ?}\ \text{(T-DYFIELD)} \qquad \frac{\Gamma \vdash \mathtt{e} : ? \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}}{\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : ?}\ \text{(T-DYINVK)}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{C} \qquad \mathit{mtype}(\mathtt{m},\mathtt{C}) = \overline{\mathtt{S}} \to \mathtt{T} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}} \qquad \overline{\mathtt{T}} \lesssim \overline{\mathtt{S}}}{\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{T}}\ \text{(T-INVK)}$$

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{S}}\ \overline{\mathtt{f}} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}} \qquad \overline{\mathtt{T}} \lesssim \overline{\mathtt{S}}}{\Gamma \vdash\ \mathtt{new\ C}(\overline{\mathtt{e}}) : \mathtt{C}}\ \text{(T-CREAT)} \qquad \frac{\Gamma \vdash \mathtt{e} : \mathtt{T}}{\Gamma \vdash (?)\mathtt{e} : ?}\ \text{(T-DYCAST)}$$

$$\frac{\Gamma \vdash \mathtt{e} : ?}{\Gamma \vdash (\mathtt{C})\mathtt{e} : \mathtt{C}}\ \text{(T-DYCAST2)}$$

Method typing:

$$\frac{\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e}_0 : \mathtt{S}_0 \qquad \mathtt{S}_0 \lesssim \mathtt{T}_0 \qquad \mathtt{class\ C\ extends\ D\ \{\ \ldots\ \}} \quad \text{if } \mathit{mtype}(\mathtt{m},\mathtt{D}) = \overline{\mathtt{U}} \to \mathtt{U}_0, \text{then } \overline{\mathtt{T}} = \overline{\mathtt{U}} \text{ and } \mathtt{T}_0 = \mathtt{U}_0}{\mathtt{T}_0\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \{\ \mathtt{return\ e}_0;\ \}\ \mathtt{OK\ IN\ C}}\ \text{(T-METHOD)}$$

Fig. 3. Selected typing judgments for FJ$^?$.

requirement to only require consistency, but we have declined to do so to let us use a very straightforward implementation of runtime monitoring of type errors in the next section.

The full Alore language supports assignment, `if`, `while` and `for` statements, exceptions and other features, but FJ$^?$ highlights important aspects of the Alore type system.

3 Optional Runtime Monitoring

When a dynamically-typed value is bound to a statically-typed variable, which we call *crossing the dynamic-static boundary*, we would like to verify that the dynamic type

matches the static type at runtime. However, the semantics of FJ$^?$ do not generally allow this, and type errors may be silently ignored or discovered only later in the program execution, making debugging difficult.

We propose adding an optional runtime-type-error monitoring system that tracks these kinds of type violations by adding *type guards* (described later in this section), wrapper functions or wrapper objects when necessary to track type errors that may happen when or after crossing the dynamic-static boundary (similar in spirit to gradual typing [15] and contracts [7]). Unlike previous work, the monitoring is independent of the runtime semantics of the programming language.

In particular, the monitoring system logs detected type errors (to a file or to a terminal, for example), but the program execution is unaffected by them (unless the type error was also caught by the runtime semantics). It is important that the program cannot respond to the logged type errors, at least without examining the file system or the environment in a non-portable fashion. As an important implementation detail, the log size must be capped – otherwise the log file of a long-running program may fill the file system, making the monitoring system not quite semantics-preserving!

A pluggable type system does not enforce any particular method for tracking runtime type errors that are not caught by the runtime semantics. Different virtual machines can support different mechanisms. Even the context might be relevant: type errors within or between certain modules could be suppressed at the will of the programmer, if some modules have not been yet fully adapted to the type system, for example.

A monitoring system We present a simple runtime monitoring system for FJ$^?$ below. It is based on a guard insertion transformation $\Gamma \vdash \mathtt{e} \rightsquigarrow \mathtt{e}' : \mathtt{T}$, which translates FJ$^?$ expressions to FJ$^?_{\mathrm{G}}$, which is FJ$^?$ augmented with *guarded expressions* $\langle \mathtt{C} \rangle \mathtt{e}$ and *guarded method invocations* $\langle \mathtt{e.m}(\overline{\mathtt{e}}) \rangle_{\overline{\mathtt{T}}}$. Guards resemble casts, but they only log any detected type errors (or report them as warnings) and allow the program execution to always continue.

If the base type of a method invocation is known during type checking, we insert runtime guards $\langle \cdot \rangle$ for arguments using a $\langle\langle \mathtt{T} \Leftarrow \mathtt{S} \rangle\rangle$ form, but to limit the number of guards, only when we cannot statically check the compatibility of the types:

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{C} \qquad \mathit{mtype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{S}} \to \mathtt{S} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}} \qquad \overline{\mathtt{T}} \lesssim \overline{\mathtt{S}}}{\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) \rightsquigarrow \mathtt{e.m}(\langle\langle \mathtt{S}_1 \Leftarrow \mathtt{T}_1 \rangle\rangle \mathtt{e}_1, \ldots) : \mathtt{S}}$$

The $\langle\langle \ldots \rangle\rangle$ form is just a notational convenience and can be simplified away or replaced with a guard during the transformation. We use $\Longrightarrow$ to represent this transformation (note that T may be ? in G-INSERT):

$$\frac{\mathtt{D} <: \mathtt{C}}{\langle\langle \mathtt{C} \Leftarrow \mathtt{D} \rangle\rangle \mathtt{e} \Longrightarrow \mathtt{e}} \text{ (G-IGNORE1)} \qquad \frac{\mathtt{T} \not<: \mathtt{C}}{\langle\langle \mathtt{C} \Leftarrow \mathtt{T} \rangle\rangle \mathtt{e} \Longrightarrow \langle \mathtt{C} \rangle \mathtt{e}} \text{ (G-INSERT)}$$

$$\langle\langle ? \Leftarrow \mathtt{T} \rangle\rangle \mathtt{e} \Longrightarrow \mathtt{e} \text{ (G-IGNORE2)}$$

A few interesting new evaluation rules are needed for the guards. A successful guard is ignored, while a failed guard causes an error to be logged. However, unlike a failed cast, a failed guard does not terminate the evaluation of the program:

$$\frac{\mathtt{C} <: \mathtt{D}}{\langle \mathtt{D} \rangle \mathtt{new\ C}(\overline{\mathtt{e}}) \longrightarrow \mathtt{new\ C}(\overline{\mathtt{e}})} \qquad \frac{\mathtt{C} \not<: \mathtt{D} \qquad \textit{log error}}{\langle \mathtt{D} \rangle \mathtt{new\ C}(\overline{\mathtt{e}}) \longrightarrow \mathtt{new\ C}(\overline{\mathtt{e}})}$$

Inserting guards to method invocations with base type ? is postponed until evaluation, as shown below. Note that a guarded method invocation $\langle \mathtt{e.m}(\overline{\mathtt{e}})\rangle_{\overline{\mathtt{T}}}$ is distinct from guarded expressions and has a different evaluation rule. We omit the transformation rules for `new` expressions and method bodies; these are straightforward. Here is the transformation rule for guarded method invocations:

$$\frac{\Gamma \vdash \mathtt{e} : ? \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}}{\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) \rightsquigarrow \langle \mathtt{e.m}(\overline{\mathtt{e}})\rangle_{\overline{\mathtt{T}}} : ?}$$

The evaluation rule for guarded method invocation looks up the method signature based on the runtime type and inserts necessary guards for the arguments:

$$\frac{\mathit{mtype}(\mathtt{m},\mathtt{C}) = \overline{\mathtt{S}} \to \mathtt{S}}{\langle \mathtt{new\ C}(\overline{\mathtt{e}})\mathtt{.m}(\overline{\mathtt{e}})\rangle_{\overline{\mathtt{T}}} \longrightarrow \mathtt{new\ C}(\overline{\mathtt{e}})\mathtt{.m}(\langle\langle \mathtt{S}_1 \Leftarrow \mathtt{T}_1\rangle\rangle \mathtt{e}_1, \ \ldots)}$$

The system outlined above reports a runtime error for all programs that bind an instance of an incompatible type to a typed variable at runtime. After reporting the first runtime type error, the declared types do not always hold any more: a variable may have a reference to a value of an invalid type. In order to avoid getting multiple reports from a single error, we only verify type errors in the dynamic-static boundary. Thus some subsequent but related type errors within a statically-typed section of a program may be suppressed.

We can alter the system slightly to catch all runtime type errors, at the cost of introducing a potentially large number of additional type guards (and reported type errors), by replacing the rules G-INSERT1 and G-IGNORE with this new rule:

$$\langle\langle \mathtt{C} \Leftarrow \mathtt{T}\rangle\rangle \mathtt{e} \Longrightarrow \langle \mathtt{C}\rangle \mathtt{e} \quad \text{(G-INSERT')}$$

Alternatively, we could start using guards inserted by the rule G-INSERT' only after encountering the first runtime type error. This would catch exactly the same errors as always using G-INSERT', but we would need to have two representations of the program and the ability to switch between them during program execution.

Discussion This approach has a number of benefits. As monitoring is optional, we still enable a simple, purely dynamically-typed implementation to run all programs. Thus also the type system can be purely optional. Of course, a dynamically-typed implementation will not report all runtime type errors, or error messages may show up far away from the actual source of the error. Many scripting languages have multiple independent implementations. For example, Python has C, JVM and .NET implementations (and more). Not all of the implementation maintainers have to spend the effort of implementing runtime type monitoring, which can be considerable for a more complex language and type system.

Runtime monitoring enables reporting type errors in the boundary of statically-typed and dynamically-typed code, similar to gradual typing. Multiple configurable and *pluggable* monitoring implementations may be available, since they all retain the dynamically-typed semantics (modulo the details of error logging).

Novice or casual programmers do not have to learn the type system to use statically-typed code; they can safely strip away or ignore all type annotations in statically-typable

code and incorporate it in their dynamically-typed programs. If a type system is mandatory, it is also almost essential for programmers to understand it, or they may have difficulty interacting with statically-typed code, including library code that is statically typed.

More complex type systems than our FJ$^?$ require adding potentially long-lived object or function wrappers to track runtime type errors precisely as values are passed across the static-dynamic boundary. These may impose a significant performance overhead for certain implementations. High-performance Alore implementations do not have to support runtime monitoring of type errors, or the support may be limited. During development, a less efficient implementation with better error reporting can be used.

Practical experience with using the monitoring system is needed to evaluate the different guard insertion strategies outlined above, and how they affect the implementation of virtual machines and compilers.

4 Extending the Type System

The core type system presented above does not model many interesting aspects of Alore. In this section we informally describe additional features that we feel are important additions for a Python-like language. These features are based on insights gained from implementing a pluggable type system for Alore that uses static analysis for inferring types. The complexity and poor scaling behaviour of global type inference directed us to adopt our current approach with explicit type annotations.

Bindable interfaces Alore supports explicit named interfaces. Alore classes can be extended with new interface mappings by clients, outside the class definition. This gives us some benefits of structural subtyping in a nominal type system. We call this feature *bindable interfaces*.

Dynamically-typed code often takes advantage of implicit, ad-hoc interfaces ("duck typing"), that were not envisaged by the implementers of classes. For example, consider a function that can deal with any objects that have a `close()` method (note that we use a slightly more Python-like syntax in this section):

```
def finish(o) { o.close(); }
```

We can define an interface and bind it to any classes that happen to implement this method using a `bind` declaration, even if the original implementers did not foresee this possibility, and without needing to modify the source code or definitions of these classes. In the example below, we assume that classes `Stream` and `ServerSocket` provide `close()`:

```
interface Closeable {
  bind Stream;
  bind ServerSocket;
  def close() : () -> Unit;
} ...
def finish(o) : (Closeable) -> Unit { o.close(); }
```

Now `finish` accepts one argument of type `Closeable`. If `finish` is called with a dynamically-typed argument that does not implement `Closeable` but supports `close()`, the runtime monitoring system will log an error, but the call will succeed. Adding new interface declarations to existing dynamically-typed programs is safe, even if some type declarations or interface bindings turn out to have errors that were not caught in testing.

Bindable interfaces can also be used as a partial replacement for union types. For example, consider a variable that can hold either an `Int` or a `Str` object (both types are built-in). `Int` and `Str` are unrelated types, but we can define an interface `IntOrStr`, and bind it to the built-in types `Int` and `Str`:

```
interface IntOrStr {
    bind Int;
    bind Str;
} ...
IntOrStr x; Str s;
x = 1; x = "s"; s = (Str)x; // No type errors
```

The interface `IntOrStr` could define functionality supported by both `Int` and `Str` (in this example, it has no methods). A cast expression can be used to get back to the `Int` or `Str` type. This is more descriptive and statically catches more programming errors than using the type `Object` or ? to hold the heterogeneous reference `x`.

Intersection types Our semantics do not support Java-style method overloading based on method signatures, since signatures can be erased during evaluation. An overloaded method can, however, be represented using an *intersection type* [14]: a single implementation can have multiple types. For example, a multiply-and-add method accepts both integer and floating point arguments:

```
: (Int, Int, Int) -> Int
: (Float, Float, Float) -> Float
def mulAdd(x, y, z) {
  return x * y + z;
}
```

The method `mulAdd` has an intersection type with two components; it can be invoked either with 3 `Int` or 3 `Float` arguments. Note that the Java-style method syntax of FJ$^?$ is inconvenient for representing intersection types.

We must restrict intersection types somewhat compared to static method overloading in Java. The runtime monitoring system only sees runtime types of the arguments, and it must be able to deduce the called signature from the runtime types uniquely. So if `B` inherits `A`, we cannot generally have an intersection type $(\texttt{A}) \to \texttt{X} \wedge (\texttt{B}) \to \texttt{Y}$, since the return type is not unique when the argument has runtime type `B`: we could use either of the items in the intersection type.

Intersection types are also useful for representing callable, function-like objects. Each Alore class is represented by an object of type `Type`, similar to the Python type `type`. Applying a `Type` object constructs instances of the type, without requiring a keyword such as `new`. A type object can also be used for querying the runtime type

inclusion of values. Type objects can be assigned to variables and manipulated like any other values:

```
? t = Int;          // Declare t, initialize to type object Int
t("15");            // Evaluates to Int instance 15
3 instanceof t;     // True
```

We model the above behaviour by giving type objects an intersection type. A valid type for variable `t` in the example is $\texttt{Type} \wedge (\texttt{Str}) \to \texttt{Int}$ (here $\to$ binds more tightly than $\wedge$). The `Type` component enables us to use the `instanceof` operator while the second component allows application. Function objects can also be applied, but they are not types and cannot be used as the right operand of `instanceof`. A similar function object would have a type $\texttt{Function} \wedge (\texttt{Str}) \to \texttt{Int}$ or simply $(\texttt{Str}) \to \texttt{Int}$.

Genericity Support for generic types and generic functions is important for static type safety of container types. Casts and the ? type can be used to work around the lack of genericity, but at the cost of sacrificing some static type safety.

Different runtime monitoring implementations might check generic types in different ways. Type errors in generic container items could be checked eagerly (verifying the contents of a dynamically-typed container when binding to a statically-typed variable) or lazily (only when reading or modifying the container). Combining runtime monitoring of type errors with genericity and intersection types is not yet well-understood.

Arbitrary mixing of dynamic and static types For simplicity, FJ$^?$ does not allow a method with ? types in the signature to override a statically-typed method, or vice versa. To support that, the runtime monitoring system can use techniques similar to Siek and Taha's gradual typing for objects [15], but adapted to a nominal type system.

Tuple types We include first-class tuple types as a straightforward extension, as tuples are common in idiomatic Python and Alore programs.

5 Comparison with Python

In this section we compare Alore with Python to highlight common features and differences. Other imperative scripting languages such as Ruby and Lua also share many of the common properties. Alore has enough convenience features for practical experimentation using fairly complex programs, and as we showed in Section 2, Alore is well-suited for formal study of pluggable type systems.

Common properties In both Alore and Python, every value is a reference to a class instance. Both languages use call by value. There is no semantic difference between values of primitive and class-based types, unlike Java, so there is no need for boxing and unboxing in the semantics.

Alore and Python have similar sets of basic program structuring primitives (variables, expressions, statements, functions, classes with single dispatch, modules and exceptions). They also share a similar expression syntax, similar basic types (but Alore has fewer) and similar programming idioms related to basic types. For example, returning multiple values from a function is implemented as returning a tuple. Parallel

assignment is supported for tuples and array-like sequences. Neither Alore nor Python has separate types for characters and strings. Array-like and hash-table-like containers are used more commonly than linked lists.

Differences Unlike Python, Alore classes and objects currently do not allow members to be added or redefined at runtime. Python also has a very general *eval* and introspection capabilities in the built-ins, while Alore has more restricted introspection capabilities. We argue that not only does this simplify adding static types and runtime monitoring of type errors to Alore, but it also improves runtime efficiency, supports concurrency more naturally and is preferable from a software engineering point of view by making large programs easier to understand and maintain. We are planning to add support for additional dynamic features to Alore, but only when their undesirable effects can be isolated from those parts of a program that do not use these features.

The Python standard library[3] contains commonly-used features that are subtly difficult to model *precisely* in a type system. For example, file object construction could benefit from dependent types: `open('f.ext', 'r')` constructs a file object that produces `str` objects, but after changing the call slightly to `open('f.ext', 'rb')`, the resulting object produces `bytes` objects. In contrast, Alore has separate constructors for binary and text files. We have designed the Alore standard library carefully to provide static type safety with a relatively simple type system. Only inherently dynamic library functionality such as reflection use the ? type.

6 Related Work

The Strongtalk variant of Smalltalk [4] has a pluggable type system based on structural subtyping. Bracha [3] argues for pluggable type systems, and some of his arguments are similar to ours. Our biggest contributions over previous work on pluggable types are the use of optional runtime monitoring to track type errors across the static-dynamic boundary and the introduction of bindable interfaces.

Our runtime monitoring was inspired by Siek and Taha's gradual typing [16, 15] and Findler and Felleisen's contracts for higher-order functions [7]. They have some similar goals as our system, but they use runtime type checking that affects the semantics of programs and causes programs to terminate on type errors. Our approach also resembles tools such as Valgrind [12] that allow instrumenting programs in order to detect various runtime errors, while otherwise retaining the original program semantics.

The concept of blame can be used to track the origin of type errors in a mixed type system [7, 19]. Gray et al. [9] use contracts with blame tracking for detecting and reporting runtime type errors in a system for Java-Scheme interoperability. Typed Scheme [18] enables mixing dynamically-typed and statically-typed modules in the same program, and provides support for contracts and blame, but the language lacks object-oriented features.

Ina and Igarashi [11] extend FJ with gradual typing, and their work has many similarities with our FJ$^?$. They also discuss the addition of generics to a gradual type system.

[3] We are referring to Python 3.x, which is significantly different from previous Python versions.

Ahmed et al. [1] add parametric polymorphism to a system resembling gradual types using sealing. Their system also supports blame.

Cecil [6], Thorn [2] and Boo [13] are examples of languages that support mixing dynamic and static typing, but with *mandatory* type systems that affect execution semantics. Cecil is an object-based language based on multiple dispatch. Cecil allows adding new supertypes, including new inherited features, to existing types. This precludes separate compilation, unlike our bindable interfaces. Cecil does not track runtime type errors at the dynamic-static boundary, and in this respect the type system is optional. Boo is class-based language for the CLI with a syntax that resembles Python, but Boo's basic types and library functionality are mostly inherited from the .NET framework. Thorn is a class-based language for the JVM. Thorn supports "like types" [21] for mixing dynamic and static types. They resemble our pluggable types without runtime monitoring: static type checking does not guarantee the absence of runtime type errors when using like types, and no wrappers are required at runtime.

Wright and Cartwright developed a *soft type system* for Scheme [20] that uses static analysis to remove redundant runtime type checks from dynamically-typed programs. Furr et al. [8] use profile-guided static analysis to find type errors in Ruby programs. Using whole-program static analysis for type checking makes it difficult to predict the impact of even small program changes, resulting in brittle systems that can be difficult to use. We experimented with static analysis before adopting our current approach.

7 Conclusions and Future Work

We formalised a pluggable type system for an extension of FJ and showed how pluggable type systems can be extended with runtime monitoring of type errors in the spirit of gradual typing. We also discussed extending the pluggable type system to a Python-like language.

This is still work in progress. A potential next step is to formalise the runtime monitoring of type errors for the extended type system, to add support for blame and to prove properties of the extended system. We will also implement a type checker and runtime monitoring system for the extended type system. Finally, we plan to translate existing Python programs to Alore and adapt them to static typing to assess the practicality of our approach.

Acknowledgements

We thank Alan Mycroft and the anonymous reviewers for valuable feedback on earlier drafts of this paper. This research was supported by the Academy of Finland, the Emil Aaltonen Foundation and the Engineering and Physical Sciences Research Council.

References

1. Ahmed, A., Findler, R.B., Matthews, J., Wadler, P.: Blame for all. In: STOP '09: Proceedings for the 1st workshop on Script to Program Evolution. pp. 1–13 (2009)

2. Bloom, B., Field, J., Nystrom, N., Östlund, J., Richards, G., Strniša, R., Vitek, J., Wrigstad, T.: Thorn: robust, concurrent, extensible scripting on the JVM. In: OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications. pp. 117–136 (2009)
3. Bracha, G.: Pluggable type systems. In: OOPSLA Workshop on Revival of Dynamic Languages (2004)
4. Bracha, G., Griswold, D.: Strongtalk: typechecking Smalltalk in a production environment. In: OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications. vol. 28, pp. 215–230 (October 1993)
5. Bracha, G., Odersky, M., Stoutamire, D., Wadler, P.: Making the future safe for the past: adding genericity to the Java programming language. In: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. pp. 183–200. OOPSLA '98 (1998)
6. Chambers, C., the Cecil Group: The Cecil language: Specification and rationale. Tech. rep., Department of Computer Science and Engineering, University of Washington (February 2004)
7. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming. vol. 37, pp. 48–59 (September 2002)
8. Furr, M., An, J.h.D., Foster, J.S.: Profile-guided static typing for dynamic scripting languages. In: OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications. pp. 283–300 (2009)
9. Gray, K.E., Findler, R.B., Flatt, M.: Fine-grained interoperability through mirrors and contracts. In: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. pp. 231–245 (2005)
10. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. 23(3), 396–450 (2001)
11. Ina, L., Igarashi, A.: Towards gradual typing for generics. In: STOP '09: Proceedings for the 1st workshop on Script to Program Evolution. pp. 17–29 (2009)
12. Nethercote, N., Seward, J.: Valgrind: A Program Supervision Framework. Electronic Notes in Theoretical Computer Science 89(2), 44–66 (October 2003)
13. de Oliveira, R.B., et al.: The Boo programming language (2011), `http://boo.codehaus.org/`
14. Pierce, B.C.: Programming with intersection types, union types, and polymorphism. Tech. Rep. CMU-CS-91-106, Carnegie Mellon University (February 1991)
15. Siek, J., Taha, W.: Gradual typing for objects. In: ECOOP '07. pp. 2–27 (2007)
16. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming Workshop (September 2006)
17. Tobin-Hochstadt, S., Felleisen, M.: Interlanguage migration: from scripts to programs. In: OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. pp. 964–974 (2006)
18. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of Typed Scheme. In: POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 395–406 (2008)
19. Wadler, P., Findler, R.B.: Well-typed programs can't be blamed. In: ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems. pp. 1–16 (2009)
20. Wright, A.K., Cartwright, R.: A practical soft type system for Scheme. ACM Trans. Program. Lang. Syst. 19(1), 87–152 (January 1997)
21. Wrigstad, T., Nardelli, F.Z., Lebresne, S., Östlund, J., Vitek, J.: Integrating typed and untyped code in a scripting language. In: POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 377–388 (2010)

Position Paper: Dynamically Inferred Types for Dynamic Languages

Jong-hoon (David) An[1], Avik Chaudhuri[2], Jeffrey S. Foster[3], and Michael Hicks[3]

[1] Epic Systems Corporation, Madison, WI, USA
[2] Advanced Technology Labs, Adobe Systems, San Jose, CA, USA
[3] University of Maryland, College Park, USA

Over the past few years we have been developing Diamondback Ruby (DRuby), a tool that brings static type inference to Ruby [1], a dynamically typed, object-oriented language. Developing DRuby required creating a Ruby front-end, which was extremely challenging: like other dynamic languages, Ruby has a complex, yet poorly documented syntax and semantics, which we had to carefully reverse-engineer. Writing our front-end took well over a year, and now that Ruby 1.9 is available, we are faced with the daunting prospect of significant additional effort to discover how the language has changed and to extend our front-end accordingly. We suspect that maintaining a static analysis system for other dynamic languages, such as Perl or Python, is similarly daunting.

To remedy this situation, we recently introduced a new program analysis technique for dynamic languages: *constraint-based dynamic type inference*, which requires no language front-end, but instead uses introspection features to gather information at run time and infer static types [2]. More precisely, at run-time we introduce type variables for fields, method arguments, and method return values. As values are passed to those positions, we dynamically *wrap* them in proxy objects to track the associated type variables. We also allow methods to have trusted type annotations, which are maintained dynamically at run time. As wrapped values are used, we generate *subtyping constraints* on the associated type variables. We solve those constraints at the end of one or more program runs, which produces a satisfying type assignment, if one exists. Importantly, despite relying on dynamic runs, we can prove a soundness theorem: if the dynamic runs from which types are inferred cover every path in the control-flow graph of every method of a class, then the inferred types for that class's fields and methods are sound for all possible runs. (Currently, this coverage criterion must be checked manually, though we could potentially automate the check.) Note this coverage criterion is in contrast to requiring that every *program* path is covered.

We have implemented this technique for Ruby, as a tool called Rubydust (where "dust" stands for dynamic unraveling of static types). An important property of Rubydust is that, rather than a standalone tool, it is a Ruby library that is loaded at run-time just like any other library. To operate, Rubydust uses Ruby's rich introspection features to wrap objects, intercept method calls, and store and retrieve any type annotations supplied by the programmer. Thus far, we have run Rubydust on a number of small programs, and have found

that Rubydust produces correct, readable types. We expect that our approach could be implemented in the same way in other languages that have sufficient introspection facilities.

We believe that it is worth exploring whether constraint-based dynamic type inference is a practical means to adding static typing support to dynamic languages. In particular, we believe that users will often want to develop their scripts without types at first, and then might like to add types (as checked annotations) later. Constraint-based dynamic type inference could be quite useful for discovering possible annotations automatically. We would hope that Rubydust's approach, made practical, could be applied to other dynamic languages such as Python or Perl, and to gradually typed languages such as ActionScript.

Before we can claim victory, however, there are a number of challenges that require further research. We list a few here. First, Rubydust's performance overhead is significant: an instrumented program can be as much as 1000× slower than the uninstrumented original. We believe the major source of slowdown is in the additional levels of indirection introduced by wrapping all objects. Thus we are currently investigating ways to improve performance by integrating wrapping directly into the Ruby interpreter. Second, Rubydust's inference is currently limited to nominal and structural types involving unions; Rubydust cannot infer intersection or polymorphic types, though it can understand such types in annotations. Our experience with DRuby gives us reason to believe that inferring intersection and polymorphic types would be very useful, but it makes the inference problem significantly harder. Third, inference relies on instrumenting the running program, but some type-relevant events escape instrumentation. In particular, we know of no way to intercept calls to blocks (i.e., closures), nor do we yet know how to *re*instrument a program after it has called `eval` to create new code or definitions. (We suspect this might be accomplished by redefining `eval`, though we have not worked out the details.) Finally, we wish to understand the practical benefits of types: once we have them, what do we do with them? One possibility is to check them, e.g., at method call boundaries. This would permit reporting errors earlier, and the results might be more informative. Another question is whether we have chosen the right design point for our type language: should types be more expressive, to convey richer properties, or perhaps less expressive, so they are easier to read? When are programmers most interested in types, e.g., during maintenance, or during initial development? Many of these questions require careful human studies, which we plan to undertake once we have worked out some technical issues.

References

1. Michael Furr. *Combining Static and Dynamic Typing in Ruby.* PhD thesis, University of Maryland, College Park, 2009.
2. Jong hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic Inference of Static Types for Ruby. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Austin, TX, USA, January 2011.

Gradual Information Flow Typing

Tim Disney and Cormac Flanagan

University of California Santa Cruz

Abstract. We present a method to support the gradual evolution of secure scripts by formalizing an extension of the simply-typed lambda calculus that provides information flow constructs. These constructs allow initially insecure programs to evolve via targeted refactoring and to provide dynamic information flow guarantees via casts, as well as static information flow guarantees via labeled types.

1 Introduction

Several decades of software engineering experience has demonstrated that writing "correct" software is close to impossible, due to the inherent complexity of software systems and the fallible nature of human programmers. Consequently, relying on security to be an emergent property of software is unwise. We argue instead that security properties, such as data confidentiality and integrity, should be monitored and enforced by small trusted parts of the code base, with help from the run-time system where appropriate.

In practice, programmers are initially concerned more with functionality than with security. It is only once a system has proven useful, and has attracted users and potential attackers, that security concerns become dominant. While it might be preferable to address security concerns right from the start of a project, competitive pressures often dictate quickly developing an initial (perhaps insecure) system that helps clarify the requirements and gain a market foothold, and then to evolve the system with additional features, including security guarantees. Hence, we would like to support a development methodology whereby the programmer first develops an initial, insecure system, and then incrementally refactors the system to add data confidentiality and integrity guarantees via information flow tracking.

Much prior work has addressed information flow security. Most of this work has focused on static type systems such as JFlow [15], Jif [14], and others [29, 16], which involve significant up-front costs. More recent investigations explore dynamic information flow [4, 5, 10], which requires less up-front investment but which cannot document security properties as types.

In this paper, we explore some initial steps towards realizing the vision of gradual evolution from untyped scripts into security typed applications. Since prior work has addressed evolving dynamic scripts into typed code [1, 27, 12, 11], the starting point for our development is a typed language. We explore how to gradually extend programs in this language with security guarantees and with security types.

To support gradual evolution of security properties, our approach provides both static and dynamic information flow guarantees. We use dynamic information flow tracking for all code, including conventionally typed code that has not been refactored to express information flow properties in the type system. Our language includes a *labeling* operation for marking data as private, and a *cast* operation for checking the labels on data. Both operations naturally extend to higher-order data by treating contravariant function arguments appropriately. The cast operation may fail if applied to incorrectly-labeled data; in this case either the term inside the cast or the context of the cast is blamed, which we call positive and negative blame respectively.

For any program, including those without security types, our approach guarantees *termination insensitive non-interference* (TINI), which means that private inputs cannot flow into or influence public outputs. Attempts to violate TINI results in cast failures. We assume a *label lattice* for expressing data confidentiality and integrity properties. For simplicity, we often use a two-element lattice with a public or low confidentiality label (L) and a private or high confidentiality label (H), but the approach generalizes to any lattice.

In addition to tracking information flow dynamically, we enrich the type system to express invariants regarding security labels on the underlying data. Our preservation theorem states that if a term t has type $\texttt{Int}^k$, where k is a security label, then t can only evaluate to a value n^m, where n is an integer and the label m satisfies $m \sqsubseteq k$. Thus, the type system documents a conservative upper bound on the label of the resulting value.

In our language, each value and type has an associated security label. To support legacy code, an unlabeled value implicitly has label $\bot$ (the bottom element in the lattice) indicating that the value is not confidential. Conversely, an unlabeled type implicitly has label $\top$, allowing it to describe values with arbitrary labels, since any label m satisfies $m \sqsubseteq \top$. In this manner, conventionally typed code can interoperate with new precisely typed code, with casts at interfaces between the two typing disciplines.

If the entire application has precise security types, then there is no need for downcasts and for dynamic tracking of information flow labels. This approach has been explored in depth in prior systems such as JFlow [15]. In contrast, the novelty of this paper is that dynamic label tracking enables downcasts, and avoids the need to statically document precise security types throughout the entire application all at once. Instead, the application can gradually evolve from (1) conventionally typed with no security guarantees, to (2) having casts and dynamic information flow guarantees, to (3) being precisely typed with no casts and with static information flow guarantees. Additionally, this evolution process can stop or pause at any point in the middle, depending on engineering, economic, and security requirements, as each intermediate step is a valid well-typed program (albeit with different run-time guarantees).

1.1 Motivating Example

To illustrate the benefits of gradual information flow, consider the following code fragment which deals with sensitive salary information:

$$
\begin{array}{l}
\texttt{let } \mathit{age} : \texttt{Int} = 42 \\
\texttt{let } \mathit{salary} : \texttt{Int} = 58000 \\
\texttt{let } \mathit{intToString} : \texttt{Int} \to \texttt{Str} = \ldots \\
\texttt{let } \mathit{print} : \texttt{Str} \to \texttt{Unit} = \lambda s{:}\texttt{Str}.\ \ldots \\
\mathit{print}(\mathit{intToString}(\mathit{salary}))
\end{array}
$$

This code does not track the flow of sensitive information. After some embarrassing salary leaks, the program manager might want to "harden" the script to limit the flow of sensitive information

In the following revised code, the labeling operation $(58000 : \texttt{Int} \Rrightarrow \texttt{Int}^H)$ marks data as private, and the cast $(s : \texttt{Str} \Rightarrow^p \texttt{Str}^L)$ checks that data is public:

$$
\begin{array}{l}
\texttt{let } \mathit{age} : \texttt{Int} = 42 \\
\texttt{let } \mathit{salary} : \texttt{Int} = 58000 : \texttt{Int} \Rrightarrow \texttt{Int}^H \\
\texttt{let } \mathit{intToString} : \texttt{Int} \to \texttt{Str} = \ldots \\
\texttt{let } \mathit{print} : \texttt{Str} \to \texttt{Unit} = \lambda s{:}\texttt{Str}.\ \texttt{let } s = (s : \texttt{Str} \Rightarrow^p \texttt{Str}^L)\ \texttt{in} \ldots \\
\mathit{print}(\mathit{intToString}(\mathit{salary}))
\end{array}
$$

The runtime system tracks the flow of information through all code. Since the *intToString* library function is applied to a confidential argument, it produces a confidential result, and so the cast inside *print* will fail at runtime. Thus, independent of bugs in the rest of the code, *print* ensures that confidential data is never printed.

As a next step, we wish to document and verify information flow invariants using the type system. We begin by extending the code with explicit labels on types. Note that $\texttt{Int}^H$ is the most general integer type, since these potentially private integers can store both public and private data.

$$
\begin{array}{l}
\texttt{let } \mathit{age} : \texttt{Int}^L = 42 \\
\texttt{let } \mathit{salary} : \texttt{Int}^H = 58000 : \texttt{Int}^L \Rrightarrow \texttt{Int}^H \\
\texttt{let } \mathit{intToString} : \texttt{Int} \to \texttt{Str} = \ldots \qquad \text{// unrefactored module} \\
\texttt{let } \mathit{print} : \texttt{Str}^H \to \texttt{Unit}^H = \lambda s{:}\texttt{Str}^H.\ \texttt{let } s = (s : \texttt{Str}^H \Rightarrow^p \texttt{Str}^L)\ \texttt{in} \ \ldots \\
\mathit{print}(\mathit{intToString}(\mathit{salary}))
\end{array}
$$

The above code incorporates information flow types but does not yet provide static guarantees since *print* accepts H values (at least statically). To provide static guarantees, we first refine *print*'s argument type to specify that it only accepts public data. This refactoring requires introducing a variant of the *intToString* function, called *intToStringL*, for handling public data, using a cast to specify that *intToString* has the desired behavior of mapping public

Figure 1: λ_{gif} Syntax

$\iota ::= \texttt{Int} \mid \texttt{Bool} \mid \texttt{Str}$	*Base Types*
$a, b ::= \iota \mid A \to B$	*Raw Types*
$A, B ::= a^k$	*Labeled Types*
$t, s ::= v \mid x \mid t\ s \mid op\ \overline{t} \mid t\colon A \Rrightarrow B \mid t\colon A \Rightarrow^p B$	*Terms*
$r ::= c \mid \lambda x\colon A.\ t$	*Raw Values*
$v, w ::= r^k$	*Labeled Values*
k, l, m	*Labels*
$\Gamma ::= \emptyset \mid \Gamma, x\colon A$	*Typing Environment*

integer inputs to public string results.

$\texttt{let}\ age : \texttt{Int}^L = 42$
$\texttt{let}\ salary : \texttt{Int}^H = 58000\colon \texttt{Int}^L \Rrightarrow \texttt{Int}^H$
$\texttt{let}\ intToString : \texttt{Int} \to \texttt{Str} = \ldots$ // unrefactored module
$\texttt{let}\ intToStringL : \texttt{Int}^L \to \texttt{Str}^L = intToString\colon (\texttt{Int}^H \to \texttt{Str}^H) \Rightarrow^q (\texttt{Int}^L \to \texttt{Str}^L)$
$\texttt{let}\ print : \texttt{Str}^L \to \texttt{Unit}^L = \lambda s\colon \texttt{Str}^L.\ \ldots$
$print(intToStringL(salary))$

Using these more precise types, bugs such as $print(intToStringL(salary))$ now are revealed at compile time. The programmer then corrects the code to the intended $print(intToStringL(age))$, which passes both static and dynamic security checks. Note that this security typed code interoperates with the legacy *intToString* module via the security interface specification (aka cast) inside *intToStringL*.

2 The Gradual Information Flow Language

We formalize our ideas for gradual security for an idealized language called λ_{gif}, which extends the simply typed λ-calculus with gradual information flow.

The syntax of λ_{gif} is presented in figure 1. Raw types a include integers (`Int`), booleans (`Bool`), strings (`Str`), and function types ($A \to B$). Types A are labeled raw types (a^k). Security labels (k) denote the confidentiality or integrity of a particular value or term. The set of labels form a lattice, with a ordering operation $\sqsubseteq$, join operation $\sqcup$, least element $\bot$, and top element $\top$.

Terms t include variables (x), function applications ($t\ s$), primitive operations ($op\ \overline{t}$), and values (v). Raw values r can be either constants (c), such as 42 or `true`, or functions ($\lambda x\colon A.\ t$). Values v are labeled raw values (r^k). The classification operation ($t\colon A \Rrightarrow B$) adds a label to a value. For example, $3^L : \texttt{Int}^L \Rrightarrow \texttt{Int}^H$ evaluates to 3^H.

Casts ($t\colon A \Rightarrow^p B$) attempt to coerce a term t of type A into a new type B. If the labels on the value are not compatible with type B, the cast will fail, in which case the blame label p assigns blame to the appropriate code fragment. For example, attempting to *downcast* a private integer 42^H to public via the cast $42^H\colon \texttt{Int}^H \Rightarrow^p \texttt{Int}^L$ will fail. An *upcast* of a public integer $42^L\colon \texttt{Int}^L \Rightarrow^p \texttt{Int}^H$ to a private integer however will succeed, and return the value 42^L unchanged. That is, casts do not change values, they only change static types (or else fail).

2.1 Operational Semantics

We formalize the dynamic semantics of λ_{gif} using the big-step evaluation relation $t \Downarrow v$, which evaluates a term t to value v: see figure 2. The [E-APP] rule for function application $(t\ s)$ evaluates t to a function $(\lambda x\colon A.\ t_1)^k$ with a security label k, evaluates the argument s to a value v, and then evaluates the substituted function body $t_1[x := v]$ to a labeled value r^m. The result of the application depends on the function that is invoked, so the rule adds the label k of the callee to the resulting value, yielding $r^{m \sqcup k}$.

The [E-PRIM] rule for primitive operations $(op\ \overline{t})$ refers to the δ_{op} function, which defines the semantics of primitive operations on raw values.

There are three rules to support the *cast* operation, which checks if a runtime label is compatibile with a specified static label. If the check fails then the rules use a *blame label* p to identify the code that is at fault. We say that *positive* blame (p) means the term within the cast is at fault and *negative* blame ($\overline{p}$) means the context containing the cast is at fault. The negation of negative blame is the original blame label: $\overline{\overline{p}} \stackrel{\text{def}}{=} p$.

The [E-CAST-BASE] rule is for casts of base types ι (i.e. non-functions). The cast $(t\colon \iota^k \Rightarrow^p \iota^l)$ evaluates t to a value r^m and checks that the label m on the value is less than the label l on the target type; if not then the [B-CAST-BAD] rule will blame p. The other [B-...] rules simply propagate blame.

The [E-CAST-FN] rule for $t\colon (A \to B)^k \Rightarrow^p (A' \to B')^l$ is similar to [E-CAST-BASE], except that the value r^m produced by t is wrapped in a new function:

$$(\lambda x'\colon A'.\ (r^m\ (x'\colon A' \Rightarrow^{\overline{p}} A))\colon B \Rightarrow^p B')^{\bot}$$

which satisfies the target type $(A' \to B')$. The wrapper function is used to cast the argument and result to the appropriate types. The argument x' is cast from the new type A' to the original type A, which the original function r can accept. The blame in this cast is inverted $\overline{p}$ to indicate that if this cast fails blame is assigned to the cast context (which invoked the function with an incompatible argument). The result of calling the function is cast to B'.

For an example of the cast rule, consider a function f of type $\texttt{Int}^L \to \texttt{Bool}^L$. If we strengthen its range via the cast $f\colon (\texttt{Int}^L \to \texttt{Bool}^L) \Rightarrow^p (\texttt{Int}^L \to \texttt{Bool}^H)$, calling the resulting wrapper function f' : $\texttt{Int}^L \to \texttt{Int}^H$ will always succeed since the result res of f is guaranteed to be public and f' casts res to a private boolean, which will always succeed. If, however, we strengthen the domain with the cast $f\colon (\texttt{Int}^L \to \texttt{Bool}^L) \Rightarrow^p (\texttt{Int}^H \to \texttt{Bool}^L)$, the argument x' must be downcast $(x'\colon \texttt{Int}^H \Rightarrow^{\overline{p}} \texttt{Int}^L)$ and will fail when x' is private.

The final two rules support *classification*, marking data as having higher confidentiality (or alternatively lower integrity). The [E-CLASSIFY-BASE] rule is used for classifying base types. The classification $t\colon \iota^k \Rrightarrow \iota^l$ adds the target label l to the data by evaluating t to a value r^m and joining l to label m.

The [E-CLASSIFY-FN] rule for $t\colon (A \to B)^k \Rrightarrow (A' \to B')^l$ returns a wrapper function

$$(\lambda x'\colon A'.\ (r^m\ (x'\colon A' \Rrightarrow A))\colon B \Rrightarrow B')^{m \sqcup l}$$

Figure 2: λ_{gif} Operational Semantics

$\boxed{t \Downarrow v}$

$$\frac{}{v \Downarrow v}\ [\text{E-VALUE}] \qquad \frac{t \Downarrow (\lambda x{:}\, A.\ t_1)^k \quad s \Downarrow v \quad t_1[x := v] \Downarrow r^m}{t\ s \Downarrow r^{m \sqcup k}}\ [\text{E-APP}]$$

$$\frac{r = \delta_{op}(r_1, \cdots, r_n) \quad t_i \Downarrow r_i^{k_i} \quad k = \sqcup k_i}{op\ \overline{t} \Downarrow r^k}\ [\text{E-PRIM}] \qquad \frac{t \Downarrow r^m \quad m \sqsubseteq l}{(t{:}\, i^k \Rightarrow^p i^l) \Downarrow r^m}\ [\text{E-CAST-BASE}]$$

$$\frac{t \Downarrow r^m \quad m \sqsubseteq l \quad v = (\lambda x'{:}A'.\ (r^m\ (x'{:}\, A' \Rightarrow^{\overline{p}} A)){:}\, B \Rightarrow^p B')^{\perp}}{(t{:}\, (A \to B)^k \Rightarrow^p (A' \to B')^l) \Downarrow v}\ [\text{E-CAST-FN}]$$

$$\frac{t \Downarrow r^m}{(t{:}\, i^k \Rrightarrow i^l) \Downarrow r^{m \sqcup l}}\ [\text{E-CLASSIFY-BASE}]$$

$$\frac{t \Downarrow r^m \quad v = (\lambda x'{:}A'.\ (r^m\ (x'{:}\, A' \Rrightarrow A)){:}\, B \Rrightarrow B')^{m \sqcup l}}{(t{:}\, (A \to B)^k \Rrightarrow (A' \to B')^l) \Downarrow v}\ [\text{E-CLASSIFY-FN}]$$

$\boxed{t \Downarrow \mathit{blame}\ p}$

$$\frac{t \Downarrow r^m \quad m \not\sqsubseteq l}{(t{:}\, a^k \Rightarrow^p b^l) \Downarrow \mathit{blame}\ p}\ [\text{B-CAST-BAD}] \qquad \frac{t_i \Downarrow v_i \quad \forall i \in 1..j-1 \quad t_j \Downarrow \mathit{blame}\ p}{op\ \overline{t} \Downarrow \mathit{blame}\ p}\ [\text{B-PRIM}]$$

$$\frac{t \Downarrow \mathit{blame}\ p}{t\ s \Downarrow \mathit{blame}\ p}\ [\text{B-APP-L}] \qquad \frac{t \Downarrow v \quad s \Downarrow \mathit{blame}\ p}{t\ s \Downarrow \mathit{blame}\ p}\ [\text{B-APP-R}]$$

$$\frac{t \Downarrow \mathit{blame}\ p}{(t{:}\, a^k \Rightarrow^p b^l) \Downarrow \mathit{blame}\ p}\ [\text{B-CAST}] \qquad \frac{t \Downarrow \mathit{blame}\ p}{(t{:}\, A \Rrightarrow B) \Downarrow \mathit{blame}\ p}\ [\text{B-CLASSIFY}]$$

that adds the labels in A to the argument and the labels in B' to the result. In addition, the security label of the function type is maintained by giving the wrapper function the label from the original function (m) joined with the label from the function being cast to (l).

3 Termination Insensitive Non-Interference

The central guarantee provided by our semantics is non-interference, which informally states that two runs of the same program that differ only in private data will not produce different public results. We formalize the notion of two terms differing only in private data via the equivalence relation ($\sim_H$) defined in figure 3. Essentially, two values are equivalent if either (1) both are at least as secure as H (where H is an arbitrary lattice element) or (2) their subterms are equivalent.

Theorem 1 (Termination Insensitive Non-Interference).
If $t_1 \sim_H t_2$ and $t_1 \Downarrow v_1$ and $t_2 \Downarrow v_2$ then $v_1 \sim_H v_2$.

Figure 3: Equivalence

$\boxed{v \sim_H v}$

$$\frac{H \sqsubseteq m_1 \quad H \sqsubseteq m_2}{r_1^{m_1} \sim_H r_2^{m_2}} \text{ [EQ-VAL1]} \qquad \frac{r_1 \sim_H r_2}{r_1^{m} \sim_H r_2^{m}} \text{ [EQ-VAL2]}$$

$\boxed{r \sim_H r}$

$$\frac{t_1 \sim_H t_2}{(\lambda x{:}A.\ t_1) \sim_H (\lambda x{:}A.\ t_2)} \text{ [EQ-FUN]} \qquad \frac{}{c \sim_H c} \text{ [EQ-CONST]}$$

$\boxed{t \sim_H t}$

$$\frac{}{x \sim_H x} \text{ [EQ-VAR]} \qquad \frac{t_1 \sim_H t_2 \quad s_1 \sim_H s_2}{(t_1\ s_1) \sim_H (t_2\ s_2)} \text{ [EQ-APP]}$$

$$\frac{t_1 \sim_H t_2}{(t_1 : A \Rightarrow^p B) \sim_H (t_2 : A \Rightarrow^p B)} \text{ [EQ-CAST]} \qquad \frac{t_i \sim_H t_i' \quad i \in 1..n}{(op\ \overline{t}) \sim_H (op\ \overline{t'})} \text{ [EQ-PRIM]}$$

$$\frac{t_1 \sim_H t_2}{(t_1 : A \Rrightarrow B) \sim_H (t_2 : A \Rrightarrow B)} \text{ [EQ-CLASSIFY]}$$

Proof. By induction on the derivation of $t_1 \Downarrow v_1$ and case analysis on the last rule used in the derivation.

Note that since non-interference is *termination insensitive* two different program runs could differ in their termination behavior (e.g. one could run to normal completion while the other terminates due to an attempted leaking of private data). The termination behavior permits an attacker to learn at most one bit of information about a value per execution[1]. Termination *sensitive* non-interference is a stronger guarantee but requires verifying that every loop with a confidential loop test eventually terminates, which is rather difficult (see for example [7]).

Note that blame is an additional termination channel. We could have two equivalent terms where one term evaluates to a value and the other fails by assigning blame. This does not affect termination insensitive non-interference since assigning blame is just another method of termination.

4 Gradual Information Flow Types

The runtime semantics detects bad downcasts in order to guarantee termination insensitive non-interference. However, we also want to catch security violations at compile time, where possible. To achieve this goal, we next develop a gradual type system where the labels on static types provide an upper bound on the labels of corresponding dynamic values.

[1] Though Askarov et al. [3] point out that an attacker could use intermediary output channels to leak more than a single bit, but only through a brute-force attack

The type system is given by the typing relation $\Gamma \vdash t : A$, which judges a term t to have type A under the typing environment Γ: see figure 4. The [T-PRIM] rule enforces that for each primitive operation $op\ \bar{t}$, the raw types a_i are compatible with the type signature $type(op) : a_1 \times \ldots \times a_n \to b$. It also joins the labels from each argument type ($l = \sqcup l_i$) into the result type b^l so that the resulting type will be at least as secure as the most secure argument.

The [T-APP] rule for function application $(t\ s)$ judges t to have the function type $(A \to b^k)^l$ and the argument s to have a type A' that is a subtype of A. Subtyping allows a function expecting a private input to also accept public arguments, since it will use both safely. In addition, the resulting type b^k is joined with the function's label l since the result depends on the function being used.

The [T-CAST] rule for $t : A \Rightarrow^p B$ enforces that A and B are identical apart from security labels. The operation $\lfloor . \rfloor$ defined in figure 4 strips labels from a λ_{gif} type to consider just the base types. Note that a well-typed cast may fail at runtime if the runtime security labels are not compatible.

The [T-CLASSIFY] rule for $t : A \Rrightarrow B$ checks that B has higher security labels than A. This rule uses the *positive subtyping* relation ($<:^+$) instead of the *standard subtyping* relation ($<:$) since it is not acceptable to lower the security label of a function's domain with a classification. If this rule used standard subtyping, then the classification $t : \texttt{Int}^H \to \texttt{Int}^H \Rrightarrow \texttt{Int}^L \to \texttt{Int}^H$ would be valid, which we do not want.

The full subtyping relation is given in figure 4. Two types are subtypes if they have the same base type and their labels are compatible ($l \sqsubseteq k$). If the types are function types, then the labels must be compatible and the domains must be contravariant ($A' <: A$) and the ranges covariant ($B <: B'$).

The typing system ensures that the labels in each static type is a conservative upper bound on the labels of corresponding runtime values.

We note that if a term t is well typed and we evaluate t then the resulting value v will still be well typed with the same type.[2]

Theorem 2 (Preservation).
If $\emptyset \vdash t : A$ *and* $t \Downarrow v$ *then* $\emptyset \vdash v : A$

The type system defers cast checks to the runtime system, since the safety of downcasts cannot be determined by the typing rules. For example, $v : \texttt{Int}^H \Rightarrow^p \texttt{Int}^L$ will succeed if v has a public runtime label but it will fail if the label is private. However, we can still use types to partially reason about which cast failures may occur. In particular, if two types are subtypes in a cast, then it is not possible for either positive or negative blame to occur. Furthermore, we can decompose the subtyping relation into positive and negative subtyping (see figure 4), in a manner similar to [1, 2, 27]. If the types in a cast are positive (resp. negative) subtypes then the cast cannot produce positive (resp. negative) blame.

[2] Since we are using a big-step semantics to simplify the proof of non-interference we omit the standard progress theorem, which is difficult to show in a big-step semantics.

Figure 4: λ_{gif} Typing Rules

$\boxed{\Gamma \vdash t : A}$

$$\frac{}{\Gamma \vdash c^l : type(c)^l} \text{ [T-CONST]} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \text{ [T-VAR]}$$

$$\frac{\Gamma, x\colon A \vdash t : B}{\Gamma \vdash (\lambda x\colon A.\ t)^l : (A \to B)^l} \text{ [T-ABST]} \qquad \frac{\Gamma \vdash t : A \quad A <:^+ B}{\Gamma \vdash (t\colon A \Rrightarrow B) : B} \text{ [T-CLASSIFY]}$$

$$\frac{\Gamma \vdash t : (A \to b^k)^l \quad \Gamma \vdash s : A' \quad A' <: A}{\Gamma \vdash t\ s : b^{k \sqcup l}} \text{ [T-APP]} \qquad \frac{\Gamma \vdash t : A \quad \lfloor A \rfloor = \lfloor B \rfloor}{\Gamma \vdash (t\colon A \Rightarrow^p B) : B} \text{ [T-CAST]}$$

$$\frac{\Gamma \vdash t_i : a_i^{l_i} \quad i \in 1..n \quad type(op) : a_1 \times \ldots \times a_n \to b \quad l = \sqcup l_i}{\Gamma \vdash op\ \overline{t} : b^l} \text{ [T-PRIM]}$$

$\boxed{A <: B}$

$$\frac{l \sqsubseteq k}{\imath^l <: \imath^k} \text{ [SUB-BASE]} \qquad \frac{l \sqsubseteq k \quad A' <: A \quad B <: B'}{(A \to B)^l <: (A' \to B')^k} \text{ [SUB-APP]}$$

$$\frac{l \sqsubseteq k}{\imath^l <:^+ \imath^k} \text{ [SUB-P-BASE]} \qquad \frac{l \sqsubseteq k \quad A' <:^- A \quad B <:^+ B'}{(A \to B)^l <:^+ (A' \to B')^k} \text{ [SUB-P-APP]}$$

$$\frac{k \sqsubseteq l}{\imath^l <:^- \imath^k} \text{ [SUB-N-BASE]} \qquad \frac{k \sqsubseteq l \quad A' <:^+ A \quad B <:^- B'}{(A \to B)^l <:^- (A' \to B')^k} \text{ [SUB-N-APP]}$$

$\boxed{\lfloor A \rfloor : \lambda_{gif}\ types \to \lambda_{stlc}\ types}$

$$\lfloor (A \to B)^k \rfloor = \lfloor A \rfloor \to \lfloor B \rfloor$$
$$\lfloor a^k \rfloor = a$$

Theorem 3 (Blame Theorem).

1. *If* $\emptyset \vdash t : A$ *and* $\forall (t' : B \Rightarrow^p C) \in t$, $B <: C$ *then* $t \not\Downarrow$ *blame* p *and* $t \not\Downarrow$ *blame* $\overline{p}$.
2. *If* $\emptyset \vdash t : A$ *and* $\forall (t' : B \Rightarrow^p C) \in t$, $B <:^+ C$ *then* $t \not\Downarrow$ *blame* p.
3. *If* $\emptyset \vdash t : A$ *and* $\forall (t' : B \Rightarrow^{\overline{p}} C) \in t$, $B <:^- C$ *then* $t \not\Downarrow$ *blame* $\overline{p}$.

Proof. By contradiction assuming that blame has occurred.

5 Related Work

Information flow has a long history of investigating both static and dynamic approaches to track information going back to the work of Denning [8, 9]. Sabelfeld and Myers have an extensive survey of the field [18]. Our paper provides a synthesis of prior static and dynamic techniques.

There are a number of approaches that use type systems for information flow. Volpano et al. [26] formulate the work of Denning as a type system and prove its soundness. Heintze and Riecke [13] extend a simple calculus that uses

a type system to track direct and indirect object creators and readers. Pottier and Simonet [17] present information flow type inference for a simplified ML.

Some approaches use purely dynamic techniques. Austin and Flanagan [4, 5] dynamically track information flow. Shroff et al. [19] dynamically track information flow by tracking indirect dependencies of program points. Devriese and Piessens [10] take an alternative approach called secure multi-execution that runs the program multiple times, once for each security level.

Several approaches use a hybrid of static analysis with dynamic checks during runtime to enforce information-flow guarantees. This idea is similar to our work but our contribution is to allow the programmer to choose when to use dynamic checks and when to use static typing. Chandra and Franz [6] use both static and dynamic techniques for the Java Virtual Machine and allow policies to be changed at runtime. Myers [15] defines an extension to Java called JFlow (which has become Jif [14]) using the hybrid method.

Research on integrating static and dynamic type systems also has a large body of work which we take as our starting point for extending types with security labels. Thatte [22] uses structural subtyping and the notion of *quasi-static* typing to integrate static and dynamic types. Tobin-Hochstadt and Felleisen [23] automatically infer contracts on untyped modules and formulate Typed Racket [24, 25]. Gronski et al. [12] use hybrid type checking, which integrates static type checking with dynamic contract checking in the Sage language. Siek and Taha [20] present gradual typing which uses runtime casts when types are not known at compile time. Wrigstad et al. [28] use the notion of *like types* in the Thorn language. Ahmed et al. [21, 1, 27] combine static and dynamic types with casts and *blame*; much of our formulation follows their methods and notation but with the addition of security labels and information flow.

6 Conclusion

We have presented an idealized language for *gradual security*. The language enables programmers to mark data as confidential, and the language runtime tracks confidential data through all program operations, allowing subsequent cast checks to ensure that sensitive data is not released inappropriately. In this way, termination insensitive non-interference is guaranteed in a dynamic manner.

In addition, types can be gradually refined with security labels to document interface expectations and to statically reason about the data. These labels need not be added all at once; instead, dynamic casts mediate between conventionally typed code (with no security labels) and precisely typed code (with labels).

We show how the notions of positive and negative subtyping help reason about which casts may fail at run-time, and who may be blamed for such failures.

This work represents an initial exploration in terms of an idealized language, illustrating some key ideas and correctness properties. Much work remains to scale up these techniques to realistic languages and to validate the practical utility of gradual security. In particular, we have not yet addressed assignments, which introduce some difficulties for dynamic information flow due to implicit flows, and which remains an important topic for future work.

References

1. A. Ahmed, R. Findler, J. Matthews, and P. Wadler. Blame for all. In *Proceedings for the 1st workshop on Script to Program Evolution*, pages 1–13. ACM, 2009.
2. A. Ahmed, R. Findler, J. G. Siek, and P. Wadler. Blame for all, 2011. Draft copy, to appear in POPL 2011.
3. A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS '08: Proceedings of the 13th European Symposium on Research in Computer Security*, pages 333–348, Berlin, Heidelberg, 2008. Springer-Verlag.
4. T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS '09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pages 113–124, New York, NY, USA, 2009. ACM.
5. T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 1–12. ACM, 2010.
6. D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. In *ACSAC*, pages 463–475. IEEE Computer Society, 2007.
7. B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv. Proving conditional termination. In *Computer Aided Verification*, pages 328–340. Springer, 2008.
8. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.
9. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.
10. D. Devriese and F. Piessens. Noninterference through secure multi-execution. *Security and Privacy, IEEE Symposium on*, 0:109–124, 2010.
11. R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the International Conference on Functional Programming*, pages 48–59, 2002.
12. J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Practical hybrid checking for expressive types and specifications. In *Proceedings of the Workshop on Scheme and Functional Programming*, pages 93–104, 2006.
13. N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Symposium on Principles of Programming Languages*, pages 365–377, 1998.
14. Jif homepage. http://www.cs.cornell.edu/jif/, accessed October 2010.
15. A. C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.
16. A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Symposium on Operating System Principles*, pages 129–142, 1997.
17. F. Pottier and V. Simonet. Information flow inference for ML. *Transactions on Programming Languages and Systems*, 25(1):117–158, 2003.
18. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, Jan 2003.
19. P. Shroff, S. F. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *CSF*, pages 203–217. IEEE Computer Society, 2007.
20. J. G. Siek and W. Taha. Gradual typing for functional languages. In *Proceedings of the Workshop on Scheme and Functional Programming*, 2006.

21. J. G. Siek and P. Wadler. Threesomes, with and without blame. In *POPL*, pages 365–376, 2010.
22. S. Thatte. Quasi-static typing. In *POPL 90 Proceedings of the 17th ACM SIGPLANSIGACT symposium on Principles of programming languages*, pages 367–381. ACM, 1990.
23. S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *Companion to the 21st ACM SIGPLAN symposium on Objectoriented programming systems languages and applications*, pages 964–974. ACM, 2006.
24. S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. *ACM SIGPLAN Notices*, 43(1):395–406, 2008.
25. S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 117–128. ACM, 2010.
26. D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.
27. P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Proceedings of the Workshop on Scheme and Functional Programming*, 2007.
28. T. Wrigstad, F. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 377–388. ACM, 2010.
29. S. A. Zdancewic. *Programming languages for information security*. PhD thesis, Cornell University, 2002.

Type Inference with Run-time Logs (Work in Progress)

Ravi Chugh, Ranjit Jhala, and Sorin Lerner

University of California, San Diego

Abstract. Gradual type systems offer the possibility of migrating programs in dynamically-typed languages to more statically-typed ones. There is little evidence yet that large, real-world dynamically-typed programs can be migrated with a large degree of automation. Unfortunately, since these systems typically lack principal types, fully automatic type inference is beyond reach. To combat this challenge, we propose using logs from run-time executions to assist inference. As a first step, in this paper we study how to use run-time logs to improve the efficiency of a type inference algorithm for a small language with first-order functions, records, parametric polymorphism, subtyping, and bounded quantification. Handling more expressive features in order to scale up to gradual type systems for dynamic languages is left to future work.

1 Introduction

Dynamic languages have become increasingly popular in recent years, stimulating renewed interest in the long-studied question of how to mix the guarantees of static type systems with the flexibility of dynamic languages. The *gradual typing* approach [10, 11, 2] extends a static type system with a type `dynamic` that all values inhabit. Unlike values of a `Top` type, which cannot be used to do any computation, values of type `dynamic` can be implicitly downcast to any type. The benefit of this approach is that portions of a program can be annotated with non-trivial (non-`dynamic`) types that are statically checked in standard ways, while other portions annotated with `dynamic` fall back on run-time checks. The continuous spectrum offered by this approach is appealing for migrating existing programs written in dynamic languages to more statically-typed languages.

There is little evidence yet that such programs can easily be migrated, however. The first barrier is defining a type system (even with the `dynamic` type) that can assign types to non-trivial programs in practice. Many attempts [13, 14, 6, 5] are unable to type all of the features in a full dynamic language and so fall back on manual annotation or refactoring. Even if this challenge is overcome, the problem of how to add type annotations to existing unannotated programs remains. Requiring programmers to provide type annotations can hinder adoption, so a successful approach will likely require a large degree of automation.

Unfortunately, there are barriers to static type reconstruction for statically-typed object systems, let alone gradual ones. Both partial and full type inference

for System F are undecidable [8, 15], and many object systems [3] are based on yet more expressive type theories like $F_{\leq}$, which extends F with subtyping and bounded quantification [4, 9]. The lack of principal types in these systems stand in the way of effective type inference. Constraint-based systems have been used to recover principal types despite the presence of subtyping [12], but types in these systems can become more complex than those in syntactic systems like $F_{\leq}$. Since our ideal gradual type system should types that are easy to understand, in this work we focus on systems with syntactically-limited forms of subtyping constraints. When extending these systems with `dynamic`, to create gradual object systems, the problem becomes harder still because `dynamic` annotations can be assigned in multiple incomparable ways. One approach is to insert the minimal number of casts required [7]. Another approach, however, might be to insert casts to minimize how frequently they are executed during typical executions.

To combat these challenges, we propose an approach that uses traces from run-time executions to assist type inference for languages without principal types. The idea is that although a program may be well-typed in many ways, a particular set of executions may eliminate some of the incomparable possible types. The test suites that often accompany dynamically-typed programs could provide our approach with the run-time observations it needs. As a first step towards this goal, in this paper we study a language with first-order functions and records, defined in Section 2. We present several type systems for this language, fully-static type inference algorithms, and improvements that can be made with the help of run-time information. In Sections 3 and 4, we start with a system that has parametric polymorphism and subtyping; we call it System E due to its similarity to System F besides the lack of higher-order functions. In Section 5, we extend the system with a simple form of bounded quantification. Although we consider the type inference algorithms in a fair amount of detail, we will not prove desired formal properties in this work. In Section 6, we outline the future challenges for our approach, including adding support for recursion, higher-order functions, and `dynamic`.

Recent work on type inference for Ruby [1] leverages run-time executions to assign types to a program. They wrap run-time values with type variables and then generate subtype constraints when wrapped values are used at primitive operations, field operations, and method invocations. Constraints are then solved after execution to assign static types. Our approach, which instead starts with completely static type inference and then uses run-time information to improve its efficiency, deviates from their fully-dynamic approach for two reasons. Since dynamic code generation, a major obstacle to static type inference in Ruby, is not present in our language, we would like to do as much static inference as possible. Furthermore, even though not all expressions in our language have principal types, some do. We would like to statically infer types in these situations and only rely on run-time information when necessary. If the challenges for our approach are overcome, then the combination of these two approaches might be fruitful, because most dynamic languages have both dynamic code generation and features that prevent the existence of principal types.

2 Programs

Each type system in this paper classifies programs from the following grammar. A program is a sequence of function declarations.

$$
\begin{array}{rcl}
d & ::= & \texttt{def}\ z\texttt{(}x\texttt{)\{}e\texttt{\}} \mid \texttt{def}\ z\texttt{[}\overline{X}\texttt{](}x\texttt{:}\pi\texttt{)\{}e\texttt{\}} \\
e & ::= & x \mid c \mid e_1\ \texttt{op}\ e_2 \mid \texttt{\{}\overline{f}\texttt{=}\overline{e}\texttt{\}} \mid e\texttt{.}f \mid z\texttt{(}e\texttt{)} \mid z\texttt{[}\overline{\tau}\texttt{](}e\texttt{)} \\
 & \mid & \texttt{if}_k\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2
\end{array}
$$

We use the metavariables d for function declarations, e for expressions, x and y for arguments and let-bound variables, z for function identifiers, c for base value constants, f and g for field names, B for base types, X for type variables, and τ, π, and σ for arbitrary types. Expressions are variables, base values (naturals, booleans, etc.), primitive operations, record literals, field projections, function calls, if-expressions labeled with unique integer identifiers k, and let-bindings. We use vector notation for sequences and use integer subscripts to index them. Note that function definitions and calls both have untyped and typed versions. Strictly speaking, terms of the external language contain no type annotations and leave the burden to the type inference algorithm, and terms of the internal language always provide type annotations to be checked by the type checker. To keep the presentation simple, we informally represent both forms with the same grammar, and we rely on context to disambiguate between typed and untyped programs.

3 Type Inference for System E$^-$

We start with E$^-$ ("E-Minus"), a system without a typing rule for if-expressions. The type language, subtyping relation, two expression typing rules, and a top-level function typing rule for E$^-$ are shown below.

$$\tau \quad ::= \quad B \mid X_{x.l} \mid \texttt{\{}\overline{f}\texttt{:}\overline{\tau}\texttt{\}}_\bullet \mid \texttt{\{}\overline{f}\texttt{:}\overline{\tau}\texttt{\}}_{x.l}$$

$$\frac{}{\tau \le \tau}\ \textsc{S-Refl} \qquad \frac{\forall i.\ \exists ji.\ f_i = g_{ji} \wedge \pi_{ji} \le \tau_i}{\texttt{\{}\overline{g}\texttt{:}\overline{\pi}\texttt{\}} \le \texttt{\{}\overline{f}\texttt{:}\overline{\tau}\texttt{\}}}\ \textsc{S-Rcd}$$

$$\frac{S; \Gamma \vdash e : \pi \quad \pi \le \texttt{\{}f\texttt{:}\tau\texttt{\}}}{S; \Gamma \vdash e\texttt{.}f : \tau}\ \textsc{T-Proj} \qquad \frac{S(z) = \forall \overline{X}.\ \pi \to \pi' \quad S; \Gamma \vdash e : \tau \quad |\overline{X}| = |\overline{\sigma}| \quad \tau \le \pi[\overline{\sigma}/\overline{X}]}{S; \Gamma \vdash z\texttt{[}\overline{\sigma}\texttt{](}e\texttt{)} : \pi'[\overline{\sigma}/\overline{X}]}\ \textsc{T-App}$$

$$\frac{S; x\texttt{:}\pi \vdash e : \pi'}{S \vdash \texttt{def}\ z\texttt{[}\overline{X}\texttt{](}x\texttt{:}\pi\texttt{)\{}e\texttt{\}} : \pi'}\ \textsc{T-Fun}$$

A signature S maps functions z to function types $\forall \overline{X}.\ \pi \to \pi'$ and a context Γ maps variables x to expression types τ. We write subscripts on type variables, using $X_{x.l}$ for the type of formal parameter x projected on the sequence of fields l. We also write subscripts on record types, using $\texttt{\{}\overline{f}\texttt{:}\overline{\tau}\texttt{\}}_\bullet$ to denote that the record

type corresponds to a record literal, and $\{\overline{f}:\overline{\tau}\}_{x.l}$ to denote that it corresponds to the parameter x projected on fields l. We omit subscripts on record types when they are irrelevant. Subscripts on type variables and record types are used only during inference and are ignored by the type checking rules. Notice that in the T-FUN rule, we check that the number of type actuals supplied i equals the number of type parameters required j. We expect E^- to have a standard soundness property but have not checked it.

Consider the following function and three of its infinitely many valid types.[1]

```
def wrap(x){ {orig=x;asucc=succ x.a} }
```

$$\begin{array}{lll} \sigma_1 & & \texttt{\{a:Nat\}}_{\texttt{x}} \to \texttt{\{orig:\{a:Nat\}}_{\texttt{x}}\texttt{;asucc:Nat\}}_{\bullet} \\ \sigma_2 & \forall X_{\texttt{x.b}}. & \texttt{\{a:Nat;b:}X_{\texttt{x.b}}\texttt{\}}_{\texttt{x}} \to \texttt{\{orig:\{a:Nat;b:}X_{\texttt{x.b}}\texttt{\}}_{\texttt{x}}\texttt{;asucc:Nat\}}_{\bullet} \\ \sigma_3 & & \texttt{\{a:Nat;b:Bool\}}_{\texttt{x}} \to \texttt{\{orig:\{a:Nat;b:Bool\}}_{\texttt{x}}\texttt{;asucc:Nat\}}_{\bullet} \end{array}$$

There is no best type among these; σ_2 is better than σ_3, but σ_1 and σ_2 are incomparable since σ_1 would allow `wrap` to be called in more contexts but σ_2 would allow its return value to used in more contexts. This demonstrates that E^- does not have the principal type property, so our type inference algorithm considers the calling contexts of `wrap` to determine which type to assign.

3.1 Iterative Static Inference

In this section, we outline an iterative, fully-static type inference algorithm for E^-. We develop the intuition for the algorithm by considering how to type `wrap` and the following calling context.

```
def main(){ let o = wrap({a=1;b=true}).orig in o.b }
```

The first time we process `wrap`, we gather constraints on its argument based on its uses only within the function body. Since the only requirement is that `x` has an `a` field, we assign σ_1 before moving on to `main`. Because of the function's return type and the projection on field `orig`, the variable `o` gets type $\texttt{\{a:Nat\}}_{\texttt{x}}$, so the expression `o.b` is not well-typed. The subscript `x` indicates that the record type originated from the parameter of `wrap`, so if we could require `x` to have a `b` field, then `o.b` would be well-typed. At this point, we record the *caller-induced* field constraint (as opposed to *callee-induced* constraints generated from the body of a function) that `x` has `b` and restart inference from the beginning of the program.[2] In light of this constraint, when we process `wrap` for the second time, we assign σ_2. Notice that the type of the `b` field is left unconstrained. When we process `main` for the second time, the type of `o` is $\texttt{\{a:Nat;b:Bool\}}_{\texttt{x}}$, so `o.b` is well-typed. In general, when a function type is changed from one iteration to the next, calls that were well-typed with the old type may not be with the new type. Multiple valid function types may be incomparable, so there might not be one that satisfies all of its callers. Similarly, although one type may satisfy all

[1] This example is used to motivate bounded quantification in TAPL [9].

[2] An optimized version would process only `wrap` and its (transitive) callers again.

calling contexts in a given program, it may not satisfy other well-behaved calls added to the program in the future.

Our inference algorithm for E^- generates constraint sets C with two kinds of constraints. A type variable can either be equated to a base type when used at a primitive operation or required to have a field because of a projection. Notice that we do not generate arbitrary subtyping constraints between types, only subtyping constraints to denote that a record must have a particular field. This limited form of subtyping constraints enables a simple solving algorithm (Appendix A). Caller-induced constraints will always be of the second kind.

$$C \quad ::= \quad C \cup \{X_{x.l} = B\} \;\mid\; C \cup \{X_{x.l} \leq \{\, f : X_{x.l.f} \,\}\} \;\mid\; \emptyset$$

Constraint Typing for Expressions. In this section, we define a constraint typing relation that derives a type for an expression if constraints induced by the expression are satisfiable. A term e is rewritten to e', because type parameters for function calls must be filled in. For all other kinds of expressions, the original and rewritten terms are equal. We intend the following soundness proposition, as well as a similar completeness one, to hold, though we do not prove it.

$$\text{If } S; \Gamma \vdash e \Rightarrow e' : \tau \mid C \text{ and } \theta = \mathit{Solve}(C), \text{Then } S; \theta\Gamma \vdash \theta e' \,:\, \theta\tau$$

The interesting cases are for field projection and function application. In the following, we assume that $\mathtt{x}$ is the formal parameter for the function currently being processed and y ranges over the parameters of previously processed functions. We first consider two projection rules.

$$\frac{S; \Gamma \vdash e \Rightarrow e' : \{\, \overline{g} : \overline{\tau} \,\} \mid C \qquad \exists j.\ f = g_j}{S; \Gamma \vdash e.f \;\Rightarrow e'.f : \tau_j \mid C}\ \text{CT-Proj1} \qquad \frac{S; \Gamma \vdash e \Rightarrow e' : X_{\mathtt{x}.l} \mid C \qquad C' = C \cup \{X_{\mathtt{x}.l} \leq \{\, f : X_{\mathtt{x}.l.f} \,\}\}}{S; \Gamma \vdash e.f \;\Rightarrow e'.f : X_{\mathtt{x}.l.f} \mid C'}\ \text{CT-Proj2}$$

If e is a record type with the desired field f, CT-Proj1 concludes that $e.f$ has the type of the field. If the type of e is a variable $X_{\mathtt{x}.l}$ (one of the arguments for the function that we are currently analyzing), CT-Proj2 imposes the appropriate subtype constraint on $X_{\mathtt{x}.l}$ and $X_{\mathtt{x}.l.f}$ and concludes that $e.f$ has type $X_{\mathtt{x}.l.f}$.

To support backtracking, we define a second constraint expression typing relation that indicates failure with a caller-induced constraint. CT-Proj3 triggers backtracking by producing a caller-induced constraint on the parameter y of a previous function, which immediately propagates through expressions with rules like CT-PropSucc and CT-PropApp1.

$$\frac{S; \Gamma \vdash e \Rightarrow e' : \{\, \overline{g} : \overline{\tau} \,\}_{y.l} \mid C \qquad \nexists j.\ f = g_j}{S; \Gamma \vdash e.f \nwarrow \{X_{y.l} \leq \{\, f : X_{y.l.f} \,\}\}}\ \text{CT-Proj3}$$

$$\frac{S; \Gamma \vdash e \nwarrow C}{S; \Gamma \vdash \mathtt{succ}\ e \nwarrow C}\ \text{CT-PropSucc} \qquad \frac{S; \Gamma \vdash e \nwarrow C}{S; \Gamma \vdash z(e) \nwarrow C}\ \text{CT-PropApp1}$$

The last case that we discuss here is for function calls. The *Call* judgment (defined in Appendix A) takes a formal type π, an actual type τ, and returns a substitution θ that instantiates the free variables of π together with additional

constraints C' required for τ to be a subtype of π (after instantiation). The additional constraints may fail because of missing fields, so there is also a failure version of $Call$, which is propagated by CT-PropApp2.

$$\frac{\begin{array}{c} S(z) = \forall \overline{X}.\ \pi \to \pi' \\ S; \Gamma \vdash e \Rightarrow e' : \tau \mid C \\ Call(\pi, \tau) \vdash (C', \theta) \qquad \overline{\sigma} = \theta \overline{X} \end{array}}{S; \Gamma \vdash z(e) \Rightarrow z[\overline{\sigma}](e') : \theta\pi' \mid C \cup C'}\ \text{CT-App} \qquad \frac{\begin{array}{c} S(z) = \forall \overline{X}.\ \pi \to \pi' \\ S; \Gamma \vdash e \Rightarrow e' : \tau \mid C \\ Call(\pi, \tau) \nwarrow C' \end{array}}{S; \Gamma \vdash z(e) \nwarrow C'}\ \text{CT-PropApp2}$$

Constraint Typing for Functions. We define a constraint typing relation for function definitions that processes one at a time by appealing to the constraint expression typing relation for its body. The judgment refers to a set of caller-induced constraints C_0 from previous iterations, as well as the list of previously processed untyped functions $\overline{d}$, which must be processed again if there are new caller-induced constraints.

$$\frac{\begin{array}{c} S; x : X_x \vdash e \Rightarrow e' : \tau' \mid C \qquad \theta = Solve(C_0 \cup C) \\ \pi = \theta X_x \qquad \pi' = \theta\tau' \qquad \overline{X} = FTV(\pi) \qquad e'' = \theta e' \end{array}}{C_0; \overline{d}; S \vdash \texttt{def}\ z(x)\{e\} \Rightarrow \texttt{def}\ z[\overline{X}](x:\pi)\{e''\} : \pi'}\ \text{CT-Fun}$$

$$\frac{\begin{array}{c} S; x : X_x \vdash e \nwarrow C \qquad C_1 = C_0 \cup C \\ C_1; -; - \vdash \overline{d} \Rightarrow \overline{d'} : S' \qquad C_1; \overline{d}; S' \vdash d_1 \Rightarrow d_2 : \pi' \end{array}}{C_0; \overline{d}; S' \vdash d_1 \Rightarrow d_2 : \pi'}\ \text{CT-Iter}$$

The CT-Fun rule applies when the body of function z has a successful constraint typing derivation. In addition to the constraints C that it produces, the caller-induced constraints C_0 must also be satisfied. The $Solve$ function (defined in Appendix A) attempts to solve C and C_0. If a solution exists, it is used to ascribe a function type to the declaration. The CT-Iter rule handles the case where constraint typing on the function body fails with new caller-induced constraints C. These are combined with the existing caller-induced constraints C_0, and all previous functions $\overline{d}$ are processed again (the third premise). We omit the definition of this relation. If all previous functions can be typed with the updated caller-induced constraints, then typing of the current function resumes (the fourth premise). Notice that the signature S' may differ from S, since the types of previously processed functions may have changed.

Like with constraint typing for expressions, we intend that the following soundness property holds, as well as a similar completeness one.

If $C_0; \overline{d}; S \vdash \texttt{def}\ z(x)\{e\} \Rightarrow \texttt{def}\ z[\overline{X}](x:\pi)\{e''\} : \pi'$,
Then $S \vdash \texttt{def}\ z[\overline{X}](x:\pi)\{e''\} : \pi'$.

We expect that this property will be considerably harder to prove than the one for expressions. One important lemma will be that if the third premise of CT-Iter is satisfied, then resumption of the current function (the fourth premise), will get strictly farther than did the previous attempt (the first premise). Because expressions are finite, this will enable proving termination.

3.2 Inference with Run-time Logs

Now that we have sketched out how static iterative type inference for E^- works, we turn to the question of how run-time logs might help. The need for iteration in our constraint typing rules comes from the fact that when processing a function, we do not know how its return value will be used in calling contexts. We will define an evaluation semantics for E^- to record precisely this information, so that we can define a modified inference algorithm that does not need to backtrack.[3]

In our evaluation semantics, we wrap run-time values with sets of type variables T. We use v to range over raw values, tv over tagged values, and L over run-time logs. Notice that every run-time log is a valid constraint set.

$$\begin{array}{rcl} v & ::= & c \mid \texttt{\{}\,\overline{f}\texttt{=}\,\overline{tv}\,\texttt{\}} \\ tv & ::= & (v, T) \\ L & ::= & L \cup \{X_{x.l} \leq \texttt{\{}\,f : X_{x.l.f}\,\texttt{\}}\} \mid \emptyset \end{array}$$

Our big-step evaluation relation evaluates an expression e in an environment E, which maps variables to tagged values, and produces a tagged value along with a log. The cases for application and projection are the interesting ones.

$$\frac{\begin{array}{c} S(z) = \lambda x.e' \\ (E, e) \Downarrow ((v, T), L) \\ tv_1 = (v, T \cup \{X_x\}) \\ (E[x \mapsto tv_1], e') \Downarrow (tv_2, L') \end{array}}{(E, z\texttt{(}e\texttt{)}) \Downarrow (tv_2, L \cup L')}\ \text{E-App} \qquad \frac{\begin{array}{c} (E, e) \Downarrow ((\texttt{\{}\,\overline{g}\texttt{=}\,\overline{tv}\,\texttt{\}}, T), L) \\ \exists j.\ f = g_j \quad (v_j, T_j) = tv_j \\ L' = \{\, X_{y.l} \leq \texttt{\{}\,f : X_{y.l.f}\,\texttt{\}} \mid X_{y.l} \in T \,\} \\ T' = \{\, X_{y.l.f} \mid X_{y.l} \in T \,\} \end{array}}{(E, e\,.\,f) \Downarrow ((v_j, T_j \cup T'), L \cup L')}\ \text{E-Proj}$$

When a tagged value is passed as an argument to a function with parameter x, E-App adds the variable X_x to its tag set before evaluating the function body. When a tagged value is projected on field f, E-Proj records constraints that each of its type variables have the f field. There are no rules that strip tags from values, so once E-App adds a tag to a value, it will carry around that tag for the rest of its execution. Consequently, if a function returns one of its arguments, the run-time log will capture all subsequent caller-induced constraints.

We can now improve our constraint typing for functions. In place of CT-Fun and CT-Iter, we define a new relation that refers to a run-time log and does not need to maintain previously processed functions. CTL-Fun refers to a log L instead of caller-induced constraints C_0 like CT-Fun does.

$$\frac{\begin{array}{c} S; x : X_x \vdash e \Rightarrow e' : \tau' \mid C \qquad \theta = \mathit{Solve}(L \cup C) \\ \pi = \theta X_x \qquad \pi' = \theta\tau' \qquad \overline{X} = \mathit{FTV}(\pi) \qquad e'' = \theta e' \end{array}}{L; S \vdash \texttt{def}\ \ z\texttt{(}x\texttt{)\{}\,e\,\texttt{\}} \Rightarrow \texttt{def}\ \ z\texttt{[}\overline{X}\texttt{](}x\texttt{:}\pi\texttt{)\{}\,e''\,\texttt{\}} : \pi'}\ \text{CTL-Fun}$$

If the execution that produced L exercised every expression in the program, then L will contain all the caller-induced constraints that the iterative static algorithm generates.[4] Thus, by using the run-time log, the modified inference algorithm

[3] We could instead use normal evaluation and instrument programs to emit logs.

[4] To check, we can add integer identifiers to expressions and log them during execution.

(which runs after execution) does not need to backtrack. We intend this modified algorithm to satisfy analagous soundness and completeness properties to the iterative version.

The fully-static and modified inference algorithms for E^-, which we considered fairly closely, form the basis of the algorithms for the remaining systems, which we will discuss in less detail.

4 Type Inference for System E

System E extends E^- with a typing rule for if-expressions. We extend the type language with three forms: a maximal type `Top`, a type variable that stands for the result of if-expression k and then projected on fields l, and a new kind of record type to indicate that its provenance is an if-expression.

$$\tau \quad ::= \quad \cdots \;\mid\; \texttt{Top} \;\mid\; X_{k.l} \;\mid\; \{\overline{f}:\overline{\tau}\}_{k.l}$$

The new subscripted type variables and record types are used only by the inference algorithm; they are "expanded away" during inference.

Consider the following function and several valid incomparable types. We use tuple notation freely since we can encode them as records with fields `1` and `2`. Each return type has a subscript to indicate its origin from the if-expression.

```
def choose(y,z){ if1 y.n > 0 then y else z }
```

$$\begin{array}{lll}
\sigma_4 & & (\{\texttt{n}:\texttt{Nat}\}_{\texttt{y}} * \{\}_{\texttt{z}}) \to \{\}_1 \\
\sigma_5 & & (\{\texttt{n}:\texttt{Nat}\}_{\texttt{y}} * \{\texttt{n}:\texttt{Top}\}_{\texttt{z}}) \to \{\texttt{n}:\texttt{Top}\}_1 \\
\sigma_6 & & (\{\texttt{n}:\texttt{Nat}\}_{\texttt{y}} * \{\texttt{n}:\texttt{Nat}\}_{\texttt{z}}) \to \{\texttt{n}:\texttt{Nat}\}_1 \\
\sigma_7 & & (\{\texttt{n}:\texttt{Nat};\texttt{b}:\texttt{Bool}\}_{\texttt{y}} * \{\texttt{n}:\texttt{Nat}\}_{\texttt{z}}) \to \{\texttt{n}:\texttt{Nat}\}_1 \\
\sigma_8 & & (\{\texttt{n}:\texttt{Nat};\texttt{b}:\texttt{Bool}\}_{\texttt{y}} * \{\texttt{n}:\texttt{Nat};\texttt{b}:\texttt{Bool}\}_{\texttt{z}}) \to \{\texttt{n}:\texttt{Nat};\texttt{b}:\texttt{Bool}\}_1 \\
\sigma_9 & & (\{\texttt{n}:\texttt{Nat};\texttt{b}:\texttt{Top}\}_{\texttt{y}} * \{\texttt{n}:\texttt{Nat};\texttt{b}:\texttt{Top}\}_{\texttt{z}}) \to \{\texttt{n}:\texttt{Nat};\texttt{b}:\texttt{Top}\}_1 \\
\sigma_{10} & \forall X_{\texttt{y.b}}. & (\{\texttt{n}:\texttt{Nat};\texttt{b}:X_{\texttt{y.b}}\}_{\texttt{y}} * \{\texttt{n}:\texttt{Nat};\texttt{b}:X_{\texttt{y.b}}\}_{\texttt{z}}) \to \{\texttt{n}:\texttt{Nat};\texttt{b}:X_{\texttt{y.b}}\}_1
\end{array}$$

Although some of these types are better than others, there is no best one. For example, given the following calling contexts, σ_4 can type `main1`, σ_5 can type `main2`, and σ_6 can type `main3`, but these three types are incomparable.

```
def main1(){ let t1 = choose({n=1},{}) in t1 }
def main2(){ let t2 = choose({n=1},{n=true}) in t2.n }
def main3(){ let t3 = choose({n=1},{n=2}) in succ t3.n }
```

As before, our inference algorithm for System E iteratively adds constraints to arguments based on calling contexts. Consider how we arrive at σ_6 for `main3` in three iterations. In the first iteration, we assign σ_4 based on the constraints on `y` within the body of `choose`. When we get to the projection `t2.n`, we restart since `z` must have the `n` field for this projection to succeed. Thus, in the second iteration we assign σ_5. When we get to the `succ` operation, we require another restart since `y` and `z` must have the same type for the `n` field for this primitive operation to succeed. Thus, in the third iteration we assign σ_6.

Our constraint expression typing relation – in particular, the CT-If rule – now produces a "join tree" J that maps if-expression identifiers k to the types of their branches. We also generate two new kinds of constraints, both of which can be caller-induced.[5]

$$\frac{\begin{array}{c} S; \Gamma \vdash e_1 \Rightarrow e_1' \ : \ \texttt{Bool} \,;\, J_1 \mid C_1 \\ S; \Gamma \vdash e_2 \Rightarrow e_2' \ : \ \tau_2 \,;\, J_2 \mid C_2 \qquad S; \Gamma \vdash e_3 \Rightarrow e_3' \ : \ \tau_3 \,;\, J_3 \mid C_3 \\ J = J_1 \circ J_2 \circ J_3 \circ [k \mapsto (\tau_2, \tau_3)] \qquad C = C_1 \cup C_2 \cup C_3 \end{array}}{S; \Gamma \vdash \texttt{if}_k \ e_1 \ \texttt{then} \ e_2 \ \texttt{else} \ e_3 \Rightarrow \texttt{if}_k \ e_1' \ \texttt{then} \ e_2' \ \texttt{else} \ e_3' \ : \ X_k \,;\, J \mid C} \ \text{CT-If}$$

$$C \quad ::= \quad \cdots \ \mid \ C \cup \{X_{k.l} \leq \texttt{\{}\, f : X_{k.l.f} \,\texttt{\}}\} \ \mid \ C \cup \{X_{k.l} \ \texttt{primop}\}$$

To solve a constraint of the form $X_{k.l} \leq \texttt{\{}\, f : X_{k.l.f} \,\texttt{\}}$, the inference algorithm uses the join tree to expand $X_{k.l}$ to its sources. For each source of the form $X_{y.l'}$, it adds the constraint $X_{y.l'.l} \leq \texttt{\{}\, f : X_{y.l'.l.f} \,\texttt{\}}$, which is a constraint on ordinary type variables. To solve a constraint of the form $X_{k.l}$ `primop`, the algorithm expands $X_{k.l}$ to its sources and equates them. The definition of *Expand* can be found in Appendix B.

Like we did for E^-, we can instrument evaluation so that the inference algorithm never needs to backtrack.

$$L \quad ::= \quad \cdots \ \mid \ L \cup \{X_{k.l} \leq \texttt{\{}\, f : X_{k.l.f} \,\texttt{\}}\} \ \mid \ L \cup \{X_{k.l} \ \texttt{primop}\}$$

$$\frac{(E, e_1) \Downarrow ((\texttt{true}, T_1), L_1) \qquad (E, e_2) \Downarrow ((v, T_2), L_2)}{(E, \texttt{if}_k \ e_1 \ \texttt{then} \ e_2 \ \texttt{else} \ e_3) \Downarrow ((v, T_2 \cup \{X_k\}), L_1 \cup L_2)} \ \text{E-IfTrue}$$

After E-IfTrue or E-IfFalse adds the X_k tag to the return value of the if-expression, the modified E-Proj rule (not shown), which treats $X_{k.l}$ variables like it does $X_{x.l}$ variables, gathers the caller-induced constraints that would require iteration in the fully-static algorithm.

5 Type Inference for System $\mathrm{E}_{\leq}$

We now add support for a simple form of bounded quantification. Each type variable in a function definition is declared with a single upper bound, which must be a base type, `Top`, or a record of bounds. In particular, a bound cannot be another type variable. This restriction allows constraint solving for $\mathrm{E}_{\leq}$ to remain simple.

$$\begin{array}{rcl} d & ::= & \cdots \ \mid \ \texttt{def} \ z \texttt{[}\overline{X} \leq \overline{\delta}\texttt{]}\texttt{(}x : \pi\texttt{)\{}\, e \,\texttt{\}} \\ \delta & ::= & B \ \mid \ \texttt{Top} \ \mid \ \texttt{\{}\, \overline{f} : \overline{\delta} \,\texttt{\}} \end{array}$$

[5] So that we can discuss the algorithm at a high-level, we omit a subtle detail from the discussion: the `Top` might also carry a subscript. Constraints of the form $X_{k.l}$ `primop` can only be directly generated within the function containing the if-expression, since type variables $X_{k.l}$ are eliminated when computing the function type. We use $\texttt{Top}_{k.l}$ to enable generating $X_{k.l}$ `primop` from a calling context.

We use U to range over lists of bounded type variables $\overline{X \leq \delta}$. Function types become $\forall U.\ \pi \to \pi'$. The subtyping relation is defined to read subtyping assumptions from U, transitively if necessary. The typing judgment for expressions includes the bounds U of the particular function being typed. Before checking the type of the value argument, the function application rule must check that the type parameters supplied satisfy the bounds specified by the function's type.

$$\frac{X_{x.l} \leq \delta \in U}{U \vdash X_{x.l} \leq \delta}\ \text{S-BQ-Bound} \qquad \frac{U \vdash \delta_1 \leq \pi \quad U \vdash \pi \leq \delta_2}{U \vdash \delta_1 \leq \delta_2}\ \text{S-BQ-Trans}$$

$$\frac{\begin{array}{c} S(z) = \forall \overline{X \leq \delta}.\ \pi \to \pi' \quad n = |\overline{X}| = |\overline{\sigma}| \\ \forall i \in 1..n \quad \theta_i = [X_1 \mapsto \sigma_1] \circ \cdots \circ [X_{i-1} \mapsto \sigma_{i-1}] \quad U \vdash \sigma_i \leq \theta_i \delta_i \\ S; U; \Gamma \vdash e : \tau \quad U \vdash \tau \leq \theta_n \pi \quad \tau' = \theta_n \pi' \end{array}}{S; U; \Gamma \vdash z\,\texttt{[}\overline{\sigma}\texttt{]}\,\texttt{(}e\texttt{)} : \tau'}\ \text{T-BQ-App}$$

We now revisit the `wrap` and `choose` examples from previous sections. In $\mathrm{E}_{\leq}$, we can assign the following type to `wrap`.

$$\sigma_{11} \quad \forall X_{\mathtt{x}} \leq \texttt{\{a:Nat\}}.\ X_{\mathtt{x}} \to \texttt{\{orig:}X_{\mathtt{x}}\texttt{;asucc:Nat\}}_{\bullet}$$

This type is more general than σ_1, σ_2, and σ_3, the types that we saw in Section 3. Indeed, this is the most general type that can be assigned to `wrap`. In $\mathrm{E}_{\leq}$, we can assign the following type to `choose`.

$$\sigma_{12} \quad \forall X_{\mathtt{y}} \leq \texttt{\{n:Nat\}}.\ \texttt{(}X_{\mathtt{y}}\texttt{*}X_{\mathtt{y}}\texttt{)} \to X_{\mathtt{y}}$$

This type is more general than types σ_6 through σ_{10} that we saw in Section 4. However, σ_{12} is still incomparable with σ_4 and σ_5.

The previous example demonstrates that not every program has a principal type in System $\mathrm{E}_{\leq}$.[6] The source of ambiguity is whether or not to "share" a type variable for two parameters returned by an if-expression. To help demonstrate, consider the following versions of σ_4 and σ_5 equivalent to the original ones.

$$\begin{array}{ll} \sigma_4 & \forall X_{\mathtt{y}} \leq \texttt{\{n:Nat\}}.\ \forall X_{\mathtt{z}} \leq \texttt{\{\}}.\ \texttt{(}X_{\mathtt{y}}\texttt{*}X_{\mathtt{z}}\texttt{)} \to \texttt{\{\}}_1 \\ \sigma_5 & \forall X_{\mathtt{y}} \leq \texttt{\{n:Nat\}}.\ \forall X_{\mathtt{z.n}} \leq \texttt{Top}.\ \forall X_{\mathtt{z}} \leq \texttt{\{n:}X_{\mathtt{z.n}}\texttt{\}}.\ \texttt{(}X_{\mathtt{y}}\texttt{*}X_{\mathtt{z}}\texttt{)} \to \texttt{\{n:Top\}}_1 \end{array}$$

Notice that in σ_4 and σ_5 the constraints on `y` and `z` are kept separate with different type variables. In σ_{12}, on the other hand, the constraints are combined into a single type variable (arbitrarily named $X_{\mathtt{y}}$, though $X_{\mathtt{z}}$ would work just as well). The first factor to consider is whether the separate bounds for two type variables are compatible. Consider the following example.

```
def separateVars(v,w){ if2 iszero v.f && not w.f then v else w }
```

Since `v.f` and `w.f` have different base types, $X_{\mathtt{v}}$ and $X_{\mathtt{w}}$ must be kept separate; there is no single bound that can type both `v` and `w` correctly.

[6] In a system more powerful than $\mathrm{E}_{\leq}$, `choose` might have the principal type $\forall X_{\mathtt{y}}, X_{\mathtt{z}}.\ \{X_{\mathtt{y}} \leq \texttt{\{n:Nat\}}, X_{\mathtt{y}} \leq X_{\mathtt{z}}\} \Rightarrow \texttt{(}X_{\mathtt{y}}\texttt{*}X_{\mathtt{z}}\texttt{)} \to X_{\mathtt{z}}$.

If the constraints on two type variables for a branch are compatible, we start by using a shared variable. In general, sharing a type variable may impose more constraints on a parameter than if it was assigned its own type variable. Some call sites may not provide actuals that are well-typed with these additional requirements. For example, σ_{12} imposes more requirements on `z` than does σ_4. In the presence of a calling context like `choose({n=1},{})`, which is not well-typed when one type variable is shared for `y` and `z`, we trigger a restart and keep $X_{\mathtt{y}}$ and $X_{\mathtt{z}}$ separate in subsequent iterations. Whenever we are forced to keep two type variables separate – because their constraints are incompatible or because there is some incompatible call site – we must track fields for each type variable just we do for System E.

We would like to take advantage of run-time logs to eliminate backtracking. However, it is not clear that the new source of backtracking can be completely avoided, because observing the fields of actuals at run-time does not give an accurate description of their corresponding static types. Consider the following example.

```
def projFG(p,q){ if3 p.f > p.g then p else q }
def rememberF(r){ if4 true then r else {f = 1} }
def main(){ let o = {f=1;g=2} in projFG(o,rememberF(o)) }
```

At run-time, both values passed to `projFG` have fields `f` and `g`, so we might be tempted to conclude that this call site would be well-typed if `p` and `q` share the same type variable $X_{\mathtt{p}}$. Statically, however, the second actual has only `f` because it passes through `rememberF`. Thus, the run-time information about record actuals is not sufficent to deduce that $X_{\mathtt{p}}$ and $X_{\mathtt{q}}$ must be kept separate. On the other hand, we can be sure if two type variables must be kept separate, since if some run-time actual does not have a required field, then its static type will certainly not. For example, the call site `projFG(o,{})` would rule out the possibility of sharing the same type variable for `p` and `q`.

Thus, our modified type inference can rule out situations in which sharing bounded type variables is certainly not possible, so bounds must be kept separate. We can avoid backtracking in this case, since the run-time log will have all fields that might be used, as was the case for System E. If the log does not rule out sharing, the inference algorithm attempts to share the same type variable and proceeds. If a call site is ill-typed, then a restart is necessary so that separate type variables are used. At this point, the log contains all additional fields that will be needed, so further backtracking is avoided. Thus, using run-time logs improves the fully-static, iterative approach but does not completely eliminate the need for iteration.

6 Conclusion

We have sketched out static type inference algorithms for several versions of System E that lack principal types and then improved them with information from run-time executions. Clearly, the formal development of the desired soundness

and completeness properties must be addressed in future work. Furthermore, additional features like recursion, recursive types, nominal object types, and most importantly, higher-order functions, need to be studied for their amenability to inference with run-time logs. In particular, it will be important to determine whether partial or full type inference for System F becomes any easier with the presence of run-time logs. If so, then we may be able to apply these techniques to inferring types for realistic dynamic languages, since many object systems require System F-based type theories. If not, then more heuristic-based approaches will likely be required.

References

1. Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic inference of static types for ruby. In *POPL*, 2011.
2. Gavin Bierman, Erik Meijer, and Mads Torgersen. Adding dynamic types to c#. In *ECOOP*, 2010.
3. Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 1999.
4. Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 1985.
5. Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. In *OOPSLA*, 2009.
6. Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for ruby. In *SAC*, 2009.
7. Fritz Henglein and Jakob Rehof. Safe polymorphic type inference for a dynamically typed language: Translating scheme to ml. In *Functional Programming Languages and Computer Architecture*, 1995.
8. Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *POPL*, 1988.
9. Benjamin C. Pierce. *Types and Programming Languages*. 2002.
10. Jeremy Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006.
11. Jeremy Siek and Walid Taha. Gradual typing for objects. In *ECOOP*, 2007.
12. Martin Sulzmann, Martin Odersky, and Martin Wehr. Type inference with constrained types. In *FOOL*, 1997.
13. Peter Thiemann. Towards a type system for analyzing javascript programs. In *ESOP*, 2005.
14. Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *POPL*, 2008.
15. Joe Wells. Typability and type checking in the second-order lambda calculus are equivalent and undecidable. In *LICS*, 1994.

A Some Additional Definitions for System E$^-$

$$\frac{}{Call(X_{y.l}, \tau) \vdash (\emptyset, [X_{y.l} \mapsto \tau])}$$

$$\frac{}{Call(B, B) \vdash (\emptyset, [])} \qquad \frac{}{Call(B, X_{y.l}) \vdash (\{X_{y.l} = B\}, [])}$$

$$\frac{\begin{array}{c}\forall i.\ Call(\tau_i, X_{y.l.f_i}) \vdash (C_i, \theta_i) \\ C = \cup_i\ (\{X_{y.l} \leq \{\, f_i : X_{y.l.f_i}\,\}\} \cup C_i) \\ \theta = \theta_1 \circ \cdots \circ \theta_n\end{array}}{Call(\{\,\overline{f} : \overline{\tau}\,\}, X_{y.l}) \vdash (C, \theta)} \qquad \frac{\begin{array}{c}\forall i.\ \exists ji.\ f_i = g_{ji} \\ Call(\tau_i, \pi_{ji}) \vdash (C_i, \theta_i) \\ C = \cup_i\ C_i \qquad \theta = \theta_1 \circ \cdots \circ \theta_n\end{array}}{Call(\{\,\overline{f_i} : \overline{\tau}\,\}, \{\,\overline{g} : \overline{\pi}\,\}) \vdash (C, \theta)}$$

$$\frac{\begin{array}{c}\forall i.\ \exists ji.\ f_i = g_{ji} \\ \exists i'.\ Call(\tau_{i'}, \pi_{ji'}) \nwarrow C\end{array}}{Call(\{\,\overline{f} : \overline{\tau}\,\}, \{\,\overline{g} : \overline{\pi}\,\}) \nwarrow C} \qquad \frac{\begin{array}{c}\exists i'.\ \forall j.\ f_{i'} \neq g_j \\ C = \{X_{y.l} \leq \{\, f_{i'} : X_{y.l.f_{i'}}\,\}\}\end{array}}{Call(\{\,\overline{f} : \overline{\tau}\,\}, \{\,\overline{g} : \overline{\pi}\,\}_{y.l}) \nwarrow C}$$

$Consistent(C) = true$ iff 1. If $X_{y.l} = B \in C$ and $X_{y.l} = B' \in C$, then $B = B'$
2. If $X_{y.l} = B \in C$ and $X_{y'.l'} \leq \tau \in C$, then $(y, l) \neq (y', l')$

$$TypOf(X_{y.l}, C) = \begin{cases} B & \text{if } X_{y.l} = B \in C \\ \{\, f_k : \theta_k X_{y.l.f_k}\,\}_{y.l} & \text{otherwise, where} \\ & \quad X_{y.l} \leq \{\, f_k : X_{y.l.f_k}\,\} \in C \\ & \quad \theta_k = TypOf(X_{y.l.f_k}, C) \end{cases}$$

$$Solve(C) = \begin{cases} [X_{y.l} \mapsto \tau_{y.l}] & \text{if } Consistent(C) = true \\ & \quad \text{where } \tau_{y.l} = TypOf(X_{y.l}, C) \\ & \quad \text{for each } X_{y.l} \in C \\ fail & \text{if } Consistent(C) = false \end{cases}$$

B Some Additional Definitions for System E

$$\begin{aligned} Path(\tau, []) &= \tau \\ Path(B, f :: l) &= fail \\ Path(\{\,\overline{g} : \overline{\tau}\,\}, f :: l) &= \begin{cases} Path(\tau_j, l) & \text{if } \exists j.\ f = g_j \\ fail & \text{ow} \end{cases} \\ Path(X_{x.l'}, f :: l) &= X_{x.l'.f.l} \end{aligned}$$

$$\begin{aligned} Expand(J, B) &= \{B\} \\ Expand(J, X_{x.l}) &= \{X_{x.l}\} \\ Expand(J, \{\,\overline{f} : \overline{\tau}\,\}) &= \{\{\,\overline{f} : \overline{\tau}\,\}\} \\ Expand(J, X_{k.l}) &= \{\ Path(\tau, l) \mid \tau \in Expand(J, \tau_2) \cup Expand(J, \tau_3)\ \} \\ &\qquad \text{where } J(k) = (\tau_2, \tau_3) \end{aligned}$$

The Ciao Approach to the Dynamic vs. Static Language Dilemma

(Position/System/Demo Paper[1])

M. V. Hermenegildo[1,2] F. Bueno[1] M. Carro[1] P. López-García[2,4]
E. Mera[3] J. F. Morales[2] G. Puebla[1]

[1]Universidad Politécnica de Madrid (UPM)
{bueno,mcarro,german,herme}@fi.upm.es
[2]Madrid Institute of Advanced Studies
in SW Development Technology (IMDEA Software Institute)
{manuel.hermenegildo,pedro.lopez,jose.morales}@imdea.org
[3]Universidad Complutense de Madrid (UCM)
edison@fdi.ucm.es
[4]Scientific Research Council (CSIC)

Dynamic vs. Static Languages

The environment in which much software needs to be developed nowadays (decoupled software development, use of components and services, increased interoperability constraints, need for dynamic update or self-reconfiguration, mash-up development, etc.) is posing requirements which align with the classical arguments for dynamic languages and which in fact go beyond them. Examples of often required dynamic features include making it possible to (partially) test and verify applications which are partially developed and which will never be "complete" or "final," or which evolve over time in an asynchronous, decentralized fashion (e.g., software service-based systems). These requirements, coupled with the intrinsic agility in development of dynamic programming languages such as Python, Ruby, Lua, JavaScript, Perl, PHP, etc. (with Scheme or Prolog also in this class) have made such languages a very attractive option for a number of purposes that go well beyond simple scripting. Parts written in these languages often become essential components (or even the whole implementation) of full, mainstream applications.

At the same time, detecting errors at compile-time and inferring many properties required in order to optimize programs are still important issues in real-world applications. Thus, strong arguments are still also made in favor of static languages. For example, many modern logic and functional languages (such as, e.g., Mercury [24] or Haskell [12]) impose strong type-related requirements such as that all types (and, when relevant, modes) have to be defined explicitly or

[1] In addition to the other references, this recent tutorial overview of Ciao [11] covering more fully the points made in this position paper can be downloaded from: http://clip.dia.fi.upm.es/papers/hermenegildo10:ciao-design-tplp-tr.pdf

that all procedures have to be "well-typed" and "well-moded." One argument supporting this approach is that it clarifies interfaces and meanings and facilitates "programming in the large" by making large programs more maintainable and better documented. Also, the compiler can use the static nature of the language to generate more specific code, which can be better in several ways (e.g., performance-wise).

The Ciao Approach

In the design of Ciao [7,6,2,10,11] we certainly had the latter arguments in mind, but we also wanted Ciao to be useful (as the "scripting" languages) for highly dynamic scenarios such as those listed above, for "programming in the small," for prototyping, for developing simple scripts, or simply for experimenting with the solution to a problem. We felt that compulsory type and mode declarations, and other related restrictions, can sometimes get in the way in these contexts. Ciao aims at combining the flexibility of dynamic/scripting languages with the guarantees of static languages, to bridge programming in the small and programming in the large, while performing efficiently on platforms ranging from small embedded processors to powerful multicore architectures.

Important components of the solution we came up with are the rich Ciao assertion language and the Ciao methodology for dealing with such assertions [3,8,22], which implies *making a best effort to infer and check properties statically, even highly complex ones, by using powerful and rigorous static analysis tools based on safe approximations, while accepting that complete verification may not always be possible (at least in a fully automated way) and run-time checks may sometimes be needed*. This approach opens up the possibility of dealing in a uniform way with a wide variety of properties besides traditional types (e.g., rich modes, determinacy, non-failure, shapes, sharing/aliasing, term linearity, time, memory, general resources,...), while at the same time allowing all assertions to be *optional*.

The Ciao assertion language provides a homogeneous framework which allows, among other things, static and dynamic verification (including unit testing [17]) to work cooperatively in a unified way. It is also instrumental for auto-documentation. The Ciao Preprocessor (CiaoPP) [3,8,21,9]) is a compile-time tool, based on abstract interpretation and other related techniques, which is capable of statically finding non-trivial bugs, verifying that the program complies with specifications (written in the assertion language), introducing run-time checks for (parts of) assertions that cannot be verified statically, and performing many types of program optimizations (including automatic parallelization). Such optimizations produce code that is highly competitive not only with other dynamic (or "scripting") languages but even that of static languages, when the optimizing compiler is used, all while retaining the interactive development environment of a dynamic language. This static/dynamic compilation architecture supports modularity and separate compilation throughout.

In the Ciao approach many properties used in assertions, including for example types, are written directly (or with convenient syntactic sugar) in the source

language, so that they can be *run* and experimented with. I.e., one can test interactively if a certain data structure belongs to a type, has a particular size, or does not contain aliased pointers by just passing the data structure to the definition of the corresponding property and executing it. Furthermore, properties can often be used to enumerate (produce examples) of data which meet the property, such as, e.g., generating concrete examples of a type. This is all instrumental in the implementation of run-time checks and testing. The underlying logic engine and meta-programming capabilities of Ciao are fundamental in these tasks.

As mentioned above, the assertion language and preprocessor design also allows a smooth integration with unit testing. Unit tests are expressed as assertions. Then, as with other assertions, the (parts of) unit tests that can be verified at compile-time are eliminated, and sometimes it not not necessary whole sets of tests.

We argue that the solutions that were adopted in the Ciao design allow programming both in the small and in the large, combining effectively the advantages of the strongly typed and untyped language approaches. In contrast, systems which focus exclusively on automatic compile-time checking are often rather strict about the properties which the user can write. This is understandable because otherwise the underlying static analyses are of little use for proving the assertions.

Some Related Work

The Ciao model is related to the *soft typing* approach [4]. However, compile-time inference and checking in the Ciao model is not restricted to types (nor requires properties to be decidable), and it draws many new synergies from its novel combination of assertion language, properties, certification, run-time checking, testing, etc. The practical relevance of the combination of static and dynamic features is in fact illustrated by the many other languages and frameworks which have been proposed lately aiming at bringing together ideas of both worlds. This includes the very interesting recent work in gradual typing for Scheme [25] (and the related PLT-Scheme/Racket language), the recent uses of "contracts" in verification [16,19], and the pragmatic viewpoint of [14], but applied to programming languages rather than specification languages. The fifth edition of ECMAScript [5], on which the JavaScript and ActionScript languages are based, includes optional (soft-)type declarations to allow the compiler to generate more efficient code and detect more errors. The Tamarin project [18] intends to use this additional information to generate faster code. For Python, the PyPy project [23] has designed a language, RPython [1], that imposes constraints on the programs to ensure that they can be statically typed. RPython is moving forward as a general purpose language. This line of thought has also brought the development of safe versions of traditional languages, such as, e.g., CCured [20] or Cyclone [13] for C, as well as of systems that offer functionality similar to those of the Ciao assertion preprocessor, such as Deputy (`http://deputy.cs.berkeley.edu/`) or Spec# [15]. In summary, we argue that Ciao pioneered what are becoming rela-

tively widely accepted approaches to marrying the static and dynamic language worlds.

Language Extensibility in Ciao

While not as directly related to the dynamic vs. static dilemma, another important characteristic of Ciao is that it is built up from a kernel that includes significant extensibility capabilities, i.e., it includes an easily programmable and modular way of defining new syntax and giving semantics to it in terms of that kernel language. This idea is not exclusive to Ciao, but in Ciao the facilities that enable building up from a simple kernel are extensive and explicitly available from the system programmer level to the application programmer level.

Also, this mechanism to add new syntax to the language and give semantics to such syntax can be activated or deactivated on a per-compilation unit basis without interfering with others. In fact, all Ciao operators, "builtins," and most other syntactic and semantic language constructs are user-modifiable and live in *libraries*. Using these facilities, Ciao provides the programmer with a large number of useful features from different programming paradigms and styles, and the use of each of these features can be turned on and off at will for each program module. Thus, a given module may be using, e.g., higher order functions and constraints, while another module may be using imperative operations, objects, predicates, Prolog meta-programming builtins, and concurrency.

Conclusions

We believe that Ciao has pushed and is continuing to push the state of the art in solving the currently very relevant and challenging conundrum between statically and dynamically checked languages. It pioneered what we believe is the most promising approach in order to be able to obtain the best of both worlds: the combination of a flexible, multi-purpose assertion language with strong program analysis technology. This allows support for dynamic language features while at the same time having the capability of achieving the performance and efficiency of static systems. We believe that a good part of the power of the Ciao approach also comes from the synergy that arises from using the same framework and assertion language for different tasks (static verification, run-time checking, unit testing, documentation, ...) and its interaction with the design of Ciao itself (its module system, its extensibility, or the support for predicates and constraints).

References

1. Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: a Step towards Reconciling Dynamically and Statically Typed OO Languages. In *DLS '07: Proceedings of the 2007 Symposium on Dynamic Languages*, pages 53–64, New York, NY, USA, 2007. ACM.

2. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla-(Eds.). The Ciao System. Ref. Manual (v1.13). Technical report, School of Computer Science, T.U. of Madrid (UPM), 2009. Available at `http://www.ciaohome.org`.
3. F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.
4. Robert Cartwright and Mike Fagan. Soft Typing. In *Programming Language Design and Implementation (PLDI 1991)*, pages 278–292. SIGPLAN, ACM, 1991.
5. ECMA International. ECMAScript Language Specification, Standard ECMA-262, Edition 5. Technical report, September 2009. Available at `http://wiki.ecmascript.org`.
6. M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.
7. M. Hermenegildo and The CLIP Group. Some Methodological Issues in the Design of CIAO - A Generic, Parallel, Concurrent Constraint System. In *Principles and Practice of Constraint Programming*, number 874 in LNCS, pages 123–133. Springer-Verlag, May 1994.
8. M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.
9. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2), 2005.
10. M. Hermenegildo and The Ciao Development Team. Why Ciao? –An Overview of the Ciao System's Design Philosophy. Technical Report CLIP7/2006.0, Technical University of Madrid (UPM), School of Computer Science, UPM, December 2006. Available from: `http://cliplab.org/papers/ciao-philosophy-note-tr.pdf`.
11. M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J.F. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. Technical Report CLIP2/2010.0, Technical University of Madrid (UPM), School of Computer Science, March 2010. Under consideration for publication in *Theory and Practice of Logic Programming (TPLP)*.
12. P. Hudak, S. Peyton-Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzman, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the Programming Language Haskell. *Haskell Special Issue, ACM Sigplan Notices*, 27(5), 1992.
13. Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In Carla Schlatter Ellis, editor, *USENIX Annual Technical Conference, General Track*, pages 275–288. USENIX, 2002.
14. Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems*, 21(3):14, May 1999.

15. Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Asp. Comput.*, 19(2):159–189, 2007.
16. Francesco Logozzo et al. Clousot. `http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx`.
17. E. Mera, P. López-García, and M. Hermenegildo. Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In *25th International Conference on Logic Programming (ICLP'09)*, number 5649 in LNCS, pages 281–295. Springer-Verlag, July 2009.
18. Mozilla. Tamarin Project, 2008. Available at `http://www.mozilla.org/projects/tamarin/`.
19. MSR. Code contracts. `http://research.microsoft.com/en-us/projects/contracts/`.
20. George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.
21. G. Puebla, F. Bueno, and M. Hermenegildo. A Generic Preprocessor for Program Validation and Debugging. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 63–107. Springer-Verlag, September 2000.
22. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.
23. A. Rigo and S. Pedroni. PyPy's Approach to Virtual Machine Construction. In *Dynamic Languages Symposium 2006*. ACM Press, October 2006.
24. Z. Somogyi, F. Henderson, and T. Conway. The Execution Algorithm of Mercury: an Efficient Purely Declarative Logic Programming Language. *Journal of Logic Programming*, 29(1–3):17–64, October 1996.
25. Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 395–406. ACM, 2008.