

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 implementation of the dynamically-typed subset of Alore with extensive documentation is available for download¹.

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.

$$\begin{aligned}
L &::= \text{class } C \text{ extends } C \{ \bar{T} \bar{f}; K \bar{M}; \} \\
K &::= C(\bar{T} \bar{f}) \{ \text{super}(\bar{f}); \text{this}.\bar{f}=\bar{f}; \} \\
M &::= T m(\bar{T} \bar{x}) \{ \text{return } e; \} \\
e &::= x \mid e.f \mid e.m(\bar{e}) \mid \text{new } C(\bar{e}) \mid (T)e
\end{aligned}$$
Fig. 1. Syntax of FJ[?].

of the inheritance hierarchy. Several notational conventions simplify the presentation. \bar{x} stands for the sequence x_1, \dots, x_n . The comma operator is also used for sequence concatenation. $\bar{C} \bar{f}$ is shorthand for $C_1 f_1; \dots; C_n f_n$. Similarly, we use $\text{this}.\bar{f}=\bar{f}$ to mean $\text{this}.f_1=f_1; \dots; \text{this}.f_n=f_n$. 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:

$$\begin{array}{c}
T <: T \qquad \frac{S <: T \quad T <: U}{S <: U} \qquad \frac{\text{class } C \text{ extends } D \{ \dots \}}{C <: D} \\
\\
T \sim T \qquad T \sim ? \qquad ? \sim T \qquad \frac{S <: T}{S \lesssim T} \qquad \frac{S \sim T}{S \lesssim T}
\end{array}$$

The auxiliary function $fields(C) = \bar{T} \bar{f}$ gives the field names and their types of class C . $mtype(m, C) = \bar{T} \rightarrow T$ gives the signature of method m of class C . $mbody(m, C) = \bar{x}. e$ 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 type-checks 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

$$\begin{array}{c}
\frac{fields(C) = \bar{T} \bar{f}}{new\ C(\bar{e}).f_i \longrightarrow e_i} \text{ (R-FIELD)} \qquad \frac{mbody(m, C) = \bar{x}. e_0}{new\ C(\bar{e}).m(\bar{d}) \longrightarrow [\bar{d}/\bar{x}, C(\bar{e})/this]e_0} \text{ (R-INVK)} \\
\\
\frac{C <: T}{(T)new\ C(\bar{e}) \longrightarrow new\ C(\bar{e})} \text{ (R-CAST)} \qquad (?)new\ C(\bar{e}) \longrightarrow new\ C(\bar{e}) \text{ (R-DYCAST)} \\
\\
\frac{e_0 \longrightarrow e'_0}{e_0.f \longrightarrow e'_0.f} \text{ (RC-FIELD)} \qquad \frac{e_0 \longrightarrow e'_0}{e_0.m(\bar{e}) \longrightarrow e'_0.m(\bar{e})} \text{ (RC-INVK-RECV)} \\
\\
\frac{e_i \longrightarrow e'_i}{e_0.m(\dots, e_i, \dots) \longrightarrow e_0.m(\dots, e'_i, \dots)} \text{ (RC-INVK-ARG)} \\
\\
\frac{e_i \longrightarrow e'_i}{new\ C(\dots, e_i, \dots) \longrightarrow new\ C(\dots, e'_i, \dots)} \text{ (RC-NEW-ARG)} \qquad \frac{e_0 \longrightarrow e'_0}{(T)e_0 \longrightarrow (T)e'_0} \text{ (RC-CAST)}
\end{array}$$

Fig. 2. Reduction rules for FJ².

$$\begin{array}{c}
\text{Expression typing:} \\
\frac{\Gamma \vdash e : ?}{\Gamma \vdash e.f : ?} \text{ (T-DYFIELD)} \qquad \frac{\Gamma \vdash e : ? \quad \Gamma \vdash \bar{e} : \bar{T}}{\Gamma \vdash e.m(\bar{e}) : ?} \text{ (T-DYINVK)} \\
\\
\frac{\Gamma \vdash e : C \quad mtype(m, C) = \bar{S} \rightarrow T \quad \Gamma \vdash \bar{e} : \bar{T} \quad \bar{T} \lesssim \bar{S}}{\Gamma \vdash e.m(\bar{e}) : T} \text{ (T-INVK)} \\
\\
\frac{fields(C) = \bar{S} \bar{f} \quad \Gamma \vdash \bar{e} : \bar{T} \quad \bar{T} \lesssim \bar{S}}{\Gamma \vdash new\ C(\bar{e}) : C} \text{ (T-CREAT)} \quad \frac{\Gamma \vdash e : T}{\Gamma \vdash (?)e : ?} \text{ (T-DYCAST)} \\
\\
\frac{\Gamma \vdash e : ?}{\Gamma \vdash (C)e : C} \text{ (T-DYCAST2)} \\
\\
\text{Method typing:} \\
\frac{\bar{x} : \bar{T}, this : C \vdash e_0 : S_0 \quad S_0 \lesssim T_0 \quad \text{class } C \text{ extends } D \{ \dots \} \\ \text{if } mtype(m, D) = \bar{U} \rightarrow U_0, \text{ then } \bar{T} = \bar{U} \text{ and } T_0 = U_0}{T_0\ m(\bar{T}\ \bar{x}) \{ \text{return } e_0; \} \text{ OK IN } C} \text{ (T-METHOD)}
\end{array}$$

Fig. 3. Selected typing judgments for FJ².

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 FJ² 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 $\Gamma \vdash e \rightsquigarrow e' : T$, which translates FJ[?] expressions to FJ_G[?], which is FJ[?] augmented with *guarded expressions* $\langle C \rangle e$ and *guarded method invocations* $\langle e.m(\bar{e}) \rangle_{\bar{T}}$. 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 $\langle \langle T \Leftarrow S \rangle \rangle$ form, but to limit the number of guards, only when we cannot statically check the compatibility of the types:

$$\frac{\Gamma \vdash e : C \quad mtype(m, C) = \bar{S} \rightarrow S \quad \Gamma \vdash \bar{e} : \bar{T} \quad \bar{T} \lesssim \bar{S}}{\Gamma \vdash e.m(\bar{e}) \rightsquigarrow e.m(\langle \langle S_1 \Leftarrow T_1 \rangle \rangle e_1, \dots) : S}$$

The $\langle \langle \dots \rangle \rangle$ 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):

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

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

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:

$$\frac{C <: D}{\langle D \rangle_{\text{new}} C(\bar{e}) \longrightarrow \text{new } C(\bar{e})} \quad \frac{C \not<: D \quad \text{log error}}{\langle D \rangle_{\text{new}} C(\bar{e}) \longrightarrow \text{new } C(\bar{e})}$$

Inserting guards to method invocations with base type $?$ is postponed until evaluation, as shown below. Note that a guarded method invocation $\langle e.m(\bar{e}) \rangle_{\bar{T}}$ 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:

$$\frac{\Gamma \vdash e : ? \quad \Gamma \vdash \bar{e} : \bar{T}}{\Gamma \vdash e.m(\bar{e}) \rightsquigarrow \langle e.m(\bar{e}) \rangle_{\bar{T}} : ?}$$

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

$$\frac{mtype(m, C) = \bar{S} \rightarrow S}{\langle \text{new } C(\bar{e}).m(\bar{e}) \rangle_{\bar{T}} \longrightarrow \text{new } C(\bar{e}).m(\langle \langle S_1 \Leftarrow T_1 \rangle \rangle e_1, \dots)}$$

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:

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

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 FJ⁷ 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 FJ⁷ 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 $(A) \rightarrow X \wedge (B) \rightarrow Y$, 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 $\text{Type} \wedge (\text{Str}) \rightarrow \text{Int}$ (here \rightarrow 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 $\text{Function} \wedge (\text{Str}) \rightarrow \text{Int}$ or simply $(\text{Str}) \rightarrow \text{Int}$.

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, FJ² 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 library³ 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., Lebrésne, 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)