Design and Evaluation of Gradual Typing for Python

Michael M. Vitousek Andrew Kent
Jeremy G. Siek
Indiana University Bloomington
{mvitouse,andmkent,jsiek}@indiana.edu

Jim Baker
Rackspace Inc.
jim.baker@python.org

Abstract

Combining static and dynamic typing within the same language offers clear benefits to programmers. It provides dynamic typing in situations that require rapid prototyping, heterogeneous data structures, and reflection, while supporting static typing when safety, modularity, and efficiency are primary concerns. Siek and Taha (2006) introduced an approach to combining static and dynamic typing in a fine-grained manner through the notion of type consistency in the static semantics and run-time casts in the dynamic semantics. However, many open questions remain regarding the semantics of gradually typed languages.

In this paper we present Reticulated Python, a system for experimenting with gradual-typed dialects of Python. The dialects are syntactically identical to Python 3 but give static and dynamic semantics to the type annotations already present in Python 3. Reticulated Python consists of a typechecker and a source-to-source translator from Reticulated Python to Python 3. Using Reticulated Python, we evaluate a gradual type system and three approaches to the dynamic semantics of mutable objects: the traditional semantics based on Siek and Taha (2007) and Herman et al. (2007) and two new designs. We evaluate these designs in the context of several third-party Python programs.

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

Keywords gradual typing, case study, python, proxy

1. Introduction

Static and dynamic typing are well-suited to different programming tasks [19]. Static typing excels at documenting and enforcing constraints, enabling IDE support such as auto-completion, and helping compilers generate more efficient code. Dynamic typing, on the other hand, supports rapid prototyping and the use of metaprogramming and reflection. Because of these tradeoffs, different parts of a program may be better served by one typing discipline or the other. Further, the same program may be best suited to different type systems at different points in time, e.g., evolving from a dynamic script into a statically-typed program.

For this reason, combining static and dynamic typing within a single language and type system has been a popular goal, especially in the last decade. Early approaches include those of Abadi et al. [2], Thatte [31], and Bracha and Griswold [7]. Siek and Taha [26] introduced the *gradual typing* approach to merging such systems using a notion of type consistency together with higher-order casts. Numerous researchers have integrated gradual typing with other language features (Gronski et al. [14], Herman et al. [15], Siek and Taha [27], Wolff et al. [36], and Takikawa et al. [30]). Other researchers have adapted the notion of *blame tracking* [11] to gradual typing, reporting useful information when type casts fail (Tobin-Hochstadt and Felleisen [32], Wadler and Findler [35], and Siek and Wadler [28]).

In this paper, we present *Reticulated Python*,[1] a framework for developing gradual typing for the Python language. Reticulated uses a type system based on the first-order object calculus of Abadi and Cardelli [1], including structural object types. We augment this system with the dynamic type and *open* object types. Reticulated uses Python 3's annotation syntax for type annotations and a dataflow-based type inference system to infer types for local variables. Reticulated is available for download at `https://github.com/mvitousek/reticulated`.

Reticulated Python is implemented as a source-to-source translator that accepts syntactically valid Python 3 code, typechecks this code, and generates Python 3 code, which it then executes. The dynamic semantics of Reticulated differs from Python 3 in that run-time checks occur where implicit casts are needed to mediate between static and dynamically typed code. The run-time checks are implemented as calls into a Python library that we developed. In this way, we achieve a system of gradual typing for Python that is portable across different Python implementations and which may be applied to existing Python projects. We also made use of Python's tools for altering the module import process to insure that all imported modules are typechecked and translated at load time.

In addition to serving as a practical implementation of a gradually typed language, Reticulated serves as a test bed for experimenting with design choices for the semantics of casts between static and dynamically-typed code. We implemented and evaluated three distinct cast semantics for mutable objects: 1) the traditional proxy-based approach of Siek and Taha [27] and Herman et al. [15], but optimized with threesomes [28], 2) an approach that does not use proxies but involves ubiquitous lightweight checks, and 3) an approach in which casts cause the runtime types of objects to become monotonically more precise. We refer to these designs as the *guarded*, *transient*, and *monotonic* semantics. The guarded system is relatively complicated to implement and does not preserve object identity, which we found to be a problem in practice (see Section 3.2.5). The transient approach is straightforward to implement

[1] Named for *Python reticulatus*, the largest species of snake in the python genus. "Reticulated" for short.

and preserves object identity, but when runtime checks discover that type constraints have been violated, it is unable to report the location of the original cast that caused the error to occur, a process known as blame tracking. It therefore is less helpful when debugging cast errors. Monotonic preserves object identity and enables zero-overhead access of statically-typed object fields but requires locking down object values to conform to the static types they have been cast to.

To evaluate the usability of these designs, we performed case studies in which we applied Reticulated to several third-party codebases. We annotated them with types and then ran them using Reticulated. The type system design faired well, e.g. enabling statically-checked versions of statistics and cryptographic hashing libraries, while requiring only light code modifications. Further, our experiments detected several bugs extant in these projects. The experiments also revealed tensions between backwards compatibility and the ability to statically type portions of code. This tension is particularly revealed by the choice of whether to give static types to literals. Our experiments also confirmed results from other languages [33]: type systems for Python should provide some form of occurrence typing to handle design patterns that rely heavily on runtime dispatch. Regarding the evaluation of the three cast semantics, our results indicate that the proxies of the guarded design can be problematic in practice due to the presence of identity tests in Python code. The transient and monotonic semantics both faired well.

Gradual typing for Python Regardless of the choice of semantics, gradual type systems allow programmers to control of which portions of their programs are statically or dynamically typed. In Reticulated, this choice is made in function definitions, where parameter and return types can be specified, and in class definitions, where we use Python decorators to specify the types of object fields. When no type annotation is given for a parameter or object field, Reticulated gives it the dynamic (aka. unknown) type named `Dyn`. The Reticulated type system allows implicit casts to and from `Dyn`, as specified by the *consistency* relation [26]. In the locations of these implicit casts, Reticulated inserts casts to ensure, at runtime, that the value can be coerced to the target type of the cast.

One of the primary benefits of gradual typing over dynamic typing is that it helps to detect and localize errors. For example, suppose a programmer misunderstands what is expected regarding the arguments of a library function, such as the `moment` function of the `stat.py` module, and passes in a list of strings instead of a list of numbers. In the following, assume `read_input_list` is a dynamically typed function and the value it produces is a list of strings.

```
1 lst = read_input_list()
2 moment(lst, m)
```

In a purely dynamically typed language, or in a gradually typed language that does not insert casts (such as TypeScript), a runtime type error will occur deep inside the library, perhaps not even in the `moment` function itself but inside some other function it calls, such as `mean`. It is then challenging for the library's client to fix the problem, since they may not know the precise cause nor even if it is in fact their fault, rather than a bug in the library. On the other hand, if library authors make use of gradual typing to annotate the parameter types of their functions, then the error can be localized and caught before the call to `moment`, resulting in easier debugging. The following shows the `moment` function with annotated parameter and return types.

```
def moment(inlist: List(Float), m: Int)->Float:
  ...
```

labels	ℓ		type variables X
base types	B	::=	`Int` | `String` | `Float` | `Bool` | `Bytes`
types	T	::=	`Dyn` | B | X | `List`(T) | `Dict`(T,T) | `Tuple`$(\overline{T})$ | `Set`(T) | `Object`$(X)\{\overline{\ell{:}T}\}$ | `Class`$(X)\{\overline{\ell{:}T}\}$ `Function`(P,T)
parameters	P	::=	`Arb` | `Pos`$(\overline{T})$ | `Named`$(\overline{\ell{:}T})$

Figure 1. Static types for Reticulated programs

With this change, the runtime error points to the call to `moment` on line 2, where an implicit cast from `Dyn` to `List(Float)` occurred. A programmer using a library with gradual types need not use static types themselves — they gain the benefit of early localization and detection of errors even if they continue to write their own code in a dynamically typed style.

Casts on base values like integers and booleans are straightforward — either the value is of the expected type, in which case the cast succeeds and the value is returned, or the value is not, in which case it fails. However, casts on mutable values, such as lists, or higher-order values, such as functions and objects, are more complex. For mutable values, it is not enough to verify that the value meets the expected type at the site of the implicit cast because the value can later be mutated to violate the cast's target type. We discuss this issue in detail in Section 2.2.

Contributions

- We develop Reticulated Python, a source-to-source translator that implements gradual typing on top of Python 3.
- In Section 2, we discuss Reticulated's type system and discuss three dynamic semantics for mutable objects, one of which is based on proxies and two new approaches: one based on use-site checks and one in which objects become monotonically more precisely typed.
- In Section 3, we carry out several case studies of applying gradual typing to third-party Python programs.
- In Section 4, we evaluate the source-to-source translation approach and consider lessons for other implementers of gradually-typed languages.

2. The Reticulated Python Designs

We describe three language designs for Reticulated Python: guarded, transient, and monotonic. The three designs share the same static type system (Section 2.1) but have different dynamic semantics (Sections 2.2.1, 2.2.2, and 2.2.3).

2.1 Static semantics

From a programmer's perspective, the main way to use Reticulated is to annotate programs with static types. Source programs are Python 3 code with type annotations on function parameters and return types. For example, the definition of a `distance` function could be annotated to require integer parameters and return an integer.

```
def distance(x: Int, y: Int)-> Int:
  return abs(x - y)
```

In Python 3, annotations are arbitrary Python expressions that are evaluated at the function definition site but otherwise ignored. In Reticulated, we restrict the expressions that can appear in annotations to the type expressions shown in Figure 1 and to aliases for

$$\frac{}{\Gamma \vdash \texttt{class}\ \ X\colon \overline{\ell_k{:}T_k = e_k} : \texttt{Class}(X)\{\overline{\ell_k{:}T_k}\}}$$

$$\frac{\Gamma \vdash e : \texttt{Class}(X)\{\overline{\ell_k{:}T_k}\} \qquad \nexists k.\ \texttt{\_\_init\_\_} = \ell_k}{\Gamma \vdash e() : \texttt{Object}(X)\{\overline{\ell_k{:}\mathit{bind}(T_k)}\}}$$

$$\frac{\Gamma \vdash e : T \qquad T = \texttt{Object}(X)\{\ell{:}T_\ell, \ldots\}}{\Gamma \vdash e.\ell : T_\ell[X/T]}$$

Figure 2. Class and object type rules.

object and class types. The absence of an annotation implies that the parameter or return type is the dynamic type `Dyn`.

The type system is primarily based on the first-order object calculus of Abadi and Cardelli [1] with several important differences discussed in this section.

2.1.1 Function types

Reticulated's function parameter types have a number of forms, reflecting the ways in which a function can be called. Python function calls can be made using keywords instead of positional arguments; for example, the `distance` function can be called by explicitly setting `x` and `y` to desired values like `distance(y=42, x=25)`. To typecheck calls like this, we include the names of parameters in our function types using the `Named` parameter specification, so in this case, the type of `f` is `Function(Named(x:Dyn, y:Dyn), Dyn)`. On the other hand, higher-order functions most commonly call their function parameters using positional arguments, so for such cases we provide the `Pos` annotation. For example, the `map` function would take a parameter of type `Function(Pos(Dyn), Dyn)`; any function that takes a single parameter may then be passed in to `map`, because a `Named` parameters type is a subtype of a `Pos` when their lengths and element types correspond. Function types with `Arb` (for arbitrary) parameters can be called on any form of argument.

2.1.2 Class and object types

Reticulated includes types for both objects and classes, because classes are also runtime values in Python. Both of these types are structural, mapping attribute names to types, and the type of an instance of a class may be derived from the type of the class itself.

As an example of class and object typing, consider the following example:

```
1 class 1DPoint:
2   def move(self:1DPoint, x:Int)->1DPoint:
3     self.x += x
4     return self
5 p = 1DPoint()
```

Here the variable `1DPoint` has the type

```
1 Class(1DPoint){ move : Function(Named(self:1DPoint, x:
      Int),1DPoint) }
```

The occurrence of `1DPoint` inside of the class's structural type is a type variable bound to the type of an *instance* of `1DPoint`. Figure 2 shows Reticulated's typing rule for class definitions in the simple case where the class being defined has no superclass; classes that do have superclasses with static types also include the superclasses' members in their own type, and check that their instances' type is a subtype of that of their superclasses.

Values with class type may be invoked as though they were functions, as shown in the second rule of Figure 2. In the above example, `p` has type

```
Object(1DPoint) {move:Function(Named(x:Int),1DPoint)}
```

This type is derived from the class type of `1DPoint` by removing the self parameter of all the functions in the class' type. The type parameter `1DPoint` represents the self type. We use the *bind* metafunction to convert function definitions from unbound form — with an explicit self-reference as their first parameter — to a form with this parameter already bound and thus invisible. If the self-referential type parameter `X` in an object type is not used in the types of any of its members we write `Object{...}` instead of `Object(X){...}`.

The type system also includes a rule to handle the situation when the class defines an `__init__` method, in which case Reticulated checks that the arguments of the construction call site match the parameters of `__init__`.

In Python, programmers can dynamically add properties to objects at will. In recognition of this, Reticulated's object types are *open* with respect to consistency — two object types are consistent if one type has members that the other does not and their common members are consistent; in other words, implicit downcasts on width subtyping are allowed and checked at runtime. This can be seen in line 3 of the above example: `x` is not part of the type of a `1DPoint`, so when `x` is accessed, an implicit downcast on `self` occurs to check that `x` exists. In this sense, Reticulated's object types are similar to the bounded dynamic types of Ina and Igarashi [16], except that their approach is appropriate for nominal type systems while our open objects are appropriate for structural typing.

Programmers specify that object instances should have statically typed fields by using the `@fields()` decorator and supplying the expected field types. For example,

```
1 @fields({'x':Int, 'y':Int})
2 class 2DPoint:
3   def __init__(self:2DPoint):
4     self.x = 42
5     self.y = 21
6 2DPoint().x
```

This class definition requires that an instance of `2DPoint` have `Int`-typed fields named `x` and `y`; this information is included in the type of an instance of `2DPoint`, so the field access at line 6 has type `Int`. If `2DPoint`'s constructor fails to produce an object that meets this type, a runtime cast error is raised.

Lists, tuples, sets, and dictionaries have special, builtin types but they are also given object types that are used when they are the receiver of a method call.

2.1.3 Recursive type aliases

Structural object types are an appropriate match for Python's duck typing, but structural types can be rather verbose. To ameliorate this problem, we let class names be aliases for the types of their instances, as inferred from the class definition. Such aliases are straightforward when the aliased type only contains function and base types; however, obtaining the correct type for a given alias becomes more interesting in mutually recursive classes such as the following.

```
1 class A:
2   def foo(self, a:A, b:B):
3     pass
4 class B:
5   def bar(self, a:A, b:B):
6     pass
```

In the above code, `A` and `B` are type aliases when they appear in the annotations of `foo` and `bar`. For the remainder of this example, we use $\hat{\texttt{A}}$ and $\hat{\texttt{B}}$ to represent type aliases and the unadorned `A` and `B` as bound type variables. The task here is to determine the appropriate

$$
\begin{array}{rcl}
\texttt{Obj}(Y)\{\overline{\ell_i{:}T_i}\}[\hat{X}/T] & \longrightarrow & \texttt{Obj}(Y)\{\overline{\ell_i{:}T_i[\hat{X}/T']}\} \\
 & \text{where} & T' = T[\hat{Y}/Y] \\
\hat{X}[\hat{X}/T] & \longrightarrow & T \\
\texttt{List}(T_1)[\hat{X}/T_2] & \longrightarrow & \texttt{List}(T_1[\hat{X}/T_2]) \\
 & \cdots &
\end{array}
$$

Figure 3. Alias substitution

types to substitute for $\hat{\texttt{A}}$ and $\hat{\texttt{B}}$ in the type annotations of `foo` and `bar`. To arrive at these types, we start with the mapping

$$
\begin{array}{l}
[\hat{\texttt{A}} \mapsto \texttt{Object(A)\{foo:Function}([\hat{\texttt{A}}, \hat{\texttt{B}}], \texttt{Dyn})\}, \\
\hat{\texttt{B}} \mapsto \texttt{Object(B)\{bar:Function}([\hat{\texttt{A}}, \hat{\texttt{B}}], \texttt{Dyn})\}]
\end{array}
$$

The right-hand side of each pair in this mapping then has all the other pairs substituted into it using the substitution algorithm in Figure 3. This substitution repeats until it reaches a fixpoint, at which point all type aliases will have been replaced by object types or type variables. In the case of this example, the final mapping is

$$
\left[\begin{array}{l}
\hat{\texttt{A}} \mapsto \quad \texttt{Object(A)\{foo:Function}([\texttt{A}, \\
\qquad\quad \texttt{Object(B)\{bar:Function}([\texttt{A}, \texttt{B}], \texttt{Dyn})\}], \\
\qquad \texttt{Dyn})\}
\end{array}\right]
$$

$\hat{\texttt{B}}$ receives a similar mapping.

2.1.4 Types of Literals

One of the objectives of Reticulated is to achieve at least the option of full backwards-compatibility with untyped Python code — that is, if a normal Python program is run through Reticulated, we would like the result to be observationally identical to what it would be if it were run directly. [2] This goal leads to certain surprising design choices, however. For example, it is natural to expect that an integer literal have type `Int`. However, that would lead us to statically reject a program like `42 + 'hello world'`. This is a valid Python program in that it produces a result when evaluated (an exception), and the programmer has not "opted in" to static typing by using type annotations. So, to maintain maximal backwards compatibility with Python, even ridiculous programs like this cannot be rejected statically! Therefore we offer a flag in the Reticulated system to switch between giving integer literals the type `Dyn` or `Int`, and similarly for other kinds of literals. In Section 3 we discuss the practical effect of this choice.

2.1.5 Load-time typechecking

Reticulated's type system is static in the sense that typechecking is a syntactic operation, performed on a module's AST. However, when a program first begins to run, it is not always possible to know which modules will be imported and executed. Reticulated's typechecking, therefore, happens at the load time of individual modules. This can mean that a static type error is reported after other modules of a program have already run.

Reticulated does attempt to perform static typechecking ahead of time: when a module is loaded, it tries to locate the corresponding source files for any further modules that need to be loaded, typecheck them, and import the resulting type information into the type environment of the importing module. This is not always possible, however — modules may be imported at runtime from locations not visible to Reticulated statically. In general, programmers cannot count on static type errors to be reported when a program starts to execute, only when the module with the error is loaded.

[2] There are some very few places where this is violated: for example, if a Python function has pre-existing annotations that are syntactically identical to our type annotations.

2.1.6 Dataflow-based type inference

Python 3 does not provide syntax for annotating the types of local variables. We might use function decorators or comments for this purpose, but we instead choose to infer types for local variables. We perform a simple intraprocedural dataflow analysis [17] in which each variable is given the type that results from joining the types of all the values assigned to it, a process which we repeat until a fixpoint is reached. For example, consider the function

```
1 def h(i:Int):
2   x = i; y = x
3   if random():
4     z = x
5   else: z = 'hello world'
```

We infer that `x` and `y` have type `Int`, because the expressions on the right-hand sides have that type, and that `z` has type `Dyn`, which is the join of `Int` and `Str`. This join operation is over the subtyping lattice with `Dyn` as top from Siek et al. [29], and always results in a type that can be safely casted to.

2.2 Dynamic semantics

Using the Reticulated framework, we explore three different dynamic semantics for casts. The intent behind each semantics is to prevent runtime values from observably inhabiting identifiers with incompatible types. Consider the following example in which a strong (type-changing) update occurs to an object to which there is an outstanding statically-typed reference.

```
1 class Foo:
2   bar = 42
3 def g(x):
4   x.bar = 'hello world'
5 def f(x:Object({bar:Int}))->Int:
6   g(x)
7   return x.bar
8 f(Foo())
```

Function `f` passes its statically-typed parameter to the dynamically-typed `g`, which updates the `bar` field to a string. Function `f` then accesses the `bar` field, expecting an `Int`.

We could choose to allow such type errors — this is the approach taken by TypeScript [25] and described by Siek and Taha [26]. However, we prefer to detect inconsistencies between the static types specified by the program and the runtime values. In this section, we discuss the design of our three dynamic semantics for casts, which perform this checking in very different ways.

2.2.1 The guarded dynamic semantics

The guarded semantics uses casts to detect and report runtime type errors. When using this semantics, Reticulated inserts casts into programs where implicit coercions occur, such as at function call sites (like line 6 above) and field writes. The inserted cast — which is a call to a Python function which performs the cast — is passed the value being casted as well as type tags for the source and target of the cast, plus an error message and line number that will be reported if and when the cast fails (we elide this error information in our examples). For example, our above program will have casts inserted as follows:

```
3 def g(x):
4   cast(x, Dyn, Object({bar:Dyn})).bar = 'hello world'
5 def f(x:Object({bar:Int}))->Int:
6   g(cast(x, Object({bar:Int}), Dyn))
7   return x.bar
8 f(cast(Foo(), Object({bar:Dyn}), Object({bar:Int})))
```

Casts between `Dyn` and base types will simply return the value itself (if the cast does not fail — if it does, it will produce a `CastError`

$$
\begin{array}{rcl}
\texttt{ref}\ v \mid \mu & \longrightarrow & a \mid \mu, a \mapsto v \\
 & & \text{where } a \notin dom(\mu) \\
cast(a, \texttt{ref}\ T_1, \texttt{ref}\ T_2) \mid \mu & \longrightarrow & a{::}\texttt{ref}\ T_1 \Rightarrow \texttt{ref}\ T_2 \mid \mu \\
!a \mid \mu & \longrightarrow & \mu(a) \mid \mu \\
!(v{::}\texttt{ref}\ T_1 \Rightarrow \texttt{ref}\ T_2) \mid \mu & \longrightarrow & cast(!v, T_1, T_2) \mid \mu
\end{array}
$$

Figure 4. Reduction rules for guarded references.

exception), but casts between function or object types produce a proxy. This proxy contains the error message and line number information provided by the static system for this cast, which serves as blame information if the cast's constraints are violated later on. Guarded proxies do not just implement individual casts — they represent compressed sequences of casts using the threesomes of Siek and Wadler [28]. In this way, proxies do not build up on themselves, layer after layer — a proxy is always only one step away from the actual, underlying Python value.

Method calls or field accesses on proxies redirect to a lookup on the underlying object, the result of which is casted from the part of the source type that describes the member's type to the same part of the meet type, and then from the meet type to the final target type. This process occurs in reverse for member writes. Proxied functions, when invoked, cast the parameters from target to meet to source, and then cast results from source to meet to target. In the above example, when the field write occurs at line 4, the proxy will attempt to cast `'hello world'` to `Int`, which is the most precise type that the object being written to has had for `bar` in its sequence of casts. This cast fails, and a cast error is reported to the programmer. Figure 4 shows how this process proceeds when using guarded references in a core calculus similar to that of Herman et al. [15].

One important consequence of the guarded approach is that *casts do not preserve object identity* — that is, if the expression

```
x is cast(x, Dyn, Object({bar:Int}))
```

were added to function `g` in the example above, it would return `False`. Similarly, the Python runtime type is not preserved: the type of a proxy is `Proxy`, not the type of the underlying object, and this information is observable to Python programs that invoke the builtin `type` function on proxies. However, the proxy class is a subclass of the runtime Python class of the underlying value,[3] instance checks generally return the same result with the bare value as they do with the proxied value. In our case studies in Section 3, we evaluate the consequences of this issue in real-world code.

2.2.2 The transient dynamic semantics

In the transient semantics, a cast checks that the value has a type consistent with the target type of the cast, but it does not wrap the value in a proxy. Returning to our running example, just as in the guarded semantics, a cast is inserted around the argument in the call to function `f`:

```
8 f(cast(Foo(), Object({bar:Dyn}), Object({bar:Int})))
```

However, in the transient system, this cast just checks that `Foo()` is an object, that it has a member named `bar`, and that `bar` is an `Int`. It then returns the object. The cast's effect is therefore *transient*; nothing prevents the field update at line 4. To prevent `f` from returning a string value, a check is instead inserted at the point where `f` reads from `bar`:

```
5 def f(x:Object({bar:Int}))->Int:
6   g(cast(x, Object({bar:Int}), Dyn))
7   return check(x.bar, Int)
```

This check attempts to verify that `x.bar` has the expected type, `Int`. Because the call to `g` mutates `x.bar` to contain a string, this check fails, preventing an uncaught type error from occurring. In addition, it is possible that `f` could be called from a context in which its type is not visible, if it was casted to type `Dyn` for example. In this case, the call site cannot check that the argument being passed to `f` is a `List(Int)`, and unlike the guarded system, there is no proxy around `f` to check that the argument has the correct type either. Therefore, `f` needs to check that its parameters have the expected type in its own context. Thus, the final version of the function becomes

```
5 def f(x:Object({bar:Int}))->Int:
6   check(x, Object({bar:Int}))
7   g(cast(x, Object({bar:Int}), Dyn))
8   return check(x.bar, Int)
```

In general, checks are inserted at the use-sites of variables with non-base types and at the entry to function bodies and for loops. Checks are used in these circumstances to verify that values have the type that they are expected to have in a given context before operations occur that may rely on that being the case.

Figure 5 shows an excerpt of the `has_type` function used within transient casts. Some values and types cannot be eagerly checked by `has_type` function, however, such as functions — all that Reticulated can do at the cast site is verify that a value is callable, not that it takes or returns values of a particular type. Function types need to be enforced by checks at call sites. Moreover, even when eager checking is possible, the transient design only determines that values have their expected types at time of the cast site, and does not detect or prevent type-altering mutations from occurring later — instead, such mutations are prevented from being observable by use-site checks. Figure 6 illustrates this with several typing and evaluation rules from a core calculus for the transient system using references. Note that in these semantics we do not even give a type to references beyond `ref` — type information about the contents of a reference is instead encoded in the syntax of the dereference. [4]

Transient and Dart's checked mode The transient semantics for Reticulated is reminiscent of Dart's *checked mode* [12], in which values are checked against type annotations (which are otherwise ignored at runtime). However, Dart's checks are performed under different circumstances than Reticulated's, and these choices cause Dart's type system to be unsound even when checked. Consider the following Dart program:

[3] Unless the underlying value's class is non-subclassable, such as `bool` or `function`.

[4] Interested readers may find the full calculus, including a proof of type safety, at `http://homes.soic.indiana.edu/mvitouse/transient.pdf`.

```
1 def has_type(val, ty):
2   if isinstance(ty, Dyn):
3     return True
4   elif isinstance(ty, Int):
5     return isinstance(val, int)
6   elif isinstance(ty, Object):
7     return all(hasattr(val, member) and has_type(
          getattr(val, member), ty.member_type(member))
           for member in ty.members)
8   elif isinstance(ty, Function):
9     return callable(val)
10  elif ...
```

Figure 5. Definition for the `has_type` function.

$$\frac{\Gamma; E \vdash e : T}{\Gamma; E \vdash \texttt{ref}\ e : \texttt{ref}} \qquad \frac{\Gamma; E \vdash e : \texttt{ref}}{\Gamma; E \vdash\ check(!e, T) : T}$$

$$\begin{array}{rcll} \texttt{ref}\ v \mid \mu & \longrightarrow & a \mid \mu, a \mapsto v & \text{where } a \notin dom(\mu) \\ check(!a, T) \mid \mu & \longrightarrow & \mu(a) \mid \mu & \text{if } \emptyset; dom(\mu) \vdash \mu(a){:}T \\ check(!a, T) \mid \mu & \longrightarrow & \texttt{error} \mid \mu & \text{otherwise} \end{array}$$

Figure 6. Typing and reduction rules for transient references.

```
1 void g(dynamic foo) {
2   foo[0] = "hello world";
3 }
4 void f(List<int> foo) {
5   g(foo);
6   print(foo[0]);
7 }
8 main() {
9   List<dynamic> foo = new List<dynamic>();
10  foo.add(42);
11  f(foo);
12 }
```

Dart does not check the value that results from the index at line 6, unlike transient Reticulated . Therefore "hello world" will be printed despite the presence of a string in a list of integers.

Additionally, in Dart, object updates are checked against the annotated field types of the underlying object, rather than the type of the reference to the object in the scope of the update, causing the following program to fail.

```
1 class Foo {
2   int bar = 42;
3 }
4 void f(dynamic a) {
5   a.bar = "hello world";
6 }
7 main() {
8   f(new Foo());
9 }
```

In transient Reticulated, the equivalent program would run without error until the `bar` field of the object is read in a context where it is expected to be an `int`. Dart's object update checks therefore behave more like guarded than transient.

2.2.3 The monotonic dynamic semantics

Like the transient semantics, the monotonic system avoids using proxies, but rather than using transient checks, the monotonic approach preserves type safety by restricting the types of objects themselves. In this approach, when an object flows through a cast from a less precise (closer to `Dyn`) type to a more precise one, the cast has the effect of locking the type of the object at this more precise type. Objects store a type for each of their fields; this type is always *equally or more precise* than any of the types at which the field has been viewed. For example, if an object has had references to it with types {`'foo':Tuple(Int, Dyn)`} and {`'foo':Tuple(Dyn, Int)`}, the object will record that `'foo'` must be of type `Tuple(Int, Int)`, because a value of this type is consistent with both of the references to it.

When field or method updates occur, the newly written value is cast to this precise type. Back to our ongoing example:

```
3 def g(x):
4   cast(x, Dyn, Object({bar:Dyn})).bar = 'hello world'
5 def f(x:Object({bar:Int}))->Int:
6   g(cast(x, Object({bar:Int}), Dyn))
7   return x.bar
8 f(cast(Foo(), Object({bar:Dyn}), Object({bar:Int})))
```

Under the monotonic semantics, casts are inserted at the same places as they are in guarded, but their effects are different. When the newly created object goes through the cast at line 8, the object is permanently set to only accept values of type `Int` in its `bar` field. When g attempts to write a string to the object at line 4, the string is cast to `Int`, which fails. This cast is performed by the object itself, not by a proxy — the values of `x` in `f` and `g` are the same even though they appear at different types.

The monotonic system's approach of permanently, monotonically locking down object types results is one clear difference from guarded and transient — it is not possible to pass an object from untyped code into typed code, process it, and then treat it as though it is once again dynamically typed. On the other hand, monotonic's key guarantee is that the type of the actual runtime value of an object is at least as precise as any reference to it. Because of this, when a reference is of fully static type, the value of the object has the same type as the reference, and no casts or checks are needed when a field is accessed. Even if the reference is of a partially dynamic type, only an upcast needs to occur.

Figure 7 shows an excerpt of the cast insertion and evaluation rules for a calculus with monotonic references. This shows that references whose types are entirely static may be directly dereferenced, and that the requirement of monotonicity is enforced by tracking the most precise type a cell has been casted to; the type is stored as part of the cell on the heap.[5]

2.2.4 Runtime overhead in statically-typed code

Readers may have noted that both the guarded and transient semantics impose runtime overhead on statically-typed code due to checks and casts, whereas the monotonic semantics imposes no runtime overhead on statically-typed code. With the guarded and transient semantics, avoiding these overheads is difficult in an implementation based on source-to-source translation, but if the underlying language offers hooks that enable optimizations based on type information, Reticulated's libraries could utilize such hooks. This is a promising approach in the PyPy implementation of Python, where an implementation of gradual typing could enable optimizations in the JIT, and in Jython, where an implementation could utilize bytecodes such as invokedynamic to reduce overhead.

[5] A full semantics and proof of type safety for a monotonic calculus is available at `http://wphomes.soic.indiana.edu/jsiek/files/2013/06/mono-ref-july-2014.pdf`

$$\text{heaps} \quad \mu \quad ::= \quad \epsilon \mid \mu, a \mapsto (v, T)$$

$$\frac{\Gamma; \Sigma \vdash e \rightsquigarrow e' : \texttt{ref}\ T \qquad T \text{ static}}{\Gamma; \Sigma \vdash !e \rightsquigarrow !e' : T}$$

$$\frac{\Gamma; \Sigma \vdash e \rightsquigarrow e' : \texttt{ref}\ T \qquad \neg T \text{ static}}{\Gamma; \Sigma \vdash !e \rightsquigarrow !e'@T : T}$$

$$\begin{array}{rcll} \texttt{ref}\ v@T \mid \mu & \longrightarrow & a \mid \mu, a \mapsto (v, T) & \\ & \text{where} & a \notin dom(\mu) & \\ !a@T_2 \mid \mu & \longrightarrow & cast(v, T_1, T_2) \mid \mu & \\ & \text{where} & \mu(a) = (v, T_1) & \\ !a \mid \mu & \longrightarrow & fst(\mu(a)) \mid \mu & \\ cast(a, \texttt{ref}\ T_1, \texttt{ref}\ T_2) \mid \mu & \longrightarrow & a \mid \mu, a \mapsto (v', T_4) & \\ & \text{where} & \mu(a) = (v, T_3), & \\ & & T_4 = T_2 \sqcup T_3, & \\ & & v' = cast(v, T_3, T_4) & \end{array}$$

Figure 7. Cast insertion and reduction rules for monotonic references.

3. Case Studies and Evaluation

To evaluate the design of Reticulated's type system and runtime systems, we performed case studies on existing, third-party Python programs. We annotated these programs with Reticulated types and ran them under each of our semantics. We categorized the code that we were unable to type statically and identified several additional features that would let more code be typed. We discovered several situations in which we had to modify the existing code to interact well with our system, and we also discovered several bugs in these programs in the process. The annotated case studies that we used in this experiment are available at `http://bit.ly/1rqSvQM`.

3.1 Case study targets

Python is a very popular language and it is used for many applications, from web backends to scientific computation. To represent this wide range of uses, we chose several different code bases to use as case studies for Reticulated.

3.1.1 Statistics library

We chose the Harvard neuroscientist Gary Strangman's statistics library[6] as a subject of our Reticulated case study as an example of the kind of focused numerical programs that are common in scientific Python code. The `stats.py` module contains a wide variety of statistics functions and the auxiliary `pstat.py` module provides list manipulation functions.

To prepare them for use with Reticulated, we removed the libraries' dependence on the external `Numeric` array library. This had the effect of reducing the amount of "Pythonic" dynamicity that exists in the libraries — prior to our modification, two versions of most functions existed, one for `Numeric` arrays and one for native Python data structures, and a dispatch function would redirect any call to the version suited to its parameter. Although we removed this dynamic dispatch from these modules, this kind of behavior is considered in our next case study. We then simply added types to the libraries' functions based on their operations and the provided documentation, and replaced calls into the Python `math` library with calls into statically typed math functions.

3.1.2 CherryPy

CherryPy[7] is a lightweight web application framework written for Python 3. It is object-oriented and makes heavy use of callback functions and dynamic dispatch on values. Our intent in studying CherryPy was to realistically simulate how library developers might use gradual typing to protect against bad user input and report useful error messages. To accomplish this, we annotated several functions in the CherryPy system with types. We tried to annotate client-facing API functions with types, but in many cases it was not possible to give specific static types to API functions, for reasons we discuss in Section 3.2.6. In these situations, we annotated the private functions that are called by the API functions instead.

3.1.3 SlowSHA

We added types to Stefano Palazzo's implementation of SHA1/SHA2.[8] This is a straightforward 400 LOC program that provides several SHA hash algorithms implemented as Python classes.

3.2 Results

By running our case studies in Reticulated, we made a number of discoveries about both our system and the targets of the studies themselves. We discovered two classes of bugs that exist in the target programs which were revealed by the use of Reticulated. We also found several deficiencies in the Reticulated type system which need to be addressed, and some challenges that the guarded system in particular faces due to its use of proxies: the CherryPy case study relies on checks against object identity, and the inability of the guarded semantics to preserve object identity under casts causes the program to crash.

3.2.1 Reticulated reveals bugs

We discovered two potential bugs in our target programs by running them with Reticulated.

Missing return values Below is one of the functions from `stats.py` that we annotated, shown post-annotation:

```
1 def betacf(a:Float,b:Float,x:Float)->Float:
2   ITMAX = 200
3   EPS = 3.0e-7
4 # calculations elided...
5   for i in range(ITMAX+1):
6 # more calculations elided...
7     if (abs(az-aold)<(EPS*abs(az))):
8       return az
9   print('a or b too big, or ITMAX too small in Betacf.
        ')
```

This function only conditionally returns a value; if the inputs are such that the for loop executes more than `ITMAX+1` times, the function "falls off" the end of its definition. In Python this has the effect of returning the unitary `None` value. This falling-off behavior poses a problem when `betacf` is called by other functions in the library, which do not check if the result is `None`. None of the test cases supplied by the developer trigger this bug, but it could be triggered by client code. This behavior could cause a confusing error on the caller's side, or even worse, it could continue to execute, likely producing an incorrect result. The printed error message could easily be missed by the programmer, especially since `stats.py` is a library, and it may be used by clients unfamiliar with the design of its functions. By annotating `betacf` with the static return type `Float`, we force the function to always either return a float value, or to raise an exception. In this case, the appropriate fix is to replace the final line with something like

```
9   raise Exception('a or b too big, or ITMAX too small
        in Betacf.')
```

which Reticulated's type system will accept.

Parameter name mismatch Reticulated's type system is designed to guarantee that when one class is a subclass of another, the type of an instance of the subclass is a subtype of the type of an instance of the superclass. Additionally, as we discuss in Section 2.1.1, function types can include the names of their parameters so that calls with keyword arguments may be typed.

These properties of Reticulated cause it to report a static type error when a subclass overrides a superclass's method *without* using the same parameter names. We regard this situation as a latent bug in a program that exhibits it, as we illustrate below:

```
1 class Node:
2   def appendChild(self, node):
3 # ...
4 class Entity(Node): # subclass of Node
5   def appendChild(self, newChild):
6 # ...
```

If the programmer expects that any instance `nd` of `Node` or its subclasses can be used in the same fashion, then they expect that

```
7 nd.appendChild(node=Node())
```

[6] `http://www.nmr.mgh.harvard.edu/Neural_Systems_Group/gary/python.html`

[7] `http://www.cherrypy.org/`

[8] `http://code.google.com/p/slowsha/`

will always succeed. However, if `node` is actually an instance of `Entity`, this call will result in a Python `TypeError`. This is an easy mistake to make, and we have found this pattern in multiple places — even within the official Python Standard Library itself, which is where this example arises.[9] We have not encountered situations where this potential bug actually results in a runtime error; it would be unusual for `node`/`newChild` to be used as a keyword argument.

3.2.2 The Reticulated type system in practice

The choice of whether or not to include typed literals, as discussed in Section 2.1.4, greatly affects the behavior of math-heavy code. Enabling typed literals requires some code to be slightly changed, but lets substantially more code be entirely statically typed.

Invariant mutable types and typed literals mix poorly. The statistics libraries we studied frequently intermingle integers and floating point values, including within lists, as in this example:

```
1 def var (inlist:List(Float))->Float:
2   ...
3   mn = mean(inlist)
4   deviations = [0]*len(inlist)
5   for i in range(len(inlist)):
6     deviations[i] = inlist[i] - mn
7   return ss(deviations)/float(n-1)
```

In this function, the `deviations` variable is initialized to be a list of integer zeros. Our type inferencer only reasons about assignments to variables, such as that at line 4, not assignments to attributes or elements, such as that at line 6. Therefore, when we let number literals have static types, our type inference algorithm will determine that the type of `deviations` is `List(Int)`. However, float values are written into it at line 6, and at line 7 it is passed into the function `ss`, which we have given the type `Function([List(Float)],Float)`. The Reticulated type system detects this call as a static type error because list element types are invariant under subtyping (though not under consistency); our subtyping rules contain the only the rule

$$\frac{}{\Gamma \vdash \texttt{List}(T_1) <: \texttt{List}(T_1)}$$

for lists. Even though we do define `Int` as a subtype of `Float`, neither the type `List(Int)` nor `List(Float)` is appropriate for `deviations` as written. In such situations, we can simply rewrite the code to only use float values; in this case we change line 4 to

```
4   deviations = [0.0]*len(inlist)
```

Typed literals and type inference allow math to be typed. When typed literals are enabled, and any necessary edits to the source are made as above, our approach to type inference allows many of the calculation-heavy statistics functions that we studied to become almost entirely statically typed, with few or no casts to or from `Dyn`. For example, the sum-of-squares function behaves extremely well:

```
1 def ss(inlist: List(float)) ->float:
2   _ss = 0
3   for item in inlist:
4     _ss = (_ss + (item * item))
5   return _ss
```

This code above actually shows this function *after* cast insertion — no casts have been inserted at all,[10] and the computations here occur entirely in typed code. Overall, when typed literals are enabled, 48% of the binary operations in `stats.py` occur in typed code, compared to only 30% when literals are assigned the dynamic type. Reticulated, as a test-bed for gradual typing in Python, is not currently able to make use of this knowledge, but a system that does perform type optimizations would be able to execute the mathematical operations in this function entirely on the CPU, without the runtime checks that Python typically must perform.

3.2.3 Cast insertion with open structural object types

In general, structural objects and object aliases worked well for our test cases. However, we discovered one issue that arose in all of our runtime systems because our rules for cast insertion on object accesses made an assumption that clashed with accepted Python patterns. One of our rules for cast insertion is

$$\frac{\Gamma \vdash e \rightsquigarrow e' : T \qquad T = \texttt{Object}(X)\{\overline{\ell_i{:}T_i}\} \qquad \forall i.\ \ell_i \neq x}{\Gamma \vdash e.x \rightsquigarrow \texttt{cast}(e', T, \texttt{Object}(X)\{x : \texttt{Dyn}\}).x : \texttt{Dyn}}$$

That is, when a program tries to read an object member that does not appear in the object's type, the object is cast to a type that contains that member at type `Dyn`. This design clashes with the common Python pattern below:

```
1 try:
2   value.field
3   # do something
4 except AttributeError:
5   # do something else
```

If the static type of `value` does not contain the member `field`, Reticulated casts it to verify at runtime that the field exists. This implicit downcast, allowed because of our *open* design for object types, causes this program to typecheck statically even if the static type of `value` does not contain `field` — without open object types this program would not even be accepted statically. However, even with this design, this program still has a problem: if and when that cast fails, a cast error is triggered, which would not let the program continue down the `except` branch as the programmer intended. In order to support the expected behavior of this program, we design our cast errors to be caught by the try/except block. Cast errors are implemented as Python exceptions, so by letting any cast errors generated by this particular kind of object cast actually be instances of a subclass of `AttributeError`, we anticipate the exception that would have been thrown without the cast. This specialized cast failure is then caught by the program itself if it was designed to catch `AttributeErrors`. Otherwise, the error is still reported to the programmer as a cast error with whatever blame information is supplied. We use a similar solution for function casts, since programs may call values if they were functions and then catch resulting exceptions if they are not.

3.2.4 Monotonic and inherited fields

The basic principle behind the monotonic approach is that when objects are cast to a more precise type, any future values that may inhabit the object's fields must meet this new precise type as well. However, it is not always clear *where* this "locking down" should happen. Field accesses on Python objects can return values that are not defined in the object's local dictionary but rather in the object's class or superclasses. Therefore, when a monotonic object goes through a cast that affects the type of a member that is defined in the object's class hierarchy, we have two choices: either we can downcast and lock that member in its original location, or we can copy it to the local object and lock it there. Both designs have problems: the former will cause all instances of the class to behave as though they had gone through the cast, while the latter causes

[9] In the Python 3.2 standard library, in `xml/minidom.py`.

[10] Using the guarded semantics with check insertion disabled. Transient checks would be inserted to check that `inlist` is a list of floats at the entry to the function, and that `item` is a `Float` at the beginning of each iteration of the loop.

class attributes to be eagerly copied into objects, damaging space efficiency and blinding instances to mutation of class attributes.

Initially we chose the former behavior, monotonically locking down fields and methods in their original locations. However, applying this behavior to lists in the statistics library revealed an additional problem: it is impossible to monotonically downcast and lock members of builtin classes, such as lists. Even if we had been able to do so, however, we would have made it so that there could only ever be one type of list in any Reticulated program. Currently, the monotonic system copies class fields to the dictionary of the casted object, but we intend to also provide the option, on a per-member basis as part of the type, to instead lock members in-place when that behavior is explicitly desired.

3.2.5 Challenges for guarded

The guarded design for gradually-typed Python causes several problems that do not occur in the transient system, because proxies do not preserve object or type identity. We discovered that the loss of object identity was a serious problem that prevented our CherryPy case study from even running successfully, and that the loss of type identity meant that we had to modify the statistics library case study for it to run.

Object identity is a serious issue... We were aware from the outset that our design for the guarded system does not preserve object identity. However, it was unclear how significant of a problem this is in practice. In our case studies, object identity was not relevant in the statistics library, but identity checks are used in CherryPy in a number of locations. In particular, there are places where the wrong outcome from an identity check causes an object to be incorrectly initialized, as in the following:

```
1 class _localbase(object):
2   def __new__(cls, *args, **kw):
3     ...
4     if args or kw and (cls.__init__ is object.__init__
          ):
5       raise TypeError("Initialization arguments are
            not supported")
6     ...
```

If the parameter `cls` is proxied, then `cls.__init__` will be as well. In that case, the identity check at line 4 will return `False` if the underlying value is `object.__init__`. The `TypeError` exception will not be raised, and confusing errors may occur later.

In many places where identity testing is used simply replacing `is` with `==` would have the same effect and be compatible with the use of proxies. However, object identity issues also arose from calls into the Python Standard Library, causing unpredictable, incorrect behavior. Python's pickling library is unable to correctly serialize proxied objects, and file reads on proxied file objects occasionally fail for reasons we have not yet been unable to determine. The end result of this was that the CherryPy webserver was unable to run without error under the guarded semantics. We would need to change the pickling and file IO libraries at minimum to get the CherryPy webserver to work.

...and type identity sometimes is. Although the statistics library never checks object identity, we found 28 static code locations where it uses the `type` function to get a value's Python type. At these sites, unexpected behavior can arise because our proxies' Python types are not the same as the Python types of their underlying objects. Fortunately, these situations only required minor edits to resolve. One problematic location was the following code from `pstat.py`:

```
1 def abut (source, *args)->List(Dyn):
2   if type(source) not in [list, tuple]:
3     source = [source]
4   ...
```

If the value of `source` is a proxy, then the call to `type` on line 2 will return the `Proxy` class, resulting in `source` being wrapped in a list even if the underlying value already is a list. This can be solved by providing a substitute `type` function that is aware of proxies, or by rewriting this line of code to use `isinstance`:

```
2   if not any(isinstance(source, t) for t in [list,
        tuple]):
```

We arrange for the class of a proxy to be a subclass of the class of the underlying value, and with this modification, we were able to run the statistics library under the guarded semantics.

3.2.6 Classifying statically untypable code

In many circumstances we found functions that could not be given a fully static type. This is, of course, to be expected — Python is a dynamic language and many Python programs are written in an idiom that depends on dynamic typing. Wisely-chosen extensions of Reticulated's static type system would let certain classes of these currently-untypable functions be statically checked, but sometimes the appropriate thing to do is just use `Dyn`. We classify these situations, discuss how frequently they arise, and consider which features, if any, would allow them to be assigned static types.

Dynamically typed parameters may act like generics. One deficiency of Reticulated's type system is its lack of support for generics. Dynamic typing lends itself well to a generic style of programming, and thus it is no surprise that many of the functions and portions of code that we could not assign static types to would by much more typable if our type system provided generics. Near-future work will involve implementing such types, possibly as an adaptation of Ahmed et al. [3].

Dispatch occurs on Python runtime types. The below code snippet — the definition of the `update` method invoked in the previous example — shows a pattern used extensively in CherryPy which is difficult to type precisely:

```
1 def update(self, config):
2   if isinstance(config, string):
3     config = Parser().read(config).as_dict()
4   elif hasattr(config, 'read'):
5     config = Parser().readfp(config).as_dict()
6   else:
7     config = config.copy()
8   self._apply(config)
```

(This snippet actually combines together two CherryPy functions for clarity.) The method `update` may take any of three kinds of values: a string, a value with the attribute "`read`" (contextually, a file), or something else, but whatever it initially is, it is eventually converted into a value which is passed into the `_apply` method.

It is clear that `config` cannot be annotated with a single precise type — the logic of the program depends on the fact that `config` may be any of several disparate types. We could possibly achieve some kind of static typing by introducing sum types into the language, or by using the occurrence typing of Tobin-Hochstadt and Felleisen [34]. In the absence of such techniques we can still perform useful checking by annotating the `_apply` function instead, shown below:

```
9 def _apply(self, config:Dict(Str, Dyn))->Void:
10   ...
```

When we declare that this function accepts only `config` values that are dictionaries with string keys, a cast will be inserted at the call to `_apply` in `update` at line 8. This ensures that, no matter what the value passed in to `update` was, the one handed off to `_apply` will have the correct type.

Data structures can be heterogeneous. Dynamic typing enables the easy use of heterogeneous data structures, which naturally cannot be assigned static types. In our case studies, we did not see significant use of heterogeneous lists, other than ones that contain both `Int`s and `Float`s as we discuss in Section 3.2.2. The same is *not* true of dictionaries, whose values frequently display significant heterogeneity in CherryPy, as seen in this call into CherryPy's API from a client program:

```
1 cherrypy.config.update({
2   'tools.encode.on': True,
3   'tools.encode.encoding': 'utf-8',
4   'tools.staticdir.root': os.path.abspath(os.path.
        dirname(__file__)),
5 })
```

This dictionary, which contains only strings as keys but both strings and booleans as values, is representative of many similar configuration dictionaries used in CherryPy and other Python libraries. Reticulated can represent such heterogeneous types — this dictionary could be given the type `Dict(Str, Dyn)`. This example could be given a more precise type if we introduced sum types into the Reticulated type system.

`eval` and other villains Finally there are cases where fundamentally untypable code is used, such as the `eval` function. The effect of eval and its close relative `exec` is, of course, unpredictable at compile time. In some cases, `eval` is used in rather astonishing fashions reminiscent of those described by Richards et al. [23]:

```
1 def dm (listoflists,criterion):
2   function = 'filter(lambda x: '+criterion+',
        listoflists)'
3   lines = eval(function)
4   return lines
```

This `pstat.py` function is evidently written for the use of novice programmers who do not understand how to use lambdas but who still wish to use higher order functions. Examples like this, and other bizarre operations such as mutation of the active stack can throw a wrench in Reticulated's type system.

Miscellaneous In addition to these sources of dynamism, values can also be forced to be typed `Dyn` due to more mundane limitations of the Reticulated type system. Functions with variable arity and those that take arbitrary keyword arguments have their input annotations ignored and their parameter type set to `Arb`; designing a type specification for functions that captures all of such behavior is an engineering challenge and is important for practical use of Reticulated. Reticulated also does not yet typecheck metaclasses or function decorators; values that use them are simply typed `Dyn`.

3.2.7 Efficiency

None of our approaches make any attempt to speed up typed code. The mechanisms that they use to perform runtime checks slow it down, and we cannot prevent Python from performing its standard runtime checks even when the type system has proven them unnecessary. As a result, Reticulated programs perform far worse than their unchecked Python implementations — the slowSHA test suite, for example, has on the order of a 10x slowdown under transient compared to normal Python. We have not expended much effort in optimizing Reticulated's performance, so we are very confident this could be much improved, but never beyond baseline Python. However, we can compare them to each other meaningfully. The test suite included with the statistics library took 2.7 seconds to run under the guarded semantics and 5.5 under monotonic, but only 1.6 with transient. The slowSHA library test suite took 10.4 seconds with guarded and 13.6 with monotonic, but only 5.1 with transient. These figures are from an 8-core Intel i7 at 2.8 GHz, and they exclude time spent in the typechecker and cast inserter. Due to issues with object identity, CherryPy was unable to run without error at all when using the guarded semantics, as we discuss in Section 3.2.5, so we do not have a timing figure that program.

This result indicates that the simple, local checks made by the transient semantics are, taken together, more efficient than the proxy system used by guarded and (for functions only) monotonic, and that the special getters and setters used by monotonic are expensive, even if they do not cause casts to occur. This may simply be because these features rely on Python's slow reflection and introspection features, as we discuss in Section 4; in any case, this enhances the advantages of the transient design.

4. Implementation

Reticulated is implemented as a source-to-source translator, and thus the dynamic semantics for our typed Python dialects are themselves Python programs. In this section, we discuss the implementation of the cast operations for each of our designs.

4.1 Guarded

The proxies of the guarded system are implemented as instances of a `Proxy` class which is defined at the cast site where the proxy is installed. We override the `__getattribute__` property of the `Proxy` class, which controls attribute access on Python objects, to instead retrieve attributes from the casted value and then cast them. We similarly override `__setattr__` and `__delattr__`, which control attribute update and deletion respectively. This is sufficient for most proxy use cases. However, classes themselves can be casted, and therefore must be proxied in the guarded system. Moreover, when a class' constructor is called via a proxy, the result should be an object value that obeys the types of its class, and which therefore itself needs to be a proxy — even though there was no "original," unproxied object in the first place.

Python is a rich enough language to allow us to accomplish this. In Python, classes are themselves instances of metaclasses, and so a class proxy is a class which is an instance of a `Proxy` metaclass. When an object is constructed from a class proxy, the following code is executed:

```
1   underlying = object.__new__(cls)
2   proxy = retic_make_proxy(underlying, src.instance(),
        trg.instance())
3   proxy.__init__()
```

In this code, `cls` is the underlying class being proxied. The call to `object.__new__` creates an "empty" instance of the underlying class, *without* the class's initialization method being invoked. Then an object proxy is installed on this empty instance; the source and target types of the cast that this proxy implements are the types of instances of the class proxy's source and target types. Finally, the initialization method is invoked on the object proxy.

Another source of complexity in this system comes from the `eval` function. In Python, `eval` is dynamically scoped; if a string being `eval`ed contains a variable, `eval` will look for it in the scope of its caller. This poses a problem when `eval` is coerced to a static type and is wrapped by a function proxy, because then `eval` only has access to the variables in the *proxy's* scope, not the variables in the proxy's caller's scope. To handle this edge case, proxies check at their call site if the function they are proxying is the eval function. If it is, the proxy retrieves its caller's local variables using stack introspection, and run `eval` in that context.

Another challenge comes from the fact that some values in Python are not Python code. In the CPython implementation (the only major Python implementation to support Python 3 at the time of writing, and therefore our main target), many core features are implemented in C, not self-hosted by Python itself. However, C

```
def retic_cast(val, src, trg, msg):
  return retic_check(val, trg, msg)
def retic_check(val, trg, msg)
  assert has_type(val, trg, msg)
  return val
```

Figure 8. Casts and checks in the transient system.

code does not respect the indirection that our proxies use, and so when it tries to access members from the proxy, it might find nothing. We developed a partial solution for this by removing proxies before values are passed into C whenever possible. The C code is then free to modify the value without respecting the cast's type guarantees, but if a read occurs later, the reinstalled proxy will detect any incorrectly-typed values and report an error.

4.2 Transient

While the guarded semantics is challenging to implement in Python, the implementation of the transient semantics is very straightforward. Figure 8 shows the essence of the transient runtime system; the actual implementation is a bit more complicated in order provide more informative cast errors than simple assertion failures, and to perform the specialized object casts described in Section 3.2.3. The `has_type` function, used by transient's implementation, is shown in Figure 5. The guarded and monotonic approaches depend on the reflection capabilities of the host language. The transient design, on the other hand, is almost embarrassingly simple, and depends on only the ability to check Python types at runtime and check the existence of object fields; it could likely be ported to nearly any language that supports these operations.

4.3 Monotonic

Our implementation of the monotonic semantics uses techniques similar to those used by the guarded design, in that it modifies the `__getattribute__` and `__setattr__` methods of objects. In this case, however, we overwrite these methods on the class of the casted object itself, not a proxy. We also install specialized getters and setters for when the result of the read is statically known to be of a precise static type, and for when it needs to be upcast to some specific imprecise type. For example, an access of the form `o.x` will be rewritten by the cast inserter to `o.__staticgetattr__('x')` if the type of `o` provides `x` with a fully-static type. This is a call to the *original* getter for `o`, which performs no casts or checks. On the other hand, if `o.x` instead has a fully or partially dynamic type `T`, the typechecker will rewrite it as `o.__getattr_attype__('x', T)`, which reads the field and then casts the result to `T`.

Monotonic is not totally free from guarded-style proxies. Although this approach does not use object proxies, it does use proxies to implement function casts. We did not encounter any issues in our case studies that we traced to monotonic function proxies.

4.4 Discussion

When retrofitting gradual typing to an existing language, our concerns are somewhat different than they might be if gradual typing was integral to the design of the underlying language. Type safety, for example, is not of the same paramount importance, because Python is already safe — an error in the guarded cast implementation may cause unexpected behavior, but it will not lead to a segmentation fault any more than it would have if the program was run without types. On the other hand, by developing our system in this manner we cannot constrain the behavior of Python programs except through our casts. We cannot simply outlaw behavior that poses a challenge to our system when that behavior occurs in untyped code, even if it interacts with typed portions of a program.

We previously spent time implementing gradual typing directly in the Jython implementation of Python. We were able to achieve some success in increasing efficiency of typed programs by compiling them into typed Java bytecode. However, the amount of effort it took to make significant modifications to the system made it difficult to use as a platform for prototyping different designs for casts. By taking the source-to-source translation approach with Reticulated, we are able to rapidly prototype cast semantics by simply writing modules that define `cast` and (optionally) `check` functions, and we are able to use Python's rich reflection and introspection libraries rather than needing to write lower-level code to achieve the same goals. Reticulated does not depend on the details of any particular implementation of Python, and when alternative Python implementations like PyPy and Jython support Python 3, we expect that little modification of Reticulated will be necessary to support them. In the meantime, we are backporting Reticulated to Python 2.7, which is supported by several alternative implementations.

5. Related Work

Much work has gone into integrating static and dynamic typing in the same system. Early work on the subject includes the `dynamic` of Abadi et al. [2] and the quasi-static typing of Thatte [31], as well as Strongtalk [7]. Cecil [8] and the Bigloo variant of Scheme [24] allow optional type annotations, but do not encode runtime checks between static and dynamic code. Gray et al. [13] extended Java with contracts to allow interoperability with Scheme.

Siek and Taha [26] introduced the gradual typing approach to melding static and dynamic type systems. Other work has extended gradual typing to support many different language features (such as that of Herman et al. [15], Siek and Taha [27], Wolff et al. [36], Takikawa et al. [30]). Other research has adapted the notion of *blame tracking* from the work on contracts by Findler and Felleisen [11] (Siek and Wadler [28] and Wadler and Findler [35]). Rastogi et al. [21] develop an approach for inferring the types of locals variables, parameters, and return types.

Industrial language designers have taken gradual typing to heart, with several languages, including C# [5], TypeScript, Dart [12], and Hack [10] offering support for gradual typing or similar features. Work on adding gradual typing to existing languages has also appeared within the academic space, such as the work of Tobin-Hochstadt and Felleisen [33], Allende et al. [4] and Ren et al. [22]. Bloom et al. [6] developed the Thorn language, which supports the *like types* of Wrigstad et al. [37], enabling the mixing of static and dynamic code without loss of compiler optimization.

Politz et al. [20] formalized the semantics of Python by developing a core calculus for the language and a "desugarer" which converts Python programs to the calculus. Lerner et al. [18] developed TeJaS, a framework for retrofitting static type systems onto JavaScript to experiment with static semantics design choices. The authors use it to develop a type system similar to that of TypeScript.

Cutsem and Miller [9] discussed some of the challenges in designing proxies that correctly emulate the target object's behavior, including the need to preserve identity checks.

6. Conclusions

In this paper we presented Reticulated Python, a lightweight framework for designing and experimenting with gradually typed dialects of Python. With this system we develop two designs for runtime casts. The guarded system implements the design of Siek and Taha [27], while the novel transient cast semantics does not require proxies but instead uses additional use-site checking to preserve static type guarantees, and the monotonic approach causes object values to become monotonically more precise such that they are more or equally statically typed than all references to them.

We evaluated these systems by adapting several third party Python programs to use them. By annotating and running these programs using Reticulated, we determined that, while our type system is mostly sufficient, much more Python code would be able to be typed statically were we to include generics in our type system. We also discovered that with the right design choices, including supporting static types for literals and using local, dataflow-based type inference, we were able to cause significant portions of programs to be entirely statically typed, and therefore suitable for compiler optimization. We made several alterations to our type system based on how it interacted with existing Python patterns, such as modifying casts that check the existence of object members to be catchable by source-program try/except blocks. We also encountered other situations where we had to modify the original program to interact well with our type system, including by replacing object and type identity checks with other operations and preventing lists from varying between different element types. We discovered several potential bugs in our target programs by running them with Reticulated and we found that the proxies used by the guarded system cause serious problems because they do not preserve object identity.

Acknowledgments We thank Steev Young and Matteo Cimini for their help in testing the Reticulated Python implementation.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] M. Abadi, L. Cardelli, B. C. Pierce, and G. D. Plotkin. Dynamic typing in a statically-typed language. In *POPL*, pages 213–227, 1989.

[3] A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. Blame for all. In *POPL '11: ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 1–14, 2011.

[4] E. Allende, O. Callaú, J. Fabry, E. Tanter, and M. Denker. Gradual typing for smalltalk. *Science of Computer Programming*, August 2013.

[5] G. Bierman, E. Meijer, and M. Torgersen. Adding dynamic types to C#. In *European Conference on Object-Oriented Programming*, ECOOP'10. Springer-Verlag, 2010.

[6] B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strniša, J. Vitek, and T. Wrigstad. Thorn: Robust, concurrent, extensible scripting on the jvm. *SIGPLAN Not.*, 44(10):117–136, Oct. 2009. ISSN 0362-1340.

[7] G. Bracha and D. Griswold. Strongtalk: typechecking Smalltalk in a production environment. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 215–230, New York, NY, USA, 1993. ACM Press. ISBN 0-89791-587-9.

[8] C. Chambers and the Cecil Group. The Cecil language: Specification and rationale. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, 2004.

[9] T. V. Cutsem and M. S. Miller. Trustworthy proxies: virtualizing objects with invariants. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 154–178, Berlin, Heidelberg, 2013. Springer-Verlag. .

[10] Facebook. Hack, 2011. URL `http://hacklang.org`.

[11] R. B. Findler and M. Felleisen. Contracts for higher-order functions. Technical Report NU-CCS-02-05, Northeastern University, 2002.

[12] Google. Dart: structured web apps, 2011. URL `http://dartlang.org`.

[13] K. E. Gray, R. B. Findler, and M. Flatt. Fine-grained interoperability through mirrors and contracts. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 231–245, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-031-0.

[14] J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, pages 93–104, 2006.

[15] D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. In *Trends in Functional Prog. (TFP)*, page XXVIII, April 2007.

[16] L. Ina and A. Igarashi. Gradual typing for generics. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '11, 2011.

[17] G. A. Kildall. A unified approach to global program optimization. In *POPL '73: ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 194–206. ACM Press, 1973.

[18] B. S. Lerner, J. G. Politz, A. Guha, and S. Krishnamurthi. Tejas: Retrofitting type systems for JavaScript. In *Symposium on Dynamic Languages*, DLS '13, pages 1–16, 2013.

[19] E. Meijer and P. Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

[20] J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python: The full monty. In *ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 217–232, 2013.

[21] A. Rastogi, A. Chaudhuri, and B. Hosmer. The ins and outs of gradual type inference. In *ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL '12, 2012.

[22] B. M. Ren, J. Toman, T. S. Strickland, and J. S. Foster. The ruby type checker. In *SAC'13 (OOPS)*, 2013.

[23] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do: A large-scale study of the use of eval in javascript applications. In *European Conference on Object-Oriented Programming*, ECOOP'11, pages 52–78, Berlin, Heidelberg, 2011.

[24] M. Serrano. *Bigloo: a practical Scheme compiler*. Inria-Rocquencourt, April 2002.

[25] J. G. Siek. Is typescript gradually typed? part 2, Oct. 2013. URL `http://siek.blogspot.com/2012/10/is-typescript-gradually-typed-part-2.html`.

[26] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, September 2006.

[27] J. G. Siek and W. Taha. Gradual typing for objects. In *ECOOP 2007*, volume 4609 of *LCNS*, pages 2–27. Springer Verlag, August 2007.

[28] J. G. Siek and P. Wadler. Threesomes, with and without blame. In *Symposium on Principles of Programming Languages*, 2010.

[29] J. G. Siek, R. Garcia, and W. Taha. Exploring the design space of higher-order casts. In *European Symposium on Programming*, March 2009.

[30] A. Takikawa, T. S. Strickland, C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Gradual typing for first-class classes. In *Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 793–810, 2012.

[31] S. Thatte. Quasi-static typing. In *POPL 1990*, pages 367–381, New York, NY, USA, 1990. ACM Press. ISBN 0-89791-343-4.

[32] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *OOPSLA'06 Companion*, pages 964–974, NY, 2006. ACM.

[33] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *POPL '08: the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008.

[34] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 117–128, 2010.

[35] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *European Symposium on Programming*, 2009.

[36] R. Wolff, R. Garcia, E. Tanter, and J. Aldrich. Gradual typestate. In *European Conference on Object-Oriented Programming*, ECOOP'11. Springer-Verlag, 2011.

[37] T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *Symposium on Principles of Programming Languages*, 2010.