Design and Evaluation of Gradual Typing for Python

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

Jim Baker
Rackspace Inc.
jbaker@zyasoft.com

Abstract

Combining static and dynamic typing within the same language, i.e. *gradual typing*, offers clear benefits to programmers. Programs get the benefits of dynamic typing — rapid prototyping, simple heterogeneous data structures, and reflection — while enforcing static type guarantees where desired. However, many open questions remain regarding the semantics of gradually typed languages, especially concerning the integration of gradual typing with existing languages.

In this paper we present Reticulated Python, a lightweight 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, a source-to-source translator that inserts casts, and Python libraries that implement the casts. Using Reticulated Python, we evaluate a gradual type system that features structurally-typed objects and type inference for local variables. We evaluate two dynamic semantics for casts: one based on Siek and Taha (2007) and a new design that does not require proxy objects. In our evaluations, we study the effect of gradual typing on two third-party Python programs: the CherryPy web framework and a library of statistical functions.

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

Keywords gradual typing, blame, case study, python, proxy

1. Introduction

Static and dynamic typing are well-suited to different programming tasks. Static typing excels at documenting and enforcing constraints, enabling IDE support such as autocompletion, 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 trade offs, 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, interest in combining static and dynamic typing within a single language and type system has increased over the last decade. Siek and Taha [14] introduced the term *gradual typing* to describe such merged systems and showed how to add gradual typing to a language with first-class functions. Numerous authors have integrated gradual typing with other language features (such as Herman et al. [6], Siek and Taha [15], Wolff et al. [22], Takikawa et al. [18]). Other research has adapted the notion of *blame tracking* [5] to gradual typing, allowing for useful information to be reported when type casts fail (such work includes that of Wadler and Findler [21] and Siek and Wadler [16]).

In this paper, we present *Reticulated Python*,[1] a framework for developing gradual typing for the popular dynamic language Python. 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.

Rather than being a modification of any existing implementation of Python, 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 with casts inserted. These casts are themselves implemented as calls into a Python library we developed to perform the casts. 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 cast-inserted at load time.

[Copyright notice will appear here once 'preprint' option is removed.]

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

In addition to serving as a practical tool to enable programming in a gradually typed language, Reticulated serves as a test bed for experimenting with design choices for gradually typed languages. To evaluate different designs, we performed a pair of case studies in which we applied Reticulated to existing codebases. In these experiments, we took two existing Python projects — one from the world of scientific computing, and one from the world of web development — and annotated them with types, and then ran them using Reticulated. These experiments let us detect several bugs extant in these projects. They 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. The process of providing static types to our case studies gave us insights as to what Python design patterns cannot be statically typed under this type system, such as certain uses of runtime dispatch. In some cases, we determined that relatively minor extensions of our type system would greatly expand the number of values which may be typable, such as the introduction of generics.

Our creation of Python libraries to implement cast operations enables straightforward implementation of alternate casting semantics. In the case studies we compare and evaluate two distinct cast libraries with Reticulated, one that uses the proxy-based approach of Herman et al. [6] and Siek and Taha [15], but optimized with threesomes [16], and a new approach to cast checking that involves ubiquitous lightweight checks. We refer to the former design as the *guarded* approach to casts, and the latter as the *transient* semantics. The guarded system is more complicated to implement and does not preserve object identity, which we found to be a problem in practice (see Section 4.2.4). The transient approach is straightforward to implement and preserves object identity, but it does not perform blame tracking and therefore is less helpful when debugging cast errors.

Contributions

- We develop Reticulated Python, an exogenous framework for the design of gradual typing in Python 3. This system allows Python programs to be annotated with static types, performs typechecking, and inserts casts, which take the form of calls into runtime Python library that implements the cast semantics. We give an overview of this system's design in Section 2.
- In Section 3, we discuss two different dynamic semantics, one of which is based on proxies and the other, new approach, that is based on use-site checks. We also discuss Reticulated's static semantics.
- In Section 4, we carry out two case studies of applying gradual typing to third-party Python programs, validating some design choices — such as the use of dataflow-based type inference — while indicating which extensions to our system would be most valuable, e.g. generics.
- In Section 5, we evaluate the overall approach that we took for implementing gradual typing in an existing language, and consider valuable lessons for other designers of gradually-typed languages.

2. Gradual typing for Python

We present an example of programming in Reticulated and discuss the alternative designs that we explore in this paper.

Gradual type systems put programmers in 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. When values flow from dynamically typed positions to static ones, or vice versa, casts are inserted by the compiler to ensure that static type of an expression must be *consistent* with the type of the value that results from evaluating the expression [14]. For example, the types `List(Int)` and `List(Dyn)` are consistent, but `List(String)` and `List(Int)` are not.

One of the primary benefits of gradual typing over dynamic typing is that it helps to 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 that returns 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`. 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`. The following shows the `moment` function with annotated parameter and return types.

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

With this change, the runtime error will point to call to `moment` on line 2, where an implicit cast from `Dyn` to `List(Float)` occurred.

The dynamic semantics of casts is more subtle than one might think. Casts on base values like integers and booleans are indeed 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.

Herman et al. [6] prevent such type unsoundness by installing *proxies* on mutable values when they are casted from one type to another. These proxies check that values written to the proxied value have a type consistent with the source of the cast, and that values read from it have a type consistent with the target of the proxy's cast. This approach, which we refer to as the *guarded* semantics, prevents type errors from occurring in the body of `moment`, because when elements are read from `inlist`, a proxy will ensure that they are typed `Float` or else raise a cast error. In such a situation, the cast error originates from within the `moment` function, but thanks to a technique known as *blame tracking* Findler and Felleisen [5], the error message points back to the site of the implicit cast on line 2. We discuss the guarded semantics in more detail in Section 3.2.1.

The guarded design has a disadvantage in that it does not preserve object identity — a proxied value does not have the same identity as its underlying value, and this can cause programs that depend on object identity to break. We present an alternative design, called *transient casts*, that checks that the value is consistent with the target type of the cast, but does not install a proxy to prevent later violations due to mutation. Instead, it inserts dynamic checks ubuiquitiously throughout the code to make sure that values continue to behave according to the static types in the code. In our running example, the transient casts approach would trigger a cast error on line 2 when it checks whether the list `lst` is a list of numbers and instead finds strings. We discuss the transient semantics in more detail in Section 3.2.2.

3. The Reticulated Python designs

We use the Reticulated Python framework to explore two language designs, guarded and transient, which share the same static semantics (Section 3.1) but which have different dynamic semantics (Sections 3.2.1 and 3.2.2).

3.1 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 declaration of a function of type $T_1, T_2 \to T_3$ could be

```
def typed_fun(param1: T1, param2: T2)-> T3:
  ...
```

In Python 3, annotations may be any Python expression, which is evaluated at the function definition site but otherwise ignored. However, in Reticulated, we restrict the expressions which can appear in annotations to the type expressions shown in Figure 1 and to aliases for object and class types. The absence of an annotation implies that the parameter or return type is `Dyn`.

labels	ℓ		type variables X
base types	B	::=	`Int` | `String` | `Float` | `Bool`
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

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

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

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

Figure 2. Class and object type rules.

3.1.1 Features of the type system

The compile-time type system is primarily based on the first-order object calculus of Abadi and Cardelli [1], with several important differences, as we will discuss in this section.

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; that is, a function defined by `def f(x,y)` can be called by explicitly setting `x` and `y` to desired values like `f(y=42, x=25)`. To typecheck calls like this, we include the names of parameters in our function types by use of 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`, as shown by our subtyping rules in Appendix A.1. Function types with `Arb` (for arbitrary) parameters can be called on any sort of argument.

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 C:
2   def f(self:C, x:Int)->C:
3     return self
4 foo = C()
```

Here the variable `C` has the type

```
1 Class(C){f:Function(Named(self:Dyn, x:Int),C)}
```

The occurrence of `C` inside of the class's structural type is a type variable bound to the type of an *instance* of `C`. Figure 2 shows Reticulated's introduction rule for class types in the special case where the class being typed 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 types may be invoked as though they were functions, as shown in Figure 2. This rule shows the case where no special initialization method is defined on the class; if `__init__` does exist, Reticulated checks that the arguments of the construction call site match its parameters. In the above example, `foo` will have type

```
Object(X) { f : Function(Named(x:Int),X) }
```

This type is derived from the class type of `C` by removing the self parameter of all the function in the class' type. The type parameter `X` represents the self type. We use the *bind* metafunction, defined in the Appendix, section A.2, 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 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){...}`.

In Python, programmers can dynamically add properties to objects at will. In recognition of this fact, 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. In this sense, Reticulated's object types are similar to the bounded dynamic types of Ina and Igarashi [7], except that their approach is appropriate for nominal type system while our open objects are appropriate for structural typing.

Lists, tuples, sets, and dictionaries have special, builtin types, but reflecting Python semantics, they also have corresponding object types which are used when they are the receiver of a method call.

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 infered 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.

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

Figure 3. Alias substitution

```
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 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

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

and similar for $\hat{\texttt{B}}$.

Literal types 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, the result should be observationally identical to what it would be if it was run directly. 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'` which is a valid Python program. So, to maintain full 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 4 we discuss the practical effect of this choice.

3.1.2 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.

3.1.3 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 [8] 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, defined in Appendix A.3, is over a lattice which is slightly different from the subtyping lattice. In our subtyping lattice, `Dyn` is not top, for reasons described in Siek and Taha [14], but it is top in this inference lattice. Also, the inference lattice is not the naïve subtyping lattice of Siek and Wadler [16] because function parameters are contravariant. Instead, our lattice for type inference can be defined as the lattice such that casts from types lower on their lattice to types higher on it are never blamed under a cast system that follows the **D** semantics of Siek et al. [17].

3.2 Dynamic semantics

Using the Reticulated framework, we explore two different dynamic semantics for casts. The intent behind both 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 [13] and described by Siek and Taha [14]. However, we prefer to detect when there are inconsistencies between the static types specified by the program and the runtime values. In this section, we discuss the design of our two dynamic semantics for casts, which perform this checking in very different ways.

3.2.1 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` 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 [16]. 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, whether explicit or implicit (such as the `[]` lookup operator implicitly calling the `__getitem__` method on its left operand), actually 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.

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, we arrange for the proxy to be a subclass of the runtime Python class of the underlying value,[2] instance checks generally return the same result with the bare value as they do with the proxied value. In our case studies in Section 4, we evaluate the consequences of this issue in real-world code.

3.2.2 The transient dynamic semantics

In the transient semantics, a cast checks that the value has a type consisent 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:
```

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

Call sites	A check ensures that the result of the call matches the callee's expected return type
Attribute reads	A check ensures that the resulting value from the attribute read matches the expected type of the attribute
Subscription	A check ensures that the result of the subscription (the operation that includes indexing or slicing lists) has the expected type
Function definitions	At the entry to a function body, each parameter is checked to ensure its value matches the parameter's type
For loops	The iteration variable of the for loop is checked at the beginning of each iteration of the loop

Figure 4. Transient check insertion sites.

```
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. The complete list of sites where checks are inserted is in Figure 4. 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

```
def has_type(val, ty):
  if isinstance(ty, Dyn):
    return True
  elif isinstance(ty, Int):
    return isinstance(val, int)
  elif isinstance(ty, List):
    return isinstance(val, list) and all(has_type
        (elt, ty.element_type) for elt in val)
  elif isinstance(ty, Object):
    return all(hasattr(val, member) and has_type(
        getattr(val, member), ty.member_type(
        member)) for member in ty.members)
  elif isinstance(ty, Function):
    return callable(val)
  elif ...
    ...
```

Figure 5. Definition for the `has_type` function.

not detect or prevent type-altering mutations from occurring later — instead, such mutations are prevented from being observable by use-site checks.

One might worry that this design results in greater runtime overhead than the guarded approach. However, the checks inserted by the transient system are also happening in the guarded approach, except that they are invoked syntactically in the transient design rather than being performed by proxies. This results in transient actually outperforming guarded, as we demonstrate in Section 4.2.6.

3.2.3 Runtime overhead in statically-typed code

Readers may have noted that both the guarded and transient semantics impose runtime overhead on statically typed code, which is rather unfortunate. Avoiding such overhead is difficult to achieve in an exogenous implementation, but if the underlying language offers hooks that enable optimizations based on type information, Reticulated's libraries could easily utilize such hooks. This is an especially promising approach in the PyPy implementation of Python, where Reticulated could enable optimizations in the JIT, and in Jython, where Reticulated could cause typed code to execute entirely in Java bytecode rather than being translated back into Python. As a test-bed for gradual typing in Python, Reticulated does not currently use such approaches, but as it matures into a practical tool for developers using the lessons discussed in Section 4 below, such designs will become increasingly relevant.

4. 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 both the guarded and transient semantics. We categorized the code that we were unable to type statically and identified several additional features that would let more code be typed, including generics and a static-single assignment transformation. 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.

4.1 Case study targets

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

4.1.1 Statistics libraries

We chose the Harvard neuroscientist Gary Strangman's statistics library[3] as a subject of our Reticulated case study as an example of the kind of focused numerical programs that are common within the community of scientific programmers who use Python. The `stats.py` module contains a wide variety of statistics functions and the `pstat.py` sister module provides some common list manipulation functions.

To prepare them for use with Reticulated, we translated these libraries from Python 2 to Python 3. We also 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.

4.1.2 CherryPy

CherryPy[4] 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 4.2.5. In these situations, we annotated the private functions that are called by the API functions instead.

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

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

4.1.3 The Python Standard Library

Unlike the previous two cases, we did not modify the Python Standard Library. However, the previous test cases rely heavily on the Python Standard Library — especially CherryPy — and it is mostly written in Python, so when standard libraries are imported by code running under Reticulated, they are themselves typechecked and cast-inserted. This typechecking process is essentially a no-op when Reticulated is set for maximum backwards-compatibility, but meaningful casts are inserted into these system libraries under other modes of operation. In both cases, running the Python Standard Library through Reticulated helped us understand the interaction of typed Reticulated code with untyped code.

4.2 Results

By running our case studies in Reticulated under both the guarded and transient semantics, we made a number of discoveries about both our system and the targets of the studies themselves. We 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. We also discovered two classes of bugs that exist in the target programs which were revealed by the use of Reticulated.

4.2.1 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. It is clear that the developer of this library understood this behavior, but this falling-off behavior still poses a problem if, for example, `betacf` is called by some other function. The `None` value that may be returned 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. 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 3.1.1, 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` can be used in the same fashion (including instances of subclasses of `Node`), then they expect that the call

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

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.[5]

4.2.2 The Reticulated type system in practice

The choice of whether or not to include typed literals, as discussed in Section 3.1.1, 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)
```

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

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 in Appendix A.1 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)
```

One approach to solving this problem without needing to modify the source is to let the type inferencer reason about the list element assignment at line 6. However, this only helps us if the variable containing the list never escapes to another function that could modify it without the type inferencer knowing. A more general solution is to weaken our type inference algorithm to only infer types shallowly — it would be safe to infer that `deviations` has the type `List(Dyn)`, and then insert casts when it is read from or written to. The same result could also be achieved by simply disabling typed literals. However, this solution results in many more casts being inserted, and therefore, a more difficult to optimize program. For this reason, we believe that the best solution is to allow the choice between typed and untyped literals to be a local one, made on a per-function or per-module basis by the programmer. We also note that this case, and several similar ones in the statistics library, are the only situations in our case studies — even including the Python Standard Library — where type or cast errors arise from the use of typed literals.

Typed literals allow math to be typed. When typed literals are enabled, and any necessary edits to the source are made as above, many of the calculation-heavy statistics functions that we studied 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,[6] 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 testbed 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.

4.2.3 Cast insertion with structural objects

We discovered one issue that arose in both of our runtime systems because our rules for cast insertion 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 trys 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. 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.

[6] 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.

4.2.4 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 also be a proxy. In that case, the identity check at line 4 will never return `True`, if the underlying value is `object.__init__`. The `TypeError` exception will then not be raised, and errors may occur later in the program's execution. We identified 19 sites in the 70-module CherryPy library where object identity is checked against a value other than the unit value `None`. (Checks against `None` are not an issue, because `None` is never proxied.) Most of these cases could be rewritten to avoid the use of identity checking in favor of equality checking or type instance checking; in the above case, 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.

4.2.5 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. The 23 list manipulation functions in `pstat.py` take as parameters values of type `List(Dyn)` or `List(List(Dyn))` and several of these functions also return values of one of those types. Of these, eight are written in a style so suitable to the use of generics that if we introduced generics to our type system, no modification would be necessary (other than altering type annotations) to alter them to use generics. As an example of a function that would work well with generics, consider:

```
1 def unique(inlist:List(Dyn))->List(Dyn):
2   uniques = []
3   for item in inlist:
4     if item not in uniques:
5       uniques.append(item)
6   return uniques
```

This function from `pstat.py` returns a list containing the elements from the parameter `inlist` with all duplicates removed. In Reticulated the only appropriate type for this function is `Function([List(Dyn)], List(Dyn))` because the type of the contents of the list is not statically

known. In fact, the type of the list contents is irrelevant: the only operation that involves the items of `inlist` directly is checking if they are already in the output list `uniques`; this is implicitly a call to their `__eq__` method, which is supported by all Python values. In this sense, the dynamic type here fulfills the same purpose that a type variable would serve in a statically typed language with generics.

A type system that supports generics would let the `unique` function be given a fully static type. This would be beneficial to callers of `unique`, such as the `mode` function from `stats.py`:

```
1 def mode(inlist:List(Float))->(Int,List(Float)):
2   scores = pstat.unique(inlist)
3   scores.sort()
4   ...
```

Since `unique` returns `List(Dyn)`, type inference determines that `scores` should also have the type `List(Dyn)`, and thus the type of elements of `scores` will not be statically known. This causes many casts to be inserted between `Float` and `Dyn` in the remainder of the function, when elements of `scores` are used. If `scores` was known to be of type `List(Float)`, much of the body of `mode` would only involve values known to be `Floats`, which in turn would mean that it could be optimized with a type-aware compiler.

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. No matter what kind of value `config` initially is, it is eventually converted into a value which may be 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 [20]. 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.

This pattern is used for similar purposes as method overloading often is in statically typed languages, and so another approach we could take is to implement type-based method overloading in Reticulated. By letting the programmer write separate functions for when `config` is a dictionary, a string, and a file object, we could accurately typecheck calls to these functions. Python does not support function overloading, but it would be straightforward for Reticulated to translate multiple, differently typed function definitions into distinctly named ones.

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 `Ints` and `Floats` as we discuss in Section 4.2.2. 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. This example could be given a more precise type if we introduced sum types into the Reticulated type system. However, in the above example and others like it, sum types would not assist programmers in knowing which keys are expected to have values of a specific type.

`eval` and other villains Finally there are those cases where truly and 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. [12]:

```
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.

It can still be meaningfully typed, however, and providing types to similar cases can help mitigate the danger inherent in calling `eval` on externally sourced values. By annotating `dm` with a return type of `List(Dyn)`, the result of this `eval` will be casted to ensure that it at least is a list, and if it is not an appropriate error can be thrown. Still, not all uses of `eval` can be so gracefully handled, and other bizarre operations such as mutation of the active stack by use of the `inspect.getouterframes()` function can throw a wrench in Reticulated's type system.

Miscellaneous In addition to these significant 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 such behavior is something of an engineering challenge, important for practical use of Reticulated, but unlikely to reveal further interesting properties about gradual typing in Python. Reticulated also does not yet typecheck metaclasses or function decorators; values that use them are simply typed `Dyn`.

4.2.6 Efficiency

Neither the guarded system nor the transient approach make any attempt to speed up typed code, and the mechanisms that they use to perform runtime checks slow it down. Comparing the runtime efficiency of either approach to an unchecked implementation is, therefore, not especially meaningful. However, we can compare them to each other. The test suite included with the statistics library took 3.816 seconds to run under the guarded semantics, but only 1.492 with transient. This figure is over an average of 5 runs on an Intel Core2 Duo E8400 CPU at 3.00GHz, and it excludes 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 4.2.4, 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. This may simply be because proxies rely on Python's slow reflection and introspection features, as we discuss in Section 5; in any case, this enhances the advantages of transient over guarded.

5. Implementation

Reticulated is an exogenously defined implementation of gradual typing, and thus the dynamic semantics for our typed Python dialects are themselves Python programs. In this section, we discuss the implementation of the guarded and transient cast and check operations.

5.1 Guarded

Guarded's proxies 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, only then is the initialization method 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 `eval` in that context.

Another challenge comes from the fact that some functions callable from Python are written in other languages. 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 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, for circumstances when the C function is itself wrapped by a proxy. In this situation, any proxy on the function's argu-

```
1 def retic_cast(val, src, trg, msg):
2   return retic_check(val, trg, msg)
3 def retic_check(val, trg, msg)
4   assert has_type(val, trg, msg)
5   return val
```

Figure 6. Casts and checks in the transient system.

ments are removed and the underlying value is passed in. The C code is then free to modify the value as it desires, 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.

5.2 Transient

While the guarded semantics is challenging to implement in Python, the implementation of the transient semantics is very straightforward. Figure 6 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 4.2.3. The `has_type` function, used by transient's implementation, is shown in Figure 5. The guarded approach might not even be possible in other dynamic languages, and it almost certainly would have to be implemented with different techniques than the ones we use. 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 for presence or absence of object fields; it could likely be ported to nearly any language that supports these operations.

5.3 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 do not have the option of simply outlawing 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 exogenous 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 language code to achieve the same goals. As a result, 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 some alternative implementations.)

6. Related work

Siek and Taha [14] introduced the gradual typing approach, and other work has extended this approach to support many different language features (such as that of Herman et al. [6], Siek and Taha [15], Wolff et al. [22], Takikawa et al. [18]). Other research has adapted the notion of *blame tracking* from the work on contracts by Findler and Felleisen [5] (Siek and Wadler [16] and Wadler and Findler [21]). Rastogi et al. [11] develop an approach for inferring types of locals as well as the parameters and return types of functions.

Industrial language designers have taken note of the benefits offered by gradual typing, with several Microsoft languages, including C# [3] and TypeScript, offering features similar to those provided by gradual typing. Work on adding gradual typing to existing languages has also previously appeared within the academic space, such as the work of Tobin-Hochstadt and Felleisen [19] and Allende et al. [2]. Bloom et al. [4] developed the Thorn language, which supports the *like types* of Wrigstad et al. [23] which enables the mixing of static and dynamic code without loss of compiler optimizations.

Politz et al. [10] 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. [9] developed TeJaS, a framework for retrofitting static type systems onto JavaScript in order to experiment with static semantics design choices. The authors use it to develop a type system similar to that of TypeScript.

7. 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 [15], while the novel transient cast semantics does not require proxies but instead uses additional use-site checking to preserve static type guarantees.

We evaluated these systems by adapting two third party Python libraries to use them: Gary Strangman's statistics packages and the CherryPy web application framework. 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 discovered that the guarded design's failure to preserve object identity results in serious problems in practice.

References

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

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

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

[4] 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.

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

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

[7] L. Ina and A. Igarashi. Gradual typing for generics. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, 2011.

[8] G. A. Kildall. A unified approach to global program optimization. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM Press, 1973.

[9] B. S. Lerner, J. G. Politz, A. Guha, and S. Krishnamurthi. Tejas: Retrofitting type systems for javascript. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 1–16, New York, NY, USA, 2013. ACM.

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

[11] A. Rastogi, A. Chaudhuri, and B. Hosmer. The ins and outs of gradual type inference. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, 2012.

[12] 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 *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 52–78, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22654-0.

[13] 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`.

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

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

[16] J. G. Siek and P. Wadler. Threesomes, with and without blame. In *Proceedings for the 1st workshop on Script to Program Evolution*, STOP '09, pages 34–46, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-543-7. . URL `http://doi.acm.org/10.1145/1570506.1570511`.

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

[18] 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.

[19] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 2008. ACM.

[20] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 117–128, New York, NY, USA, 2010. ACM.

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

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

[23] 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.

A. Appendix: relations

A.1 Subtyping

$\boxed{\vdash T <: T}$ *Subtyping*

$$\frac{}{\vdash B <: B} \qquad \frac{}{\vdash \texttt{Dyn} <: \texttt{Dyn}}$$

$$\frac{}{\vdash \texttt{Int} <: \texttt{Float}} \qquad \frac{}{\vdash \texttt{List}(T_2) <: \texttt{List}(T_2)}$$

$$\frac{\vdash_P P_1 <: P_2 \qquad \vdash T_1 <: T_2}{\vdash \texttt{Function}(P_1, T_1) <: \texttt{Function}(P_2, T_2)}$$

$$\frac{i \geq j \qquad \forall j.\ T_{1j}[X_1/X_2] = T_{2j}}{\vdash \texttt{Object}(X_1)\{\overline{\ell_i{:}T_{1i}}\} <: \texttt{Object}(X_2)\{\overline{\ell_j{:}T_{2j}}\}}$$

$$\frac{i \geq j}{\vdash \texttt{Class}(X_1)\{\overline{\ell_i{:}T_i}\} <: \texttt{Class}(X_2)\{\overline{\ell_j{:}T_j}\}}$$

$$\frac{\begin{array}{c}\exists k \leq i.\ \ell_k = \texttt{\_\_call\_\_} \\ \vdash \mathit{bind}(T_k) <: \texttt{Function}(P, T)\end{array}}{\vdash \texttt{Object}(X_1)\{\overline{\ell_i{:}T_i}\} <: \texttt{Function}(P, T)}$$

$$\frac{\begin{array}{c}\exists k \leq i.\ \ell_k = \texttt{\_\_init\_\_} \\ \vdash \mathit{bind}(T_k) <: \texttt{Function}(P, T)\end{array}}{\vdash \texttt{Class}(X_1)\{\overline{\ell_i{:}T_i}\} <: \texttt{Function}(P, T)}$$

$\boxed{\vdash_P P <: P}$ *Parameter subtyping*

$$\frac{}{\vdash_P \texttt{Arb} <: \texttt{Arb}}$$

$$\frac{\forall i.\ \vdash T_{2i} <: T_{1i}}{\vdash_P \texttt{Named}(\overline{\ell_i{:}T_{1i}}) <: \texttt{Pos}(\overline{T_{2i}})}$$

$$\frac{\forall i.\ \vdash T_{2i} <: T_{1i}}{\vdash_P \texttt{Pos}(\overline{T_{1i}}) <: \texttt{Pos}(\overline{T_{2i}})}$$

$$\frac{\forall i.\ \vdash T_{2i} <: T_{1i}}{\vdash_P \texttt{Named}(\overline{\ell_i{:}T_{1i}}) <: \texttt{Named}(\overline{\ell_i{:}T_{2i}})}$$

A.2 Method binding

$\boxed{\mathit{bind}(T)}$

$$\begin{array}{rcl}\mathit{bind}(\texttt{Function}(P, T)) & = & \texttt{Function}(\mathit{bindp}(P), T) \\ \mathit{bind}(T) & = & T\end{array}$$

$\boxed{\mathit{bindp}(P)}$

$$\begin{array}{rcl}\mathit{bindp}(\texttt{Arb}) & = & \texttt{Arb} \\ \mathit{bindp}(\texttt{Pos}(T_k^{0 \leq k \leq n})) & = & \\ & & \texttt{Pos}(T_k^{1 \leq k \leq n}) \\ & & \text{if } n \geq 1 \\ \mathit{bindp}(\texttt{Named}(\ell_k{:}T_k^{0 \leq k \leq n})) & = & \\ & & \texttt{Named}(\ell_k{:}T_k^{1 \leq k \leq n}) \\ & & \text{if } n \geq 1\end{array}$$

A.3 Relations for type inference

$\boxed{T \sqcup T = T}$

$$\frac{}{B \sqcup B = B} \qquad \frac{B_1 \neq B_2}{B_1 \sqcup B_2 = \texttt{Dyn}}$$

$$\frac{}{\texttt{Dyn} \sqcup T = \texttt{Dyn}} \qquad \frac{}{T \sqcup \texttt{Dyn} = \texttt{Dyn}}$$

$$\frac{\begin{array}{c}X_3 \text{ fresh} \\ \forall k < \mathit{min}(i, j).\ T_{1k}[X_1/X_3] \sqcup T_{2k}[X_2/X_3] = T_{3k} \\ T = \texttt{Object}(X_3)\{\overline{\ell_k{:}T_{3k}}\}\end{array}}{\texttt{Object}(X_1)\{\overline{\ell_i{:}T_{1i}}\} \sqcup \texttt{Object}(X_2)\{\overline{\ell_j{:}T_{2j}}\} = T}$$

$$\frac{\begin{array}{c}X_3 \text{ fresh} \\ \forall k < \mathit{min}(i, j).\ T_{1k}[X_1/X_3] \sqcup T_{2k}[X_2/X_3] = T_{3k} \\ T = \texttt{Class}(X_3)\{\overline{\ell_k{:}T_{3k}}\}\end{array}}{\texttt{Class}(X_1)\{\overline{\ell_i{:}T_{1i}}\} \sqcup \texttt{Class}(X_2)\{\overline{\ell_j{:}T_{2j}}\} = T}$$

$$\frac{T = \texttt{Function}(P_1 \sqcup_P P_2, T_1 \sqcup T_2)}{\texttt{Function}(P_1, T_1) \sqcup \texttt{Function}(P_2, T_2) = T}$$

$$\frac{\mathit{head}(T_1) \neq \mathit{head}(T_2)}{T_1 \sqcup T_2 = \texttt{Dyn}}$$

$\boxed{\mathit{head}(T)}$

$$\begin{array}{rcl}\mathit{head}(B) & = & B \\ \mathit{head}(\texttt{Dyn}) & = & \texttt{Dyn} \\ \mathit{head}(\texttt{Function}(P, T)) & = & \texttt{Function} \\ \mathit{head}(\texttt{Object}(X)\{\ldots\}) & = & \texttt{Object} \\ \mathit{head}(\texttt{Class}(X)\{\ldots\}) & = & \texttt{Class}\end{array}$$

$\boxed{P \sqcup_P P = P}$

$$\frac{P_1 \not\approx P_2}{P_1 \sqcup_P P_2 = \texttt{Arb}}$$

$$\frac{}{P \sqcup_P \texttt{Arb} = \texttt{Arb}}$$

$$\frac{}{\texttt{Arb} \sqcup_P P = \texttt{Arb}}$$

$$\frac{\overline{T_{1i} \sqcap T_{2i} = T_{3i}} \qquad P = \texttt{Named}(\overline{\ell_i : T_{3i}})}{\texttt{Named}(\overline{\ell_i{:}T_{1i}}) \sqcup_P \texttt{Named}(\overline{\ell_i{:}T_{2i}}) = P}$$

$$\frac{\overline{T_{1i} \sqcap T_{2i} = T_{3i}} \qquad P = \texttt{Pos}(\overline{T_{3i}})}{\texttt{Pos}(\overline{T_{1i}}) \sqcup_P \texttt{Pos}(\overline{T_{2i}}) = P}$$

$$\frac{\overline{T_{1i} \sqcap T_{2i} = T_{3i}} \qquad P = \texttt{Pos}(\overline{T_{3i}})}{\texttt{Named}(\overline{\ell_i{:}T_{1i}}) \sqcup_P \texttt{Pos}(\overline{T_{2i}}) = P}$$

$$\frac{\overline{T_{1i} \sqcap T_{2i} = T_{3i}} \qquad P = \texttt{Pos}(\overline{T_{3i}})}{\texttt{Pos}(\overline{T_{1i}}) \sqcup_P \texttt{Named}(\overline{\ell_i{:}T_{2i}}) = P}$$

$\boxed{P \approx P}$ *Parameter length and name matching*

$$\frac{}{\texttt{Arb} \approx \texttt{Arb}}$$

$$\frac{i = j}{\texttt{Pos}(\overline{T_i}) \approx \texttt{Pos}(\overline{T_j})}$$

$$\frac{i = j}{\texttt{Named}(\overline{\ell_i{:}T_i}) \approx \texttt{Pos}(\overline{T_j})}$$

$$\frac{i = j}{\texttt{Pos}(\overline{T_i}) \approx \texttt{Named}(\overline{\ell_j{:}T_j})}$$

$$\frac{i = j \qquad \overline{\ell_i = \ell_j}}{\texttt{Named}(\overline{\ell_i{:}T_i}) \approx \texttt{Named}(\overline{\ell_j{:}T_j})}$$

$\boxed{T \sqcap T = T}$

$$\frac{}{B \sqcap B = B} \qquad \frac{B_1 \neq B_2}{B_1 \sqcap B_2 = \texttt{Dyn}}$$

$$\frac{}{\texttt{Dyn} \sqcap T = T} \qquad \frac{}{T \sqcup \texttt{Dyn} = T}$$

$$\frac{\begin{array}{c}X_3 \text{ fresh} \\ \forall k < max(i,j).\ T_{1k}[X_1/X_3] \sqcap T_{2k}[X_2/X_3] = T_{3k} \\ T = \texttt{Object}(X_3)\{\overline{\ell_k{:}T_{3k}}\}\end{array}}{\texttt{Object}(X_1)\{\overline{\ell_i{:}T_{1i}}\} \sqcap \texttt{Object}(X_2)\{\overline{\ell_j{:}T_{2j}}\} = T}$$

$$\frac{\begin{array}{c}X_3 \text{ fresh} \\ \forall k < max(i,j).\ T_{1k}[X_1/X_3] \sqcap T_{2k}[X_2/X_3] = T_{3k} \\ T = \texttt{Class}(X_3)\{\overline{\ell_k{:}T_{3k}}\}\end{array}}{\texttt{Class}(X_1)\{\overline{\ell_i{:}T_{1i}}\} \sqcap \texttt{Class}(X_2)\{\overline{\ell_j{:}T_{2j}}\} = T}$$

$$\frac{T = \texttt{Function}(P_1 \sqcap_P P_2, T_1 \sqcap T_2)}{\texttt{Function}(P_1, T_1) \sqcap \texttt{Function}(P_2, T_2) = T}$$

$$\frac{head(T_1) \neq head(T_2)}{T_1 \sqcup T_2 = \texttt{Dyn}}$$

$\boxed{P \sqcap_P P = P}$

$$\frac{P_1 \not\approx P_2}{P_1 \sqcap_P P_2 = \texttt{Arb}}$$

$$\frac{}{P \sqcap_P \texttt{Arb} = P} \qquad \frac{}{\texttt{Arb} \sqcap_P P = P}$$

$$\frac{\overline{T_{1i} \sqcup T_{2i} = T_{3i}} \qquad P = \texttt{Named}(\overline{\ell_i : T_{3i}})}{\texttt{Named}(\overline{\ell_i{:}T_{1i}}) \sqcap_P \texttt{Named}(\overline{\ell_i{:}T_{2i}}) = P}$$

$$\frac{\overline{T_{1i} \sqcup T_{2i} = T_{3i}} \qquad P = \texttt{Pos}(\overline{T_{3i}})}{\texttt{Pos}(\overline{T_{1i}}) \sqcap_P \texttt{Pos}(\overline{T_{2i}}) = P}$$

$$\frac{\overline{T_{1i} \sqcup T_{2i} = T_{3i}} \qquad P = \texttt{Pos}(\overline{T_{3i}})}{\texttt{Named}(\overline{\ell_i{:}T_{1i}}) \sqcap_P \texttt{Pos}(\overline{T_{2i}}) = P}$$

$$\frac{\overline{T_{1i} \sqcup T_{2i} = T_{3i}} \qquad P = \texttt{Pos}(\overline{T_{3i}})}{\texttt{Pos}(\overline{T_{1i}}) \sqcap_P \texttt{Named}(\overline{\ell_i{:}T_{2i}}) = P}$$