# Gradual Typing in an Open World

**Michael M. Vitousek and Jeremy G. Siek**

**Indiana University Bloomington**
{mvitouse,jsiek}@indiana.edu

## Abstract

Gradual typing combines static and dynamic typing in the same language, offering the benefits of both to programmers. Static typing provides error detection and strong guarantees while dynamic typing enables rapid prototyping and flexible programming idioms. For programmers to take full advantage of a gradual type system, however, they must be able to trust their type annotations, and so runtime checks must be performed at boundaries between static and dynamic code to ensure that static types are respected. Higher order and mutable values cannot be completely checked at these boundaries, and so additional checks must be performed at their use sites. Traditionally, this has been achieved by installing proxies on these values to moderate the flow of data between static and dynamic code, but this can cause problems if the language supports comparison of object identity or foreign function interfaces.

Reticulated Python is a gradually typed variant of Python 3 implemented via a source-to-source translator. It implements a proxy-free design named *transient casts*. This paper presents a formal semantics for transient casts and shows that not only are they sound, but they work in an open-world setting in which the Reticulated translator has only been applied to some of the program; the rest is untranslated Python. We formalize this *open world soundness* property and use Coq to prove that it holds for Anthill Python, a calculus that models Reticulated Python.

## 1 Introduction

Gradual typing [29, 37] enables the safe interaction of statically typed and dynamically typed code. In recent years, gradual typing has been of interest not only to the research community [24, 35, 34, 33, 25, 4, 3] but also to industry, as numerous new languages have arrived on the scene with features inspired by gradual typing, including Hack [12] and TypeScript [22].

Gradually typed languages use the dynamic type $\star$ and a *consistency relation* on types to govern how statically typed and untyped code interacts: types are consistent if they are equal up to the presence of $\star$. The consistency relation replaces type equality in the static type system. For example, at a function call the type of an argument is required to be consistent with the parameter type of the function, not equal to it. To prevent uncaught runtime type errors, additional checks must be performed at runtime. These checks are typically achieved by inserting casts as part of a type-directed, source-to-source translation.

Designing semantics for these casts in the presence of higher-order values and mutation is a significant research endeavor (Herman et al. [20], Siek et al. [31, 33], etc.). The traditional approach [14, 29, 20] installs runtime guards (i.e. wrappers or proxies) on values, which moderate between differently typed regions of code. Here we refer to this as the guarded approach. Alternatively, the monotonic approach of Siek et al. [33] avoids proxies by using runtime type information to lock down objects as they pass through casts. With the monotonic approach, some programs that would execute without error under guarded instead produce a runtime cast error. The related design of Swamy et al. [34] requires further restrictions. For example, a function of the type $\star \to \star$ cannot be cast to $\mathsf{bool} \to \mathsf{bool}$.

*Reticulated Python*[1] is a platform for experimenting with different dialects of gradually

[1] Named for the largestc species of snake in the Python family.

typed Python [40]. One design implemented in Reticulated Python is the transient approach, which uses pervasive use-site checks in lieu of guarding objects with proxies. This alternative avoids a serious issue for guarded: proxies are not pointer-identical to the underlying proxied object, a fact that results in significant and challenging-to-debug problems in practice. Vitousek et al. [40] use Reticulated to examine the effects of proxies on gradually typed Python programs and reproduced the problems observed by Van Cutsem and Miller [39].

Although significant research has focused on leveraging gradual typing to achieve performance improvements [20, 42, 33], efficiency is not an aim of the transient design. Instead, transient is an error-detecting technique, similar to the checked mode of Dart [17] (but sound). It can be enabled during development and debugging and disabled for release.

### Transient semantics modeled in Anthill Python

In this paper we present a complete formal semantics for *Anthill Python*,[2] a gradually typed calculus which models Reticulated Python and which uses the transient semantics. Anthill Python supports much of the complexity of the Python object system, including bound methods, multiple inheritance, and mutable class fields and methods.

Just as Reticulated programs are translated into Python and then executed, the dynamic semantics of Anthill is defined by translation into an untyped language, *μPython*. We define the dynamic semantics of μPython and a translation from Anthill to μPython, inserting checks according to the transient semantics. While Vitousek et al. [40] informally described transient, Anthill and μPython provide a mathematical description of the design.

### Open world gradual typing

In addition, in this paper we show that the distinction between guarded and transient affects more than object identity; it affects whether or not the language supports sound interaction with foreign code in an "open world".

Like many gradually typed languages, Reticulated Python translates input programs to a target language with explicit casts, in this case standard Python 3. When using guarded, the translated code cannot interact with untranslated Python code (in this paper referred to as "plain Python") without losing the usual type soundness guarantees of gradual typing [29]: plain Python programs do not include the explicit casts necessary to maintain the invariants expected by translated Reticulated programs. The transient approach, on the other hand, does not make any assumptions about its clients. Instead, typed code performs all required checks internally.

The guarded approach poses additional problems in the context of Python. CPython, the reference implementation of Python, is implemented in C, and many built-in functions are defined as C functions. Furthermore, Python supports C and C++ extension modules — modules callable from Python that are implemented in C. This code does not respect the abstractions used by guarded to ensure soundness and can mutate Python data structures as raw memory, possibly violating invariants specified by Reticulated's type system. The transient semantics can soundly coexist with such foreign functions because it performs all required checks within typed code at use sites.

In this paper we discuss a formal property called *open world soundness* which states that, if a program is translated from a gradually typed surface language into an untyped target, it can be embedded inside arbitrary untyped code without ever being the source of an uncaught

[2] Named for the smallest species of snake in the Python family.

type error. For gradually typed languages, open world soundness is a stronger property than traditional type safety. We prove that open world soundness holds for the transient design as modeled by Anthill Python and its translation to µPython. Our proof is machine-checked by the Coq proof assistant and can be found at `https://arxiv.org/src/1610.08476v1`. It relies on an auxiliary static type system for µPython that captures invariants about programs translated from Anthill. This type system discriminates between translated code and plain µPython code by tagging all reducible expressions with labels indicating the origin of the expression. The type system places strong requirements on the translated expressions, while untranslated code is unrestricted. We prove that the translation from Anthill to µPython is type-preserving.

Open world soundness also allows programmers to benefit from gradual typing without using its features themselves. Using transient, a Reticulated library writer can use static types in their code, including on API boundaries. If this library is then translated to regular Python, any Python programmer who uses the library will benefit from the improved error detection resulting from Reticulated's static types without having to use — or even know about — Reticulated itself.

### Contributions

- We define the property of *open world gradual typing* and evaluate the guarded and transient designs with respect to it (Section 3).
- We provide a formal semantics for the transient design in the form of Anthill Python, a calculus that models Reticulated Python (Section 4). This semantics is defined by translation to an untyped calculus, µPython, that models Python.
- We prove that Anthill exhibits *open world soundness*, an extension of type soundness which states that the translation from Anthill to µPython produces code that doesn't go wrong even when interoperating with plain µPython code (Section 5). The full proof in Coq is at `https://arxiv.org/src/1610.08476v1`.

Section 2 discusses background and related work, and Section 6 concludes.

## 2 Background and Related Work

In this section we discuss related work and review the prior work on Reticulated Python [40].

### 2.1 Gradual typing

Researchers and language designers have been interested in mixing static and dynamic typing in the same language for quite some time [10, 1, 9]. Gradual typing, developed by independently by Siek and Taha [29] and Flanagan [16], was preceded quasi-static typing of Thatte [36], the Java extensions of Gray et al. [18], Bigloo, [27] and Cecil [11]. Gradual typing is distinguished by its use of the consistency relation (originally defined for nominally typed languages by Anderson and Drossopoulou [6]) to govern where typed and untyped code can flow into each other, and where runtime checks need to be performed to ensure safety at runtime. See Siek et al. [32] for a detailed discussion of the core principles that gradual typing aims to satisfy.

Allende et al. [5] develop an alternate cast insertion scheme for Gradualtalk [4] that facilitates interaction between typed libraries and untyped clients without requiring client recompilation. This approach, called "execution semantics" uses callee-installed proxies on

function arguments. While similar to our semantics, this approach is still vulnerable to the problems with proxies that we address with the transient approach (see Section 2.3).

Another relevant alternative design is the *like-types* approach [8, 42]. This approach avoids proxies by splitting static type annotations into *concrete types* (whose inhabitants are never proxied, and which cannot flow into dynamically typed code) and like types, which can freely interact with dynamic code. This approach was used in designing a sound variant of TypeScript called StrongScript, which obeys open-world soundness [26]. However, because of the incompatibility of concrete and like types, it is not straightforward to evolve a program in StrongScript from dynamic to static, as is frequently desirable. As a result, the *gradual guarantee* of Siek et al. [32] does not hold for these designs.

Typed Racket [38, 35] includes first-class classes and supports open-world interaction between Typed Racked modules and untyped Racket code. In that work, the authors are aided by Racket's module semantics, which allows functions exported from Typed Racket into untyped code to be wrapped with a contract monitor that ensures that interaction between typed and untyped code is sound even though untyped clients are unaware of the static type system. These features allow Typed Racket to have open world soundness with a guarded approach. Python does not have the same capabilities for controlling module exports and faces the problems with proxies and foreign functions explained above.

In recent years, gradual typing, or ideas related to it, has become popular among industrial language designers, with C# [7] adding a dynamic type and Typescript [22], Dart [17], and Hack [12] offering static typechecking of optional type annotations. Academic language designers have also retrofitted gradual typing to existing languages, such as Racket [38], Smalltalk [4], Ruby [25], and Python [40].

## 2.2 Reticulated Python and the guarded cast semantics

Reticulated Python implements several designs for gradual typing, including guarded, the traditional design for gradual typing [29, 20]. In this design, casts are inserted at the locations of implicit conversions. For example, consider the following statically typed `filter` function, which expects a parameter of type `Callable([int], bool)` (the syntax for function type int → bool).

```
1 def filter(fn:Callable([int],bool), lst:List(int))->List(int):
2   nlst = []
3   for elt in lst:
4     if fn(elt):
5       nlst.append(elt)
6   return nlst
7 filter(lambda x: x % 2 == 0, [1, 2, 3, 4])
```

The function created on line 7 has no type annotations, so it has type `Callable([Dyn], Dyn)`, where `Dyn` is the dynamic type. It is casted from `Callable([Dyn], Dyn)` to `Callable([int], bool)` at runtime, because of a cast inserted at line 7 by the translation from Reticulated Python to Python.

Casts on first-order values are checked immediately but casts on functions and objects are not. With guarded, function casts install a wrapper on the function which, when called, casts the input, calls the underlying function, and casts the result [15]. Casts on objects return a proxy — a new object that casts fields and methods during reads and writes.

To see how object proxies are used to preserve type soundness, consider the example in Figure 1, where the object `Foo()` is expected to have type `{'bar':int}` by the typed `f` function. The `f` function passes the object to the untyped `g` function which mutates the `bar`

field to contain a string. Upon return from `g`, `f` accesses the `bar` field, expecting an `int`. The code highlighted in yellow indicates the locations where proxies are installed. The object bound to the parameter `x` in `g` is therefore not the object passed to `g` by `f` at line 6, but is instead a proxy to that object. When `g` attempts to write a string to that proxy at line 4, the proxy casts it to `int`, the type that `y.bar` is expected to have in `f`, which triggers a cast error.

## 2.3 Guarded faces practical problems

The guarded approach is well studied, with many optimizations for space and time efficiency and additional features such as blame tracking [20, 28, 3, 41], but a significant problem with guarded is that a proxy is not identical to its proxied object. This issue leads to unexpected behavior when pointer equality checks (using Python's `is` operator) are performed. Van Cutsem and Miller [39] also encountered and discussed this issue. Vitousek et al. [40] found these issues to be a significant problem in Reticulated Python in guarded, preventing many programs from running as expected.

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

**Figure 1** Soundness via proxies

A related issue is that Python programs can inspect the class of a value using the `type` function, and the class of a proxy is a proxy class (rather than the class of the proxied object). Programs that use reflection over the types of values can execute incorrectly when the `type` function returns a proxy class rather than the expected class.

## 2.4 The transient cast semantics

To solve this issue, Vitousek et al. [40] implemented an alternative, called the transient approach, that does not use proxies. (Vitousek et al. [40] report on the implementation of transient but they do not define a formal semantics.) The transient design uses *checks* — lightweight queries about the current state of a value — rather than proxy-building casts. Checks are inserted into programs at function calls and object reads. This approach transfers the responsibility for checking object reads and function return values from the object or function itself to the context that is performing the read or call. From an implementation perspective, this moves the checking from the runtime system (e.g. casts performed by object proxies) into the translated programs' code.

Figure 2 illustrates this difference. Figure 2a shows a program in which a function `f` takes and uses a typed object parameter. The argument to the function comes from untyped code, and thus any sound semantics for gradually typing will require some runtime checks when running this program. Text in yellow indicates locations where checks must occur because program values are entering into a context where they are expected to have a certain type — either by being passed in as arguments, dereferenced from mutable state, or returned from a function call.

Figure 2b shows the same program after explicit runtime checks have been inserted. Checks are performed pervasively, including in statically typed code. The write at line 2 does not need a check because `obj` is not proxied and any reads of `x` will ensure that it has the right type for its context. The highlighted parameters on line 1 indicate that the function checks that `z` and `obj` have their expected types at its entry point, translated as

(a) Pre-insertion:

```
1 def f(z:int, obj:{'x':int,
            'meth':Callable([int],int)})
            ->int:
2   obj.x = 42
3   method = obj.meth
4   result = method(z)
5   return result
6
7 def g(y):
8   f(42, y)
```

(b) Post-insertion:

```
1 def f(z, obj):
    check(z, int)
    check(obj,{'x':int,
            'meth':Callable([int],int)})
2   obj.x = 42
3   method = check(obj.meth,
        Callable([int],int))
4   result = check(method(x), int)
5   return result
6
7 def g(y);
8   return check(f(42, y), int)
```

**Figure 2** Check insertion by the transient semantics

the explicit checks at the beginning of the function body in Figure 2b. The check on line 3 verifies that `obj.meth` has a function type, and line 4 checks that the result of the function call is an `int`. This transfer of responsibility from clients (and the proxies they install) to typed code allows this design to succeed without needing proxies.

To see how transient preserves soundness in the presence of mutation, consider the example in Figure 3, which contains the same program as Figure 1. Again, sites where transient checks are made are highlighted in yellow. While guarded would install a proxy on `y` at line 6 to ensure that `y.bar` remains an `int`, the transient semantics passes `y` into `g` directly. The strong update at line 4 proceeds, but the fact that a string has been written to `y.bar` will not be observable within `f`. This is because at the dereference of `y.bar` at line 7, a transient check will attempt to verify that `y.bar` is an `int` (the type expected in that context). When this check fails, a runtime check error is reported.

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

**Figure 3** Soundness via transient

The lack of proxies and the reliance on use-site checks in the transient approach means that many of the problems with guarded are solved immediately. Object identity behaves normally because objects and their types are always the same as in plain Python, and every check that does not fail simply returns the checked value. On the other hand, checks are pervasively installed even within typed code, and in places where they would not be needed under guarded. Thus transient is primarily useful as a debugging tool, rather than something to be deployed in production code.

Other approaches also tackle the problem of object identity: Keil and Thiemann [21] present a solution based on the idea of making proxies transparent with respect to identity and type tests. TypeScript [22], Dart [17], and other languages that compile to JavaScript without any runtime checks are trivially free of proxies but their type annotations are not enforced at runtime. Dart also offers a *checked mode*, wherein function arguments are checked at runtime against optional type annotations, similar to the transient approach, but these checks do not cover all cases and uncaught runtime type errors are still possible. However,

transient solves an additional diffuculty that guarded faces: interoperability with untyped or foreign code in an open world, which we discuss in the next section.

# 3 Open World Gradual Typing

In this section, we discuss the property of open world gradual typing, show that guarded is an obstacle to it, and discuss the solution provided by transient. While this discussion is specific to Python, many issues are broadly applicable to other dynamically typed languages such as JavaScript and Clojure.

## 3.1 Guarded is unsuitable for an open world

While guarded is sound when all parts of a program have been seen by the typechecker and translator, it is not always desirable or even possible for that to be the case. We examine two situations where interactions between Reticulated programs and plain Python or foreign functions are troublesome.

### Plain Python clients cannot soundly use Reticulated programs with Guarded

Reticulated Python typechecks programs and translates them into Python 3, and once they have undergone this translation, they can be executed by a Python interpreter. As such programs are valid Python programs, we would like to use them in combination with plain Python programs from different sources. Unfortunately, when using Reticulated with guarded, this is not type safe and can cause programs to behave unexpectedly.

It is important here to distinguish plain Python programs from *untyped Reticulated code*. The latter is Reticulated Python code that does not contain type annotations, but that is translated into Python by Reticulated. This translation is *not* an identity transformation, even on untyped Reticulated code: if the untyped code makes calls into typed code, it must ensure that the values it passes into typed code are cast to their expected type.

Figure 4 shows interaction between translated Reticulated and plain Python, which is problematic under the guarded semantics. Two modules, **utils** and **fastexp**, can communicate freely with each other even though **fastexp** has no type annotations, because Reticulated translates them (the dotted $\rightsquigarrow$ arrows at the bottom of Figure 4) both into Python programs $\lfloor$**utils**$\rfloor$ and $\lfloor$**fastexp**$\rfloor$ with explicit casts. Because $\lfloor$**utils**$\rfloor$ is a standard Python module, the plain Python module **interactive** may attempt to use it. The **interactive** module passes in a string (the result of `input`) where the type system expected an int, but this error is not detected because, in the guarded approach, it was the responsibility of **interactive** to provide the correct type by checking its arguments at the call site (or, in the case of higher order values, installing a proxy around them). The result is a confusing Python error:

```
1  File "utils.py", line 2, in is_odd
2    return x % 2
3 TypeError: not all arguments converted during string formatting
```

This is the same error that would occur in plain Python without static types, but now it may be more difficult to debug: the programmer may not realize that they cannot trust the type annotations.

This issue prevents programmers from writing Reticulated programs with types on API boundaries and distributing them as servers for unknown, plain Python clients. Because typed API boundaries are an important use case for gradual typing, this is a significant

Module **utils** (Reticulated Python):

```
1 def is_odd(x:int)->int:
2   return x % 2
3 def intappend(lst:List(int),
      x)->List(int)
4   list.append(lst, x)
5   return lst
```

Module **fastexp** (Reticulated Python):

```
1 import utils
2 def fastexp(x, n):
3   ... utils.is_odd(n) ...
```

Module **interactive** (Plain Python):

```
1 import utils
2 while True:
3   print('Enter a number')
4   inp = input()
5   print(utils.is_odd(inp))
```

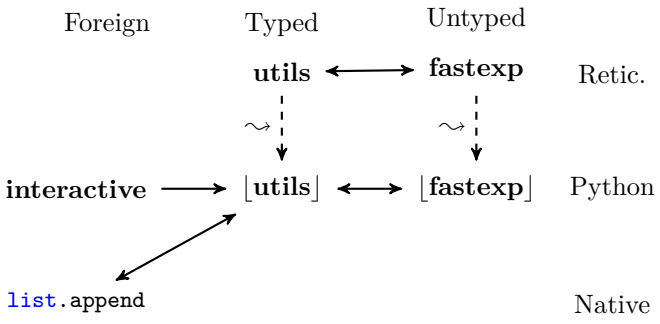


**Figure 4** Reticulated and plain Python interaction

problem that reaches well beyond Python. This can be partially solved by providing untyped API functions for plain Python clients and casting the client's calls to static types internally (an option discussed in the context of Gradualtalk by Allende et al. [5]). However, this alone cannot guarantee safety in Reticulated's context, because Python modules cannot selectively export information to clients — clients can still access the typed functions.

### Foreign functions cannot operate on proxies

Another issue arises because many of the built-in functions used by Python programs are not defined in Python itself, but rather in compiled binaries. For example, in CPython the `append` method for lists is defined in C code which is not accessible for introspection by Reticulated and which, more importantly, does not use the same machinery for attribute access as regular Python code.

This is illustrated by the `intappend` function in **utils** in Figure 4. Suppose `intappend` is called from Reticulated code as follows:

```
1 import utils
2 utils.intappend([1,2,3],4)
```

Under the guarded semantics, at the call to `utils.isodd` at line 2, the list `[1,2,3]` is wrapped in a proxy, which should prevent code from writing non-`int` values to it. However, when it is passed to `list.append` as the receiver,[3] `list.append` is not operating on the list `[1,2,3]`,

[3] For those unfamiliar with Python, `list.append` is an unbound method of the `list` class, and its first argument is set as the receiver of the call. By calling `list.append(x, ...)` instead of `x.append(...)`,

but instead on a proxy to it.

Like any Python value, the proxy has to be an instance of some class. If the proxy's class was unrelated to `list`, then a list's proxy would not appear to be a list instance to the commonly-used `isinstance` function, causing many unexpected errors. Thus in Reticulated's implementation of guarded, this proxy will be a value whose class is a dynamically generated subclass of the `list` class. However, this choice means that the proxy itself contains its own list in the heap, distinct from the list it is proxying. As a proxy, its methods are replaced with reflective calls to the underlying, proxied list, so this internal list is inaccessible to Python code. The native code that implements `list.append`, however, directly operates on the memory in the heap containing the list data of the proxy, bypassing the proxy's methods. As a result, the proxy cannot intercept and forward changes to the underlying list, and thus `list.append` mutates the (empty and otherwise inaccessible) list contained within the proxy itself and leaves the proxied list untouched. The overall result is that, quite surprisingly, the result of the call at line 2 is `[1,2,3]` — the list that was passed in, unmodified.

As a result, Reticulated programs under guarded can behave unpredictably and in ways contrary to their expected semantics. This problem is difficult to diagnose and its scope is difficult to determine.

## 3.2 The transient cast semantics is sound in an open world

Though Vitousek et al. [40] discuss transient only as a technique to solve issues with type identity, it also enables open world soundness. `Transient` transfers the responsibility for checking object reads and function return values from the object or function itself to the context that is performing the read or call. Therefore, if plain Python code passes ill-typed values into translated code, checks within the typed module will detect a type mismatch and report an error rather than causing a confusing error or not reporting the error at all. In Figure 4, when `[1,2,3]` is an argument of `list.append`, it actually is the list `[1,2,3]` being passed in, and native code can operate on it exactly as it normally would. Even if native code mutates the list in an ill-typed way, it will be detected as soon as a dereference occurrs within typed Reticulated code.

Furthermore, because transient avoids proxies, C code sees only the Python values that were expected. When the **interactive** module calls **utils**, instead of a confusing error referring to string formatting, the result will be a cast error saying that a string has been passed in instead of an int.

# 4 Formalizing Transient with Anthill Python

The above section illustrates some of the complexities of integrating typed and untyped code in Reticulated Python, and the ways that Reticulated code can be used in an open world. This section formalizes these properties, and the transient design, with two calculi: a gradually typed surface language named Anthill Python that models Reticulated and an a calculus named µPython that models plain Python.

## 4.1 Design considerations of the type system

In this section we discuss some of the distinct features of Python that Anthill Python models and how these features inform the design of Anthill's static type system.

we avoid going through the proxy on `x`.

### Python classes

Like many dynamic languages, Python's classes are first-class values which can be passed between program components like any other kind of data. Therefore a static type system for Python must give types to classes just as it does to objects and functions. In Python, classes may also directly contain attributes which can be accessed and mutated, and they can be extended with new fields. Such mutations to a class value can affect all its instances. For example, one can add methods to a class after its creation and then call those methods on any instance of the class. Classes are also themselves callable — calling a class is the Python syntax for object construction.

These observations indicate that classes share elimination forms with both objects and functions. To typecheck call sites and attribute reads or writes without undue restrictions, the class types of Reticulated (and Anthill) are subtypes of both object and function types.

### Object construction

Python objects come into existence with their object-local fields uninitialized. The user-defined code in the `__init__` method must instanciate them by mutating the receiver of the method (represented as the first parameter of the method, called `self` by convention). The `__init__` method is not restricted to only initializing fields, though — it can contain arbitrary Python code, including code that allows the partially initialized receiver to escape into other functions. Consider the following code in which an object is specified to have `x` and `y` fields of type `int` by the `@fields` decorator. The result is a failed check at line 8.

```
1 @fields({'x':int, 'y':int})
2 class Point2D:
3   def __init__(self:Point2D, x:int, y:int):
4     print(magnitude(self))
5     self.x = x
6     self.y = y
7 def magnitude(pt:Point2D)->float:
8   return math.sqrt(pt.x ** 2 + pt.y ** 2)
```

The problem here is that the annotation on line 3 that `self` is a `Point2D` is not true: at that point, `self` does not obey the type that a `Point2D` is expected to have, lacking the fields `x` and `y`. This problem is well known in other object-oriented languages such as Java, and several solutions have been proposed [23, 13]. These approaches are substantially more complex than the rest of the Reticulated type system, and by virtue of being gradually typed, Reticulated can rely on dynamic type checks to preserve soundness. Reticulated therefore requires that the receiver parameter of a constructor, i.e. `self`, is untyped. This behavior matches Python, but it means that no program with objects can ever be truly fully statically typed — every object goes through a dynamic initialization phase, after which it can be injected to a static type by inserting a check at object instantiation sites.

## 4.2 Anthill Python

Anthill Python is a gradually typed calculus which models Reticulated Python. The syntax of Anthill Python is defined in Figure 5.

$$
\begin{array}{lcl}
t & ::= & x \mid n \mid t(\overline{t}) \mid t.\ell \mid t.\ell = t \mid \texttt{let}\ x = t\ \texttt{in}\ t \mid \texttt{class}(\overline{t})\texttt{:}(\!|q, \Delta, \Delta|\!)\ \texttt{init=}c\texttt{;}\ \overline{\ell =^M m}\texttt{;}\ \overline{\ell =^F t} \mid \\
 & & \lambda \overline{x\texttt{:}A} \to A.\ t \\
m & ::= & \varsigma\ x_s, \overline{x\texttt{:}A} \to A.\ t \\
c & ::= & \sigma\ x_s, \overline{x\texttt{:}A}.\ t \\
q & ::= & \Diamond \mid \blacklozenge \\
A & ::= & \star \mid \mathsf{int} \mid \overline{A} \to A \mid \mathsf{class}\ (\!|q, \Delta, \Delta, \overline{A}|\!) \mid \mathsf{object}\ (\!|q, \Delta|\!) \\
\Delta & ::= & \overline{\ell : A}
\end{array}
$$

**Figure 5** Syntax for Anthill Python. (Overlines indicate sequencing.)

### 4.2.1 Types

Anthill Python's types include the dynamic type $\star$, integers, $n$-ary function types, and object and class types.

Object types $\mathsf{object}\ (\!|q, \Delta|\!)$ contain two parts: an *openness* descriptor $q$ and an *attribute type* $\Delta$. Attribute types, written are partial maps from attribute names to types, and represent the structural type of the object. An openness descriptor can be either $\Diamond$ for "open" or $\blacklozenge$ for "closed". Open objects support implicit width downcasts, while getting or setting fields not present in $\Delta$ causes a static type error in a closed class; see Section 4.4.1.

Class types contain an openness descriptor and two attribute types. The first attribute type contains the public interface for the class itself while the second represents the instance fields present in every instance of the class, but not in the class itself. The instance fields are used in the type system to derive object types from class types and the presence of instance fields is checked at runtime after object construction. Class types also record the parameter types of their constructors, so that object instantiations can be type checked.

### 4.2.2 Terms

Terms in Anthill Python include variables, numbers, applications, attribute access and update, and $n$-ary functions with parameter and return type annotations. Methods and fields are read from with $t.\ell$ and written to with $t.\ell = t$, where $\ell$ is the name of an attribute; there is no syntactic differentiation between accessing fields and methods.

Anonymous class definitions are also terms, and contain a number of components. The openness descriptor $q$ and the attribute types $\Delta_1$ and $\Delta_2$ specify the overall static type of the class. The class' constructor is given by $\texttt{init} = c$. In Python (and Reticulated), object constructors are simply methods named `__init__`, but to simplify the semantics, Anthill constructors $c$ are a special syntactic form. Constructors lack a return type, because they are only invoked during object construction, and they perform initialization by mutating a fresh, empty object passed to the constructor.

The superclasses of a class definition are specified by the $\overline{t}$ immediately after the `class` keyword. Fields and their initializing terms are defined by $\overline{\ell =^F t}$. Methods $m$ (specified in class definitions by $\overline{\ell =^M m}$) are similar to functions except that they always have an explicit receiver $x_s$ as their first parameter. The $F$ and $M$ superscripts help to syntactically distinguish between fields and methods. No type annotation is given for the receiver of a method; it is treated as having the type of an instance of the enclosing class, as is typical for object-oriented calculi [2]. Fields and methods are members of the class, though they are accessible from instances, and constructors can define fields specific to an instance.

## 4.3 µPython

The syntax of the target language of our translation, µPython, is given in Figure 6. µPython differs significantly from the "classic" cast calculi used by gradually typed calculi, such as $\lambda_{\rightarrow}^{\langle\tau\rangle}$ [29]. Typically, such targets are statically typed languages that use explicit casts to inject to and project from the dynamic type. In contrast, µPython is dynamically typed, like Python itself, and has an expression that performs transient checks.

$$
\begin{array}{lcl}
e & ::= & x \mid n \mid e(\overline{e}) \mid e.\ell \mid e.\ell = e \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \\
 & & \texttt{class}(\overline{e})\text{: }\texttt{init} = e; \overline{\ell = e} \mid \lambda\overline{x}.\ e \mid e \Downarrow_S \mid a \\
S & ::= & \mathsf{pyobj} \mid \mathsf{int} \mid n\!\rightarrow \mid \mathsf{class}\ (\!|\delta, C|\!) \mid \mathsf{object}\ (\!|\delta|\!) \\
C & ::= & n \mid \mathsf{Any} \\
\delta & ::= & \overline{\ell}
\end{array}
$$

**Figure 6** Syntax for µPython

### 4.3.1 Expressions

The expressions of µPython mostly correspond to the terms of Anthill. Class declarations do not syntactically distinguish between functions, methods, and fields, so all class members are defined together, and constructors still appear as a separate entry in the class declaration but are now simply expressions. The only new expressions introduced in µPython are heap addresses and the check expression, $e \Downarrow_S$, which checks that the runtime type of $e$ after evaluation corresponds to a *type tag* $S$.

### 4.3.2 Type tags

The runtime type checks of µPython check against type tags $S$. These tags represent information that can be checked *shallowly* and *immediately* about µPython values. The tag pyobj is satisfied by all objects, int is satisfied by integers, and $n\!\rightarrow$ is satisfied by functions of $n$ parameters, but makes no claims about the types of the parameters or return type of the function. Object and class tags $\delta$ contain the set of attributes for their values, but not the types of their contents. Class tags also optionally specify the arity of their constructors.

## 4.4 Translation from Anthill to µPython

Following the typical gradual typing approach, Anthill's runtime semantics are defined in terms of a type-directed translation into µPython. This translation rejects programs that are statically ill-typed and inserts runtime checks according to the transient design. As discussed in Section 3.2, transient inserts checks on all use sites of mutable and higher-order terms, such as function applications or attribute accesses. The translation is designed to ensure that, if a Anthill term $t$ has type $A$, then its µPython translation $e$ can be checked against the type tag $\lfloor A \rfloor$ (defined in Figure 8) without resulting in an error.

In this section, we discuss the details of this translation with reference to excerpts of the translation relation. This translation is an algorithmic, syntax-directed transformation. The translation is specified in full in Appendix A, Figure 16.

### 4.4.1 Attribute reads and writes

Figure 7 shows the typechecking and translation rules for object reads and writes, as well as the *mems* metafunction, which returns the attribute type $\Delta$ representing the attributes statically known to be present in the value being read from or written to.

Rules IGet and ISet are used for reads and writes from expressions that are statically known to contain the attribute. In these cases, a check is inserted around the resulting expression (IGet) or the expression to be written (ISet) to make sure it corresponds with the type specified by the type of the object. The type tag for this check is generated from a static Anthill type using the $\lfloor A \rfloor$ metafunction, shown in Figure 8. In the case of ISet, the type of the term to be written $t_2$ must also be subtype-consistent (written $\lesssim$) with the type of the attribute. This prevents static type errors, such as attempting to write an int into a field that expects an object. Subtype-consistency, which can be thought of as subtyping "up to" type dynamic [30], is defined in Figure 8, and indirectly uses the consistency relation $\sim$ [29] from Appendix A, Figure 21.

The other rules, IGet-Check and ISet-Check are used when it is not statically known that the subject of a read or write contains the appropriate attribute, either because it is of type $\star$, or because it has a class or object type that omits the attribute. In these cases, the type system only allows the read or write if the object's type is *open* (denoted $\lozenge$) — the

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

(IGet)
$$\frac{\Gamma \vdash t \rightsquigarrow e : A_1 \qquad \Delta = mems(A_1) \qquad \Delta(\ell) = A_2}{\Gamma \vdash t.\ell \rightsquigarrow (e.\ell \Downarrow_{\lfloor A_2 \rfloor}) : A_2}$$

(IGet-Check)
$$\frac{\Gamma \vdash t \rightsquigarrow e : A_1 \qquad \Delta = mems(A_1) \qquad \ell \notin dom(\Delta) \qquad queryable(A_1) = \Diamond}{\Gamma \vdash t.\ell \rightsquigarrow (e \Downarrow_{\mathsf{object}\ (\!|\ell|\!)}).\ell : \star}$$

(ISet)
$$\frac{\Gamma \vdash t_1 \rightsquigarrow e_1 : A_1 \qquad \Gamma \vdash t_2 \rightsquigarrow e_2 : A_2' \qquad \Delta = mems(A_1) \qquad \Delta(\ell) = A_2 \qquad A_2' \lesssim A_2}{\Gamma \vdash t_1.\ell = t_2 \rightsquigarrow (e_1.\ell = (e_2 \Downarrow_{\lfloor A_2 \rfloor})) : \mathsf{int}}$$

(ISet-Check)
$$\frac{\Gamma \vdash t_1 \rightsquigarrow e_1 : A_1 \qquad \Gamma \vdash t_2 \rightsquigarrow e_2 : A_2 \qquad \Delta = mems(A_1) \qquad \ell \notin dom(\Delta) \qquad queryable(A_1) = \Diamond}{\Gamma \vdash t_1.\ell = t_2 \rightsquigarrow (e_1 \Downarrow_{\mathsf{object}\ (\!|\emptyset|\!)}).\ell = e_2 : \mathsf{int}}$$

$\boxed{mems(A) = \Delta}$

$$\begin{aligned} mems(\star) &= \emptyset \\ mems(\mathsf{object}\ (\!|q, \Delta|\!)) &= \Delta \\ mems(\mathsf{class}\ (\!|q, \Delta_1, \Delta_2, \overline{A}|\!)) &= \Delta_1 \end{aligned}$$

$\boxed{queryable(A) = q}$

$$\begin{aligned} queryable(\star) &= \Diamond \\ queryable(\mathsf{object}\ (\!|q, \Delta|\!)) &= q \\ queryable(\mathsf{class}\ (\!|q, \Delta_1, \Delta_2, \overline{A}|\!)) &= q \end{aligned}$$

$\boxed{instantiate(\Delta, \Delta) = \Delta}$

$$\frac{\Delta_1' = \{x{:}inst\text{-}fun(A) \mid x{:}A \in \Delta_1\} \qquad \Delta_2' = \{x{:}A \mid x{:}A \in \Delta_2, x \notin dom(\Delta_1)\}}{instantiate(\Delta_1, \Delta_2) = \Delta_1' \cup \Delta_2'}$$

$\boxed{inst\text{-}fun(A) = A}$

$$inst\text{-}fun(A_0, \ldots, A_n \to A) = A_1, \ldots, A_n \to A$$
$$inst\text{-}fun(A) = A \text{ if } A \neq \overline{A_1} \to A_2$$

**Figure 7** Translation for attribute reads and writes

$\boxed{\lfloor A \rfloor = S}$

$$\lfloor \star \rfloor = \mathsf{pyobj} \qquad \lfloor \mathsf{int} \rfloor = \mathsf{int} \qquad \lfloor \overline{A_1} \to A_2 \rfloor = |\overline{A_1}| \to \qquad \lfloor \mathsf{object}\ (\!|q, \Delta|\!) \rfloor = \mathsf{object}\ (\!|\lfloor \Delta \rfloor|\!)$$

$$\lfloor \mathsf{class}\ (\!|q, \Delta_1, \Delta_2, \overline{A}|\!) \rfloor = \mathsf{class}\ (\!|\lfloor \Delta_1 \rfloor, |\overline{A}|\!|) \qquad \lfloor \Delta \rfloor = \{x \mid x{:}A \in \Delta\}$$

$\boxed{A \lesssim A}$

$$\star \lesssim A \qquad A \lesssim \star \qquad \mathtt{int} \lesssim \mathtt{int} \qquad \frac{|A_1| = |A_3| \qquad \overline{A_3 \lesssim A_1} \qquad A_2 \lesssim A_4}{\overline{A_1} \to A_2 \lesssim \overline{A_3} \to A_4}$$

$$\frac{\Delta_1 \lesssim \Delta_2}{\mathsf{object}\ (\!|q_1, \Delta_1|\!) \lesssim \mathsf{object}\ (\!|q_2, \Delta_2|\!)} \qquad \frac{\Delta_1 \lesssim \Delta_3 \qquad \Delta_2 \lesssim \Delta_4 \qquad |A_1| = |A_2| \qquad \overline{A_2 \lesssim A_1}}{\mathsf{class}\ (\!|q_1, \Delta_1, \Delta_2, \overline{A_1}|\!) \lesssim \mathsf{class}\ (\!|q_2, \Delta_3, \Delta_4, \overline{A_2}|\!)}$$

$$\frac{\Delta_1 \lesssim \Delta_3}{\mathsf{class}\ (\!|q_1, \Delta_1, \Delta_2, \overline{A}|\!) \lesssim \mathsf{object}\ (\!|q_2, \Delta_3|\!)} \qquad \frac{\overline{A_2 \lesssim A_1} \qquad \mathsf{object}\ (\!|q, instantiate(\Delta_1, \Delta_2)|\!) \lesssim A_3}{\mathsf{class}\ (\!|q, \Delta_1, \Delta_2, \overline{A_1}|\!) \lesssim \overline{A_2} \to A_3}$$

$\boxed{\Delta \lesssim \Delta}$

$$\frac{\forall x \in dom(\Delta_2).\ \Delta_1(x) \sim \Delta_2(x)}{\Delta_1 \lesssim \Delta_2}$$

$\boxed{\Delta \sim \Delta}$

$$\frac{\forall x \in dom(\Delta_1) \cap dom(\Delta_2).\ \Delta_1(x) \sim \Delta_2(x)}{\Delta_1 \sim \Delta_2}$$

**Figure 8** Relations used by the Anthill translation. Vertical bars denote the size of a collection.

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

$$\text{(IFun)}\ \frac{\Gamma, \overline{x : A_1} \vdash t \rightsquigarrow e : A_2' \qquad A_2' \lesssim A_2}{\Gamma \vdash (\lambda \overline{x{:}A_1} \to A_2.\ t) \rightsquigarrow (\lambda \overline{x}.\ \overline{\texttt{let } x = x \Downarrow_{\lfloor A_1 \rfloor} \texttt{ in}}\ e) : \overline{A_1} \to A_2}$$

$$\text{(IApp-Dyn)}\ \frac{\Gamma \vdash t_1 \rightsquigarrow e_1 : \star \qquad \Gamma \vdash \overline{t_2 \rightsquigarrow e_2 : A}}{\Gamma \vdash t_1(\overline{t_2}) \rightsquigarrow (e_1 \Downarrow_{|\overline{A}| \to})(\overline{e_2}) : \star}$$

$$\text{(IApp-Fun)}\ \frac{\Gamma \vdash t_1 \rightsquigarrow e_1 : \overline{A_1} \to A_2 \qquad \Gamma \vdash \overline{t_2 \rightsquigarrow e_2 : A_1'} \qquad |\overline{A_1}| = |\overline{t_2}| \qquad \overline{A_1' \lesssim A_1}}{\Gamma \vdash t_1(\overline{t_2}) \rightsquigarrow (e_1(\overline{e_2})) \Downarrow_{\lfloor A_2 \rfloor} : A_2}$$

$$\text{(IApp-Constr)}\ \frac{\Gamma \vdash t_1 \rightsquigarrow e_1 : \mathsf{class}\ (\!|q, \Delta_1, \Delta_2, \overline{A_1}|\!) \qquad \Gamma \vdash \overline{t_2 \rightsquigarrow e_2 : A_1'} \qquad |\overline{A_1}| = |\overline{t_2}| \qquad \overline{A_1' \lesssim A_1} \qquad A_2 = \mathsf{object}\ (\!|q, \mathit{instantiate}(\Delta_1, \Delta_2)|\!)}{\Gamma \vdash t_1(\overline{t_2}) \rightsquigarrow (e_1(\overline{e_2})) \Downarrow_{\lfloor A_2 \rfloor} : A_2}$$

**Figure 9** Translation for functions and applications

definition of the *queryable* metafunction means that a $\star$-typed object is always $\Diamond$. Width downcasts are implicitly performed on open types but not on closed types ($\blacklozenge$). IGet-Check inserts a check to ensure that the attribute is present, but ISet-Check only checks that the subject of the write is an object (i.e. not an integer or function) because attribute writes can be used to add new attributes to objects and classes.

### 4.4.2 Functions and applications

The translation from Anthill to µPython for functions and applications is shown in Figure 9. Anthill functions are translated into µPython functions using the IFun rule, which inserts checks to ensure that parameters have their expected types in the body of the function. Functions are not responsible for ensuring that they return results that correspond to their return types (modulo static type errors). Instead, callers check that the resultant value is of the appropriate type, as indicated by IApp-Fun and IApp-Constr. This is especially important for constructor calls, which operate by mutating a dynamically typed empty receiver (as discussed in Section 4.1 and below in Section 4.4.3). The caller is responsible for checking that the resulting object has the type of an instance of the class.

### 4.4.3 Classes and objects

The rule for translating Anthill Python class definitions into µPython is given as IClass in Figure 10. In this translation, the superclasses of the class are translated and have checks placed around them to ensure that they are, in fact, classes — an error occurs if a class declaration inherits from a non-class value.

Class fields, methods, and the constructor are also translated into µPython expressions, and the latter two cases require new judgments. Constructors and methods are similar to functions, using their first argument as the receiver. In IConstruct, the receiver $x_s$ has dynamic type, because the instance has yet to actually be constructed. No concern is paid to the return type because the return value is discared when called. Methods, translated by IMethod, are similar, except that the receiver is given the type of an instance of the

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

(ICLASS)

$$\frac{\Gamma \vdash \overline{t_s \rightsquigarrow e_s : A_s} \quad A_{class} = \mathsf{class}\ (\!|q, \Delta_1, \Delta_2, \overline{A_c}|\!) \quad \Gamma \vdash_\sigma c \rightsquigarrow e_c : \overline{A_c} \quad \Gamma; A_{class} \vdash_\varsigma \overline{m \rightsquigarrow e_m : A_m} \quad \Gamma \vdash \overline{t_f \rightsquigarrow e_f : A_f} \quad \overline{e'_s = e_s \Downarrow_{\mathsf{class}\ (\!|\lfloor mems(A_s) \rfloor, \mathsf{Any}|\!)}} \quad \forall x \in dom(\Delta_1),\ (\overline{\ell_f} \times \overline{A_f} \cup \overline{\ell_m} \times \overline{A_m} \cup \overline{mems(A_s)})(x) \lesssim \Delta_1(x)}{\Gamma \vdash \mathtt{class}(\overline{t_s}){:}(\!|q, \Delta_1, \Delta_2|\!)\ \mathtt{init}{=}c; \overline{\ell_f =^M m}; \overline{\ell_m =^F t_m} \rightsquigarrow \mathtt{class}(\overline{e'_s})\mathtt{:}\ \mathtt{init} = e_c; \overline{\ell_m = e_m}, \overline{\ell_f = e_f} : A_{class}}$$

$\boxed{\Gamma \vdash_\sigma c \rightsquigarrow e : \overline{A}}$

$$\text{(ICONSTRUCT)}\ \frac{\Gamma, x_s{:}\star, \overline{x{:}A_1} \vdash t \rightsquigarrow e : A_2}{\Gamma \vdash_\sigma \sigma\ x_s, \overline{x{:}A_1}.\ t \rightsquigarrow \lambda x_s, \overline{x}.\ \overline{\mathtt{let}\ x = x \Downarrow_{\lfloor A_1 \rfloor}\ \mathtt{in}}\ e : \overline{A_1}}$$

$\boxed{\Gamma; A \vdash_\varsigma m \rightsquigarrow e : A}$

(IMETHOD)

$$\frac{A_o = \mathsf{object}\ (\!|q, instantiate(\Delta_1, \Delta_2)|\!) \quad A'_2 \lesssim A_2 \quad \Gamma, x_s{:}A_o, \overline{x{:}A_1} \vdash t \rightsquigarrow e : A'_2}{\Gamma; \mathsf{class}\ (\!|q_1, \Delta_1, \Delta_2, \overline{A_c}|\!) \vdash_\varsigma \varsigma\ x_s, \overline{x{:}A_1}{\to}A_2.\ t \rightsquigarrow \lambda x_s, \overline{x}.\ \mathtt{let}\ x_s = x_s \Downarrow_{\lfloor A_o \rfloor}\ \mathtt{in}\ \overline{\mathtt{let}\ x = x \Downarrow_{\lfloor A_1 \rfloor}\ \mathtt{in}}\ e : \overline{A_1} \to A_2}$$

**Figure 10** Translation for classes and methods

enclosing class. In both cases, as in IFUN, checks are inserted to ensure that the parameters have the correct type.

ICLASS also ensures that all declared class attributes (those in $\Delta_1$) can be found in the fields, methods, or superclasses of the class, and that their types are subtype-consistent with their declared types.

As observed by Takikawa et al. [35], width subtyping for class types is statically unsound, and because Anthill's object types support width subtyping, classes do as well, as shown in Figure 8. Anthill's static type system does, therefore, admit some maximally annotated programs that result in runtime check failures. (Since the self-reference of a constructor is always dynamically typed, programs with classes are never *fully* annotated anyway.) However, the pervasive nature of transient's runtime checking means that this cannot lead to uncaught type errors at runtime.

### 4.4.4 Eager and delayed error detection

The translation from Anthill to µPython inserts checks only when necessary to ensure soundness, whereas Reticulated performs extra eager checks to provide better feedback to programmers. Eagerly checking objects does not aid in ensuring soundness, however, and therefore the translation from Anthill to µPython installs only shallow checks at use sites. For example, in the following program, a class specifies that its instances should have the field `x:int`. However, its constructor instead writes a function to `x`.

```
1 let C = class ()(⦇◇, ∅, x:int⦈):
2           init=(σ self. self.x=λ z. z)
3 in let c = C()
4 in c.x
```

Reticulated would detect this error at the object instantiation site at line 3, because `c.x` is a function, not an int. However, even if the constructor was corrected, `c.x` could still become a non-`int`, if some untyped reference to `c` strongly updated `x`. In that case, fully checking `c` at line 3 would have accomplished nothing in terms of ensuring soundness — it still needs to be checked again at every use site. In Anthill, the error in the above program is detected at the use site on line 4, when an inserted check attempts to verify that the result of `c.x` is an `int`.

## 4.5 Dynamic semantics of µPython

The runtime behavior of µPython is defined in terms of a single-step reduction semantics using evaluation contexts, shown in Figure 12. The runtime structures (values, heaps, etc.) are defined in Figure 11. The reduction relation $e \mid \mu \longrightarrow r$ steps from a pair of an expression $e$ and a heap $\mu$ to some result $r$, which is either an error or a pair of an expression and a heap. Values are numbers, functions, and heap addresses. Heaps contain mappings from addresses $a$ to heap values $h$, which are either classes or objects. Class heap values contain references to all superclasses and all of the class' attributes, and objects contain instance variables and a reference to the object's class. Error results come in two varieties: `casterror`, which is the result of a check $v \Downarrow_S$ that fails, or `pyerror`, which is the result of a dynamic type error (such as, for example, calling a number as though it were a function). We choose to have such cases reduce to an actual result, rather than treating them as stuck states, because µPython is a dynamically typed language which contains not just programs translated from Anthill, but other programs which produce runtime errors; a µPython program that reduces to `pyerror` is the equivalent of a Python program that raises a `TypeError` exception. As rules EPyError and ECastError in Figure 12 show, both kinds of errors are propagated upwards by the context rules.

Rules ECheck1 and ECheck2 show µPython's reduction rules for transient's type checks, which use the *check* metafunction defined in Figure 12. In all cases, the reduction results in either a `casterror` or the checked value itself. The cases where the checked value is an address are the most interesting: class values can be called like functions and read from/written to like objects, so the check can be successful with both function and object type tags. The *hasattrs* metafunction, used for verifying if all attributes in $\delta$ are reachable

$$
\begin{array}{lll}
v & ::= & n \mid a \mid \lambda \overline{x}.\ e \\
E & ::= & \square \mid E(\overline{e}) \mid v(\overline{v}, E, \overline{e}) \mid E.\ell \mid E.\ell = e \mid v.\ell = E \mid \texttt{let}\ x = E\ \texttt{in}\ e \mid E \Downarrow_S \mid \\
 & & \texttt{class}(\overline{v}, E, \overline{e})\texttt{: init} = e; \overline{\ell = e} \mid \texttt{class}(\overline{v})\texttt{: init} = E; \overline{\ell = e} \mid \\
 & & \texttt{class}(\overline{v})\texttt{: init} = v; \overline{\ell = v}, \ell = E, \overline{\ell = e} \\
\mu & ::= & \overline{a \mapsto h} \\
h & ::= & \texttt{Class}(\overline{a})\{\texttt{init} = e; M\} \mid \texttt{Object}(a)\{M\} \\
M & ::= & \overline{\ell = v} \\
r & ::= & (e \mid \mu) \mid \texttt{casterror} \mid \texttt{pyerror}
\end{array}
$$

**Figure 11** Runtime syntax for µPython

from a heap value, and *param-match*, used for checking the arity of a callable value, are defined in Appendix A, Figure 21.

Most remaining reduction rules for µPython are standard, except that every case that would be stuck in a statically typed calculus has a reduction to `pyerror`. Application requires some work: the value being called may be a class, in which case an empty object is created and passed to the object's constructor (EApp2). Method lookup on objects, shown in rule EGet1 and the *lookup* metafunction, is complex: methods have to be looked up from a class, curried, and given the receiver as the first argument. The *getattr* partial function, which traverses the inheritance hierarchy of its argument to find the definition site of an attribute, is defined in Appendix A, Figure 21.

One quirk to the *lookup* metafunction is that attempting to access a class' nullary method from an instance leads directly to a `casterror`, rather than a `pyerror`. In Python, method arities are reduced by one when called from an instance, because the instance is bound to the first parameter as the receiver. A nullary method bound like this essentially expects $-1$ arguments: calling it with zero arguments results in a type error, because too many arguments were provided. In Reticulated, a transient check on such a function always fails, but rather than providing uncallable, "negative-ary" functions in Anthill, such an evaluation simply results in `casterror`.

The semantics of µPythondo not refer to casts, types, or other features related to translation from Anthill, except in the type check rules and the method access rule (as mentioned above) — everything else is as expected for an untyped language. This supports our claim that µPython models Python, and that we do not have to modify the underlying semantics of Python itself in order to implement Reticulated with the transient semantics, nor do we require a complicated implementation of proxies capable of handling Python's features, as does guarded. The complex parts of µPython's semantics implement features like method binding, and are not affected by the existence of runtime type checks.

## 5 Open World Soundness of Anthill Python

With static and dynamic semantics for Anthill Python, we now wish to show that Anthill admits the property of *open world soundness*. That is, an Anthill program, translated into µPython, can interact with other µPython code without the translated Anthill code causing any `pyerror`s. However, in our presentation so far µPython does not distinguish between translated and untranslated code, so we begin by introducing *origin tracking* to µPython expressions. This indicates whether an expression originated in typed or untyped code.

Since µPython is a dynamically typed language, there is nothing preventing programs from resulting in runtime errors. For example, consider the following µPython program, in which a function `f` calls its argument on the integer 42.

```
1 let f = (λ v. v(42)) in
2 f(21)
```

Since `f` is passed 21 at line 2, this program will result in a `pyerror` at the call site in line 1 by rule EApp3. Suppose instead that `f` was typed Anthill Python code rather than µPython:

```
1 let f = (λ v:Callable([int], int). v(42)) in
2 ...
```

This version specifies that `f`'s parameter be a function. It can then be translated into µPython and used in place of the original `f`:

$\boxed{e \mid \mu \longrightarrow r}$

$$\text{(EStep)}\ \frac{e \mid \mu \longrightarrow e' \mid \mu'}{E[e] \mid \mu \longmapsto E[e'] \mid \mu'} \qquad \text{(EPyError)}\ \frac{e \mid \mu \longrightarrow \texttt{pyerror}}{E[e] \mid \mu \longmapsto \texttt{pyerror}} \qquad \text{(ECastError)}\ \frac{e \mid \mu \longrightarrow \texttt{casterror}}{E[e] \mid \mu \longmapsto \texttt{casterror}}$$

$$\text{(ECheck1,2)}\quad v \Downarrow_S \mid \mu \longrightarrow \begin{cases} v \mid \mu & \text{if } \mathit{check}(v, \mu, S) \\ \texttt{casterror} & \text{otherwise} \end{cases}$$

$$\text{(EApp1,2,3)}\quad v_1(\overline{v_2}) \mid \mu \longrightarrow \begin{cases} e[\overline{x/v_2}] \mid \mu & \text{if } v_1 = \lambda \overline{x}.e \\ & \text{and } |\overline{x}| = |\overline{v_2}| \\ \texttt{let } \_ = & \text{if } v_1 = a \\ \quad v'(a', \overline{v_2}) & \text{and } \mu(a) = \texttt{Class}(\overline{a''})\{\texttt{init} = v'; M\} \\ \quad \texttt{in } a' \mid \mu' & \text{and } a' \text{ fresh} \\ & \text{and } \mu' = \mu[a' \mapsto \texttt{Object}(a)\{\emptyset\}] \\ \texttt{pyerror} & \text{otherwise} \end{cases}$$

$$\text{(ELet)}\quad \texttt{let } x = v \texttt{ in } e \longrightarrow e[x/v]$$

$$\text{(EClass1,2)}\quad \texttt{class}(\overline{a})\texttt{: init} = v; M \mid \mu \longrightarrow \begin{cases} a' \mid \mu[a' \mapsto h] & \text{if } \overline{\mu(a) = \texttt{Class}(\overline{a''})\{\texttt{init} = v'; M'\}} \\ & \text{and } \mathit{param\text{-}match}(v, \mu, \mathsf{Any}) \\ & \text{and } h = \texttt{Class}(\overline{a})\{\texttt{init} = v; M\} \\ & \text{and } a' \text{ fresh} \\ \texttt{pyerror} & \text{otherwise} \end{cases}$$

$$\text{(EClass3)}\quad \texttt{class}(\overline{v_1})\texttt{: init} = v_2; M \mid \mu \longrightarrow \texttt{pyerror} \quad \text{if } \overline{v_1 \neq a}$$

$$\text{(EGet1,2)}\quad a.\ell \mid \mu \longrightarrow \begin{cases} r & \text{if } \mathit{lookup}(a, \mu(a), \ell, \mu) = r \\ \texttt{pyerror} & \text{otherwise} \end{cases}$$

$$\text{(EGet3)}\quad v.\ell \mid \mu \longrightarrow \texttt{pyerror} \quad \text{if } v \neq a$$

$$\text{(ESet1,2,3)}\quad a.\ell = v \mid \mu \longrightarrow \begin{cases} 0 \mid \mu[a \mapsto h'] & \text{if } \mu(a) = \texttt{Object}(a')\{M\} \\ & \text{and } h' = \texttt{Object}(a')\{M[\ell = v]\} \\ 0 \mid \mu[a \mapsto h'] & \text{if } \mu(a) = \texttt{Class}(\overline{a'})\{\texttt{init} = v'; M\} \\ & \text{and } h' = \texttt{Class}(\overline{a'})\{\texttt{init} = v'; M[\ell = v]\} \\ \texttt{pyerror} & \text{otherwise} \end{cases}$$

$$\text{(ESet4)}\quad v_1.\ell = v_2 \mid \mu \longrightarrow \texttt{pyerror} \quad \text{if } v_1 \neq a$$

$\boxed{e \mid \mu \longrightarrow^* r}$

$$\text{(MRefl)}\ \frac{}{e \mid \mu \longrightarrow^* e \mid \mu} \qquad \text{(MPyErr)}\ \frac{e \mid \mu \longmapsto \texttt{pyerror}}{e \mid \mu \longrightarrow^* \texttt{pyerror}}$$

$$\text{(MCastErr)}\ \frac{e \mid \mu \longmapsto \texttt{casterror}}{e \mid \mu \longrightarrow^* \texttt{casterror}} \qquad \text{(MChain)}\ \frac{e \mid \mu \longmapsto e' \mid \mu' \qquad e' \mid \mu' \longrightarrow^* r}{e \mid \mu \longrightarrow^* r}$$

**Figure 12** µPython evaluation rules

$\boxed{check(v, \mu, S)}$

$$\frac{}{check(v, \mu, \mathsf{pyobj})} \qquad \frac{}{check(n, \mu, \mathsf{int})} \qquad \frac{hasattrs(a, \delta, \mu)}{check(a, \mu, \mathsf{object}\ (\!|\delta|\!))} \qquad \frac{|\overline{x}| = n}{check(\lambda\overline{x}.e, \mu, n\rightarrow\ )}$$

$$\frac{\mu(a) = \texttt{Class}(\overline{a'})\{\texttt{init} = v; M\} \quad param\text{-}match(v, \mu, n+1)}{check(a, \mu, n\rightarrow\ )} \qquad \frac{\mu(a) = \texttt{Class}(\overline{a'})\{\texttt{init} = v; M\} \quad param\text{-}match(v, \mu, C) \quad hasattrs(a, \delta, \mu)}{check(a, \mu, \mathsf{class}\ (\!|\delta, C|\!))}$$

$\boxed{lookup(a, h, \ell, \mu) = r}$

$$\frac{M(\ell) = v}{lookup(a, \texttt{Object}(a')\{M\}, \ell, \mu) = v \mid \mu} \qquad \frac{getattr(a, \ell, \mu) = v}{lookup(a, \texttt{Class}(\overline{a'})\{\texttt{init} = v'; M\}, \ell, \mu) = v \mid \mu}$$

$$\frac{\ell \notin dom(M) \quad getattr(a, \ell, \mu) = v \quad v \neq \lambda\overline{\ell}.e}{lookup(a, \texttt{Object}(a')\{M\}, \ell, \mu) = v \mid \mu} \qquad \frac{\ell \notin dom(M) \quad getattr(a, \ell, \mu) = v \quad v = \lambda\overline{x}.e \quad |\overline{y}| = |\overline{x}| - 1}{lookup(a, \texttt{Object}(a')\{M\}, \ell, \mu) = \lambda\overline{y}.v(a, \overline{y}) \mid \mu}$$

$$\frac{\ell \notin dom(M) \quad getattr(a, \ell, \mu) = v \quad v = \lambda\overline{x}.e \quad |\overline{x}| = 0}{lookup(a, \texttt{Object}(a')\{M\}, \ell, \mu) = \texttt{casterror}}$$

**Figure 13** Relations used in µPython evaluation

```
1 let f = (λ v. let v = v⇓1→ in
2                (v(42))⇓int) in
3 f(21)
```

The result is a µPython program that is partially translated, typed Anthill code (highlighted in yellow) and partially code that originates in µPython. In this case, the result of the program is no longer `pyerror`, but a `casterror` at line 1, which indicates that `f` has been passed something of the wrong type. Open world soundness guarantees that mixed programs like this can only result in `pyerror` in those sections that originated in µPython.

To reason about such mixed programs, we describe portions of a program as originating in either Anthill or µPython. Consider again the example from above, but with origin made explicit:

```
1 let f = (λ v. let v = v⇓1→ in
2                (v(42)°)⇓int) in
3 f(21)•
```

The call site at 2 is labeled ∘ to indicate that it originated Anthill, and should never result in `pyerror`. The call site at line 3, on the other hand, is labeled • to indicate that it originated in µPython, and *can* result in `pyerror`.

In the remainder of this section, we further develop this notion of origin, and use it in defining a type system for µPython, which enforces restrictions on code that originated in Anthill while being permissive for code that does not. We then use this type system to prove open world soundness.

$$
\begin{array}{lcl}
p & ::= & \circ \mid \bullet \\
e & ::= & \ldots \mid e(\overline{e})^p \mid e.\ell^p \mid e.\ell^p = e \mid \texttt{class}(\overline{e})\texttt{:}^p\ \texttt{init} = e; \overline{\ell = e} \\
r & ::= & \ldots \mid \texttt{pyerror}(p) \\
\Sigma & ::= & \overline{a\texttt{:}S} \\
\mathcal{C} & ::= & \Box \mid \mathcal{C}(\overline{e})^\bullet \mid e(\overline{e}, \mathcal{C}, \overline{e})^\bullet \mid \mathcal{C}.\ell^\bullet \mid \mathcal{C}.\ell^\bullet = e \mid e.\ell^\bullet = \mathcal{C} \mid \texttt{class}(\overline{e}, \mathcal{C}, \overline{e})\texttt{:}^\bullet\ \texttt{init} = e; \overline{\ell = e} \mid \\
 & & \texttt{class}(\overline{e})\texttt{:}^\bullet\ \texttt{init} = \mathcal{C}; \overline{\ell = e} \mid \texttt{class}(\overline{v})\texttt{:}^\bullet\ \texttt{init} = e; \overline{\ell = e}, \ell = \mathcal{C}, \overline{\ell = e} \mid \\
 & & \texttt{let}\ x = \mathcal{C}\ \texttt{in}\ e \mid \texttt{let}\ x = e\ \texttt{in}\ \mathcal{C} \mid \mathcal{C} \Downarrow_S \mid \lambda \overline{x}.\ \mathcal{C}
\end{array}
$$

**Figure 14** Labeled syntax and contexts for µPython

## 5.1 Expression origin

Figure 14 shows the syntax of µPython extended with origin labels $p$. In this system, there are only two labels: $\circ$, for code translated from well-typed Anthill Python programs, and $\bullet$, for code that originates in µPython and is not statically typed.

Unlike typical blame labels in contracts and gradual typing, which attach to casts or contract monitors [3, 15, 35], these labels are attached to expressions, and specifically to elimination forms that can result in `pyerror`. (No labels are attached to, for example, let-binding, because let-binding cannot on its own lead to a `pyerror`.) We also change the `pyerror` result so that it reports the origin of the expression that triggered the error. Code translated from Anthill is *always* labeled with $\circ$, while µPython programs entirely labeled with $\bullet$ represent untyped Python code. The version of µPython's dynamic semantics that includes origin is shown in Appendix A, Figure 19; it differs from that shown in Figure 12 only in its use and propagation of labels.

## 5.2 Applying types to µPython

In addition to indicating whether a dynamic type error occurred in typed or untyped code, origin labels also let us define a type system for µPython in order to state and prove open world soundness. This type system relates expressions to type tags $S$, as defined in Figure 6, which are repurposed as types. An illustrative excerpt of this type system is shown in Figure 15. Two rules are provided for each labeled expression, one for $\circ$ and one for $\bullet$. The $\bullet$ rule requires that all subexpressions be typed as pyobj, which is the top of the subtyping hierarchy for $S$ as shown in Figure 8, so no programs can be rejected unless they contain $\circ$-labeled expressions. The $\circ$ rules are more restrictive — a $\circ$-labeled expression is well-typed when it cannot step to `pyerror`, and so $\circ$ rules require that subexpressions have the specific types necessary to ensure that. All expressions of either variety, other than checks and introduction forms, are typed as pyobj. This ensures runtime type checks exist in $\circ$-labeled code, because type checks are the only way to obtain expressions with precise types like int and $n\!\rightarrow$ (other than introduction forms like numbers or lambdas).

In rule TCheck, the type of a check expression is its type tag. In TApp we require that, in a call labeled with $\circ$, the callee has a function type. The only way for it to have a function type is if it is a bare lambda, if it is a variable bound by TLet, or if it is some other expression nested within a check (typed by TCheck). By contrast, TApp-Dyn places no requirements on the types of subexpressions of an untyped, $\bullet$-labeled application. Even if a subexpression has a more specific type, pyobj is the top of the subtyping hierarchy, so by TSubsump any well-typed expression can appear in function position. Therefore, an

$\boxed{\Gamma;\Sigma \vdash e : S}$

$$\frac{\Gamma;\Sigma \vdash e : S_2 \qquad S_2 <: S_1}{\Gamma;\Sigma \vdash e : S_1}\ (\textsc{TSubsump}) \qquad \frac{\Gamma;\Sigma \vdash e : \mathsf{pyobj}}{\Gamma;\Sigma \vdash e \Downarrow_S : S}\ (\textsc{TCheck}) \qquad \frac{\Gamma;\Sigma \vdash e_1 : S_1 \qquad \Gamma, x{:}S_1|\Sigma \vdash e_2 : S_2}{\Gamma;\Sigma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : S_2}\ (\textsc{TLet})$$

$$\frac{\Gamma;\Sigma \vdash e_1 : n\!\!\rightarrow \qquad \Gamma;\Sigma \vdash \overline{e_2 : \mathsf{pyobj}} \qquad |\overline{e_2}| = n}{\Gamma;\Sigma \vdash e_1(\overline{e_2})^{\circ} : \mathsf{pyobj}}\ (\textsc{TApp}) \qquad \frac{\Gamma;\Sigma \vdash e_1 : \mathsf{pyobj} \qquad \Gamma;\Sigma \vdash \overline{e_2 : \mathsf{pyobj}}}{\Gamma;\Sigma \vdash e_1(\overline{e_2})^{\bullet} : \mathsf{pyobj}}\ (\textsc{TApp-Dyn})$$

$\boxed{S <: S}$

$$S <: \mathsf{pyobj} \qquad \texttt{int} <: \texttt{int} \qquad \frac{S_1 <: S_2 \qquad S_2 <: S_3}{S_1 <: S_3} \qquad \frac{\delta_2 \subseteq \delta_1}{\mathsf{object}\ (\!|\delta_1|\!) <: \mathsf{object}\ (\!|\delta_2|\!)}$$

$$\frac{}{\mathsf{class}\ (\!|\delta, n|\!) <: \mathsf{class}\ (\!|\delta, \mathsf{Any}|\!)} \qquad \frac{\delta_2 \subseteq \delta_1}{\mathsf{class}\ (\!|\delta_1, C|\!) <: \mathsf{class}\ (\!|\delta_2, C|\!)} \qquad \frac{}{\mathsf{class}\ (\!|\delta, C|\!) <: \mathsf{object}\ (\!|\delta|\!)}$$

$$\frac{}{\mathsf{class}\ (\!|\delta, n|\!) <: n\!\!\rightarrow}$$

**Figure 15** Excerpt of type system for typed expressions in µPython

obviously ill-typed program like `4(2)` (calling 4 as though it were a function) will be ill-typed if the call is labeled with $\circ$, but allowed if it is labeled with $\bullet$. The full type system for µPython with labels is shown in Appendix A, Figure 17.

## 5.3 Code contexts allow embedding typed code in untyped

To reason about Anthill-translated code interacting with other µPython code, we use code contexts $\mathcal{C}$[19]. Contexts are defined in Figure 14; in this work we are concerned with embedding typed code in untyped contexts, so these contexts are $\bullet$-labeled. Code contexts are typed using the judgment $\mathcal{C} : \Gamma;S_1 \Rightarrow \Gamma';S_2$, where if the hole in a context $\mathcal{C}$ is filled by an expression of type $S_1$ under $\Gamma$, then the result is an expression of type $S_2$ under $\Gamma'$. We write $\mathcal{C}[e]$ for the composition of a context and an expression, which is itself an expression. For example, $\Box : \Gamma;S \Rightarrow \Gamma;S$, because a hole filled by an expression is just that expression. The full rules for typing contexts are given in Appendix A, Figure 20.

## 5.4 Open world soundness

Armed with notions of origination and a type system for µPython, we can now describe the proof of open world soundness and its key lemmas.

First, if an Anthill term $t$ has type $A$ under $\Gamma$ and is translated into a µPython expression $e$, then $e$ has type $\lfloor A \rfloor$ under $\lfloor \Gamma \rfloor$.[4]

▶ **Lemma 1** (Anthill translation preserves typing).
*If* $\Gamma \vdash t \rightsquigarrow e : A$, *then* $\lfloor \Gamma \rfloor;\emptyset \vdash e : \lfloor A \rfloor$.

[4] Here, $\lfloor \Gamma \rfloor$ is the result of applying $\lfloor A \rfloor$ (see Figure 8) to all the types in $\Gamma$.

- *If* $\Gamma \vdash c \rightsquigarrow e : \overline{A}$*, then* $\lfloor \Gamma \rfloor ; \emptyset \vdash e : |\overline{A}| \rightarrow$ .
- *If* $\Gamma ; A_1 \vdash d \rightsquigarrow e : A_2$*, then* $\lfloor \Gamma \rfloor ; \emptyset \vdash e : \lfloor A_2 \rfloor$

If $e$ is plugged in a context $\mathcal{C} : \lfloor \Gamma \rfloor ; \lfloor A \rfloor \Rightarrow \emptyset ; S$, then the resulting term $\mathcal{C}[e]$ has type $S$.

▶ **Lemma 2** (Composition). *If* $\mathcal{C} : \Gamma ; S \Rightarrow \Gamma' ; S'$ *and* $\Gamma ; \emptyset \vdash e : S$*, then* $\Gamma' ; \emptyset \vdash \mathcal{C}[e] : S'$.

Finally, when $\mathcal{C}[e]$ is evaluated, if it does not diverge it will result in a value of type $S$, a `casterror`, or a `pyerror` that points to untyped code as the source of the error (i.e. $e$ is not responsible).

▶ **Lemma 3** (Preservation). *If* $\emptyset ; \Sigma \vdash e : S$*,* $\Sigma \vdash \mu$*, and* $e \mid \mu \longrightarrow e' \mid \mu'$*, then* $\emptyset ; \Sigma' \vdash e' : S$ *and* $\Sigma' \vdash \mu'$ *and* $\Sigma \sqsubseteq \Sigma'$.

▶ **Lemma 4** (Progress with no typed pyerrors). *If* $\emptyset ; \Sigma \vdash e : S$ *and* $\Sigma \vdash \mu$*, then either* $e$ *is a value or* $e \mid \mu \longrightarrow r$ *and either:*

- $r = \texttt{pyerror}(\bullet)$*, or*
- $r = \texttt{casterror}$*, or*
- $r = e' \mid \mu'$.

These key lemmas let us prove the overall statement of open world soundness:

▶ **Theorem 5** (Open world soundness). *If* $\Gamma \vdash t \rightsquigarrow e : A$*, then for any context* $\mathcal{C}$ *such that* $\mathcal{C} : \lfloor \Gamma \rfloor ; \lfloor A \rfloor \Rightarrow \emptyset ; S$*, we have that* $\emptyset ; \emptyset \vdash \mathcal{C}[e] : S$ *and either:*

- *for some* $v, \Sigma, \mu$*,* $\mathcal{C}[e] \mid \emptyset \longrightarrow^* v \mid \mu$ *and* $\emptyset ; \Sigma \vdash v : S$ *and* $\Sigma \vdash \mu$*, or*
- $\mathcal{C}[e] \mid \emptyset \longrightarrow^* \texttt{pyerror}(\bullet)$*, or*
- $\mathcal{C}[e] \mid \emptyset \longrightarrow^* \texttt{casterror}$*, or*
- *for all* $r$ *such that* $\mathcal{C}[e] \mid \emptyset \longrightarrow^* r$*, have* $r = e' \mid \mu'$ *and there exists* $r'$ *such that* $e' \mid \mu' \longrightarrow r'$.

This theorem states that no program can ever result in `pyerror`$(\circ)$, which would indicate an uncaught type error in translated Anthill code. Type soundness in the usual sense is an immediate corollary of this theorem, by choosing $\mathcal{C}$ to be the empty context $\square$.

We proved this theorem for the Anthill and µPython languages using the Coq proof assistant; the completed proof files are available at `https://arxiv.org/src/1610.08476v1`. A sketch of the proof can also be examined in Appendix B. The proof combines a typical progress and preservation type soundness proof (for µPython, and so including many additional cases for errored terms) with proofs that the translation relation is type preserving and that composition of terms and contexts is well-typed.

## 5.5 Ramifications of open world soundness

Because Anthill admits open world soundness, a program written in Anthill can be used by native µPython clients. For example, an Anthill library can put type annotations on its API boundaries, and these types will be checked, preventing difficult-to-diagnose errors from arising deep within the library — even if the library is used by code which has no concept of static types. Furthermore, the Anthill code is protected from errors arising due to mutation. While foreign functions are not modeled directly in the Anthill and µPython calculi, note that the distinction between untyped Python programs and compiled C code is relevant in guarded because of the presence of proxies. Since Anthill and µPython lack proxies, however, this distinction is irrelevant, and such foreign functions can be modeled as untranslated µPython code, and thus open world soundness shows that calls to such code within Anthill programs will not interfere with Anthill's type soundness.

We conjecture that Reticulated Python is also open world sound when using the transient semantics; it is certainly closer to open world soundness than the guarded semantics. Evidence for this exists in the work of Vitousek et al. [40]. They found that Reticulated Python ran all of their case studies without error under transient, while it could not successfully execute one of them when using guarded. Further, more modifications to the typed programs were necessary when using guarded to avoid errors caused by proxy identity problems.

Combined with its ease of implementation, the fact that transient admits open world soundness means that it is a realistic and useful technique in designing gradually typed languages like Reticulated Python. It does require performance overhead in its pervasive checks, and we envision practical implementations offering a "switch" to allow developers to debug their program with thorough type checking and disable it for production (similar to, but more complete than, Dart's checked mode). Open world soundness demonstrates that it is useful in circumstances that challenge guarded, and its ease of implementation makes it more practical than other approaches.

## 6 Conclusions

The traditional guarded approach for the runtime semantics of gradually typed languages, based on proxies, is well-understood and powerful, but it is unsound in an open world in when applied to languages like Python. The transient design for gradual typing provides an alternative approach which is open world sound.

In this paper we develop a formal treatment of transient with calculi that model Reticulated Python and Python. Furthermore, we discuss a formal property that is relevant to source-to-source implementations of gradually typed languages, open world soundness, which states that typed code can be embedded in untranslated code without causing uncaught type errors. We prove that our calculi admit the open world soundness principle and mechanize the proof in Coq.

Many industrial gradually typed languages avoid runtime checking altogether. This is especially relevant when the system is designed to translate to an underlying language whose semantics cannot be modified by the designer of the surface language. Reticulated Python and the transient semantics demonstrate that gradual typing can be implemented straightforwardly, without modifying the target language's semantics, and while allowing interoperation and preserving soundness.

## References


1 M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Symposium on Principles of programming languages*, 1989.

2 Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

3 Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for All. In *Symposium on Principles of Programming Languages*, January 2011.

4 Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Markus Denker. Gradual typing for smalltalk. *Science of Computer Programming*, August 2013.

5 Esteban Allende, Johan Fabry, and Éric Tanter. Cast insertion strategies for gradually-typed objects. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, 2013.

6 Christopher Anderson and Sophia Drossopoulou. BabyJ - from object based to class based programming via types. In *WOOD '03*, volume 82. Elsevier, 2003.

7 Gavin Bierman, Erik Meijer, and Mads Torgersen. Adding dynamic types to C#. In *European Conference on Object-Oriented Programming*, ECOOP'10. Springer-Verlag, 2010.

8 Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn: Robust, concurrent, extensible scripting on the jvm. *SIGPLAN Not.*, 44(10):117–136, October 2009. ISSN 0362-1340.

9 Gilad Bracha and David Griswold. Strongtalk: Typechecking smalltalk in a production environment. OOPSLA '93, New York, NY, USA, 1993. ACM.

10 Robert Cartwright. User-defined data types as an aid to verifying lisp programs. In *ICALP'76*, volume 4596, 1976.

11 Craig 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.
12 Facebook. Hack, 2013. URL `http://hacklang.org`.
13 Manuel Fahndrich and Songtao Xia. Establishing object invariants with delayed types. In *OOPSLA '07*, pages 337–350, 2007.
14 R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming*, ICFP, pages 48–59, October 2002.
15 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. Technical Report NU-CCS-02-05, Northeastern University, 2002.
16 Cormac Flanagan. Hybrid type checking. In *POPL 2006: The 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–256, Charleston, South Carolina, January 2006.
17 Google. Dart: structured web apps, 2011. URL `http://dartlang.org`.
18 Kathryn E. Gray, Robert Bruce Findler, and Matthew 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.
19 Professor Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2012. ISBN 1107029570, 9781107029576.
20 David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. In *Trends in Functional Prog. (TFP)*, page XXVIII, April 2007.
21 Who Matthias Keil and Peter Thiemann. Transparent object proxies in javascript. In *European Conference on Object-Oriented Programming*, 2015.
22 Microsoft. Typescript, 2012. URL `http://www.typescriptlang.org/`.
23 Xin Qi and Andrew C. Myers. Masked types for sound object initialization. In *POPL '09*, pages 53–65, New York, NY, USA, 2009. ACM.
24 Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The ins and outs of gradual type inference. In *Symposium on Principles of Programming Languages*, POPL, pages 481–494, January 2012.
25 Brianna M. Ren, John Toman, T. Stephen Strickland, and Jeffrey S. Foster. The ruby type checker. In *SAC'13 (OOPS)*, 2013.
26 Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete Types for TypeScript. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 76–100, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
27 Manuel Serrano. *Bigloo: a practical Scheme compiler*. Inria-Rocquencourt, April 2002.
28 Jeremy G. Siek and Ronald Garcia. Interpretations of the gradually-typed lambda calculus. In *Scheme and Functional Programming Workshop*, 2012.
29 Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, September 2006.
30 Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, volume 4609 of *LCNS*, pages 2–27, August 2007.
31 Jeremy G. Siek, Ronald Garcia, and Walid Taha. Exploring the design space of higher-order casts. In *European Symposium on Programming*, ESOP, pages 17–31, March 2009.
32 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *SNAPL '15*, 2015.
33 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. Monotonic references for efficient gradual typing. In *ESOP '15*, April 2015.
34 Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. Gradual typing embedded securely in javascript. In *ACM Conference on Principles of Programming Languages (POPL)*, January 2014.
35 Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 793–810, 2012.
36 Satish Thatte. Quasi-static typing. In *POPL 1990*, pages 367–381, New York, NY, USA, 1990. ACM Press. ISBN 0-89791-343-4.
37 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium*, 2006.
38 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *POPL '08*, pages 395–406, 2008.
39 Tom Van Cutsem and Mark S. Miller. Trustworthy proxies: virtualizing objects with invariants. In *ECOOP'13*, pages 154–178, 2013.
40 Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for Python. In *Symposium on Dynamic Languages*, DLS '14, pages 1–16, 2014.
41 Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming*, ESOP, pages 1–16, March 2009.
42 Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In *Symposium on Principles of Programming Languages*, POPL, pages 377–388, 2010.

# A Appendix: Full semantics

Figure 16 shows the translation from Anthill Python to µPython, and Figure 17 shows the type system used to prove soundness for µPython. Figures 18 and 19 show the runtime semantics of µPython, and are identical to Figures 13 and 12 in the main body of the paper except for the addition of origin labels $p$. Figure 20 shows the typing judgments for contexts $\mathcal{C}$. Figure 21 shows assorted metafunctions not defined in the main body of this work.

# B Appendix: Open World Soundness

The full mechanized proof of open-world soundness for Anthill and µPython is presented in the Coq files `anthill1.v`, `anthill2.v`, `anthill3.v`, and `anthill4.v`. The semantics of the mechanized proof differ from those presented here in that the reduction is single-step without evaluation contexts, and the receiver is the last parameter of constructors and methods, rather than the first. The Coq proof also does not use a heap type $\Sigma$, but instead uses the heap directly in the µPython typechecking relation and the theorems (but ignoring the values of object fields).

This section contains proof sketches of the key lemmas and most important technical lemmas, as well as the proof of open-world soundness.

▶ Technical Lemma 1 (Environment weakening). If $\Gamma;\Sigma \vdash e : S$ and $\Gamma \subseteq \Gamma'$ then $\Gamma';\Sigma \vdash e : S$.

*Proof Sketch.* Straightforward induction on $\Gamma;\Sigma \vdash e : S$. ◀

▶ Main Lemma 1 (Anthill translation preserves typing).

- If $\Gamma \vdash t \rightsquigarrow e : A$, then $\lfloor\Gamma\rfloor;\emptyset \vdash e : \lfloor A\rfloor$.
- If $\Gamma \vdash c \rightsquigarrow e : \overline{A}$, then $\lfloor\Gamma\rfloor;\emptyset \vdash e : |\overline{A}|\rightarrow$ .
- If $\Gamma;A_1 \vdash d \rightsquigarrow e : A_2$, then $\lfloor\Gamma\rfloor;\emptyset \vdash e : \lfloor A_2\rfloor$

*Proof Sketch.* By induction on $\Gamma \vdash t \rightsquigarrow e : A$, $\Gamma \vdash c \rightsquigarrow e : \overline{A}$, and $\Gamma;A_1 \vdash d \rightsquigarrow e : A_2$. In the IFun, IMethod, and IConstruct cases, use Technical Lemma 1. ◀

▶ Technical Lemma 2. If $\Sigma \vdash \mu$ and $\emptyset;\Sigma \vdash a : S$, then $a \in dom(\Sigma)$.

*Proof Sketch.* By induction on $\emptyset;\Sigma \vdash a : S$. ◀

▶ Technical Lemma 3 (Canonical forms). If $\emptyset;\Sigma \vdash v : S$ and $\Sigma \vdash \mu$, then:

- If $S = \mathsf{int}$, then $v = n$.
- If $S = n\rightarrow$ , then either
  - $v = \lambda\overline{x}.e$ and $|\overline{x}| = n$, or
  - $v = a$ and $\mu(a) = \texttt{Class}(\overline{a'})\{\texttt{init} = v'; M\}$ and $\mathit{param\text{-}match}(v', \mu, n)$
- If $S = \mathsf{class}\ (\!|\delta, C|\!)$, then $v = a$ and
  $\mu(a) = \texttt{Class}(\overline{a'})\{\texttt{init} = v'; M\}$ and
  $\mathit{param\text{-}match}(v', \mu, C)$ and $\mathit{hasattrs}(a, \delta, \mu)$.
- If $S = \mathsf{object}\ (\!|\delta|\!)$ then $v = a$ and either
  - $\mu(a) = \texttt{Class}(\overline{a'})\{\texttt{init} = v'; M\}$ and
    $\mathit{hasattrs}(a, \delta, \mu)$, or
  - $\mu(a) = \texttt{Object}(a)\{M\}$ and $\mathit{hasattrs}(a, \delta, \mu)$.
- If $S = \mathsf{pyobj}$ then for some $S' \neq \mathsf{pyobj}$, $S' <: S$, $\emptyset;\Sigma \vdash v : S'$

*Proof Sketch.* By induction on $\emptyset;\Sigma \vdash v : S$. In case TSubsump, proceeding by cases on $S$ and frequently using Technical Lemma 2. ◀

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

$$\text{(IVar)}\ \frac{\Gamma(x) = A}{\Gamma \vdash x \rightsquigarrow x : A} \qquad \text{(IInt)}\ \frac{}{\Gamma \vdash n \rightsquigarrow n : \texttt{int}} \qquad \text{(ILet)}\ \frac{\Gamma \vdash t_1 \rightsquigarrow e_1 : A_1 \qquad \Gamma, x{:}A_1 \vdash t_2 \rightsquigarrow e_2 : A_2}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 \rightsquigarrow \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : A_2}$$

$$\text{(IGet)}\ \frac{\Gamma \vdash t \rightsquigarrow e : A_1 \qquad \Delta = \mathit{mems}(A_1) \qquad \Delta(x) = A_2}{\Gamma \vdash t.x \rightsquigarrow e.x^\circ \Downarrow_{\lfloor A_2 \rfloor} : A_2}$$

$$\text{(IGet-Check)}\ \frac{\Gamma \vdash t \rightsquigarrow e : A_1 \qquad \Delta = \mathit{mems}(A_1) \qquad x \notin \mathit{dom}(A_1) \qquad \mathit{queryable}(A_1) = \Diamond}{\Gamma \vdash t.x \rightsquigarrow (e \Downarrow_{\mathsf{object}\ (\!|x|\!)}).x^\circ : \star}$$

$$\text{(ISet)}\ \frac{\Gamma \vdash t_1 \rightsquigarrow e_1 : A_1 \qquad \Gamma \vdash t_2 \rightsquigarrow e_2 : A_2' \qquad \Delta = \mathit{mems}(A_1)}{\Gamma \vdash t_1.x = t_2 \rightsquigarrow e_1.x^\circ = e_2 : \mathsf{int}}$$

$$\text{(ISet-Check)}\ \frac{\Gamma \vdash t_1 \rightsquigarrow e_1 : A_1 \qquad \Gamma \vdash t_2 \rightsquigarrow e_2 : A_2 \qquad \Delta = \mathit{mems}(A_1) \qquad x \notin \mathit{dom}(A_1) \qquad \mathit{queryable}(A_1) = \Diamond}{\Gamma \vdash t_1.x = t_2 \rightsquigarrow (e_1 \Downarrow_{\mathsf{object}\ (\!|\emptyset|\!)}).x^\circ = e_2 : \mathsf{int}}$$

$$\text{(IFun)}\ \frac{\Gamma, \overline{x : A_1} \vdash t \rightsquigarrow e : A_2' \qquad A_2' \lesssim A_2}{\Gamma \vdash \lambda \overline{x{:}A_1} \to A_2.\ t \rightsquigarrow \lambda \overline{x}.\ \overline{\texttt{let}\ x = x \Downarrow_{\lfloor A_1 \rfloor}\ \texttt{in}}\ e : \overline{A_1} \to A_2}$$

$$\text{(IApp-Dyn)}\ \frac{\Gamma \vdash t_1 \rightsquigarrow e_1 : \star \qquad \Gamma \vdash \overline{t_2 \rightsquigarrow e_2 : A}}{\Gamma \vdash t_1(\overline{t_2}) \rightsquigarrow (e_1 \Downarrow_{|\overline{A}| \to})(\overline{e_2})^\circ : \star}$$

$$\text{(IApp-Fun)}\ \frac{\Gamma \vdash t_1 \rightsquigarrow e_1 : \overline{A_1} \to A_2 \qquad \Gamma \vdash \overline{t_2 \rightsquigarrow e_2 : A_1'} \qquad |\overline{A_1}| = |\overline{t_2}| \qquad \overline{A_1' \lesssim A_1}}{\Gamma \vdash t_1(\overline{t_2}) \rightsquigarrow (e_1(\overline{e_2})^\circ) \Downarrow_{\lfloor A_2 \rfloor} : A_2}$$

$$\text{(IApp-Constr)}\ \frac{\Gamma \vdash t_1 \rightsquigarrow e_1 : \mathsf{class}\ (\!|q, \Delta_1, \Delta_2, \overline{A_1}|\!) \qquad \Gamma \vdash \overline{t_2 \rightsquigarrow e_2 : A_1'} \qquad |\overline{A_1}| = |\overline{t_2}| \qquad \overline{A_1' \lesssim A_1} \qquad A_2 = \mathsf{object}\ (\!|q, \mathit{instantiate}(\Delta_1, \Delta_2)|\!)}{\Gamma \vdash t_1(\overline{t_2}) \rightsquigarrow (e_1(\overline{e_2})^\circ) \Downarrow_{\lfloor A_2 \rfloor} : A_2}$$

$$\text{(IClass)}\ \frac{\begin{array}{c}\Gamma \vdash \overline{t_s \rightsquigarrow e_s : A_s} \qquad A_{class} = \mathsf{class}\ (\!|q, \Delta_1, \Delta_2, \overline{A_c}|\!) \qquad \Gamma \vdash_\sigma c \rightsquigarrow e_c : \overline{A_c} \\ \Gamma; A_{class} \vdash_\varsigma \overline{m \rightsquigarrow e_m : A_m} \qquad \Gamma \vdash \overline{t_f \rightsquigarrow e_f : A_f} \qquad \overline{e_s' = e_s \Downarrow_{\mathsf{class}\ (\!|\lfloor \mathit{mems}(A_s) \rfloor, \mathsf{Any}|\!)}} \\ \forall x \in \mathit{dom}(\Delta_1),\ (\overline{\ell_f} \times \overline{A_f} \cup \overline{\ell_m} \times \overline{A_m} \cup \overline{\mathit{mems}(A_s)})(x) \lesssim \Delta_1(x)\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{class}(\overline{t_s}){:}(\!|q, \Delta_1, \Delta_2|\!)\ \texttt{init}{=}c; \overline{\ell_f =^M m}; \overline{\ell_m =^F t_m} \rightsquigarrow \\ \texttt{class}(\overline{e_s'})\texttt{:}\ \texttt{init} = e_c; \overline{\ell_m = e_m}, \overline{\ell_f = e_f} : A_{class}\end{array}}$$

$\boxed{\Gamma \vdash_\sigma c \rightsquigarrow e : \overline{A}}$

$$\text{(IConstruct)}\ \frac{\Gamma, x_s{:}\star, \overline{x{:}A_1} \vdash t \rightsquigarrow e : A_2}{\Gamma \vdash_\sigma \sigma\ x_s, \overline{x{:}A_1}.\ t \rightsquigarrow \lambda x_s, \overline{x}.\ \overline{\texttt{let}\ x = x \Downarrow_{\lfloor A_1 \rfloor}\ \texttt{in}}\ e : \overline{A_1}}$$

$\boxed{\Gamma; A \vdash_\varsigma m \rightsquigarrow e : A}$

$$\text{(IMethod)}\ \frac{A_o = \mathsf{object}\ (\!|q, \mathit{instantiate}(\Delta_1, \Delta_2)|\!) \qquad A_2' \lesssim A_2 \qquad \Gamma, x_s{:}A_o, \overline{x{:}A_1} \vdash t \rightsquigarrow e : A_2'}{\begin{array}{c}\Gamma; \mathsf{class}\ (\!|q_1, \Delta_1, \Delta_2, \overline{A_c}|\!) \vdash_\varsigma \varsigma\ x_s, \overline{x{:}A_1} \to A_2.\ t \rightsquigarrow \\ \lambda x_s, \overline{x}.\ \texttt{let}\ x_s = x_s \Downarrow_{\lfloor A_o \rfloor}\ \texttt{in}\ \overline{\texttt{let}\ x = x \Downarrow_{\lfloor A_1 \rfloor}\ \texttt{in}}\ e : \overline{A_1} \to A_2\end{array}}$$

**Figure 16** Translation from Anthill Python to µPython (including origin)

$\boxed{\Gamma;\Sigma \vdash e : S}$

$$\frac{\Gamma;\Sigma \vdash e : S_2 \qquad S_2 <: S_1}{\Gamma;\Sigma \vdash e : S_1}\ (\textsc{TSubsump}) \qquad \frac{\Gamma(x) = S}{\Gamma;\Sigma \vdash x{:}S}\ (\textsc{TVar}) \qquad \frac{\Sigma(a) = S}{\Gamma;\Sigma \vdash a{:}S}\ (\textsc{TAddr}) \qquad \frac{}{\Gamma;\Sigma \vdash n{:}\mathsf{int}}\ (\textsc{TInt})$$

$$\frac{\Gamma;\Sigma \vdash e_1 : \mathsf{pyobj} \qquad \Gamma;\Sigma \vdash \overline{e_2 : \mathsf{pyobj}}}{\Gamma;\Sigma \vdash e_1(\overline{e_2})^{\bullet} : \mathsf{pyobj}}\ (\textsc{TApp-Dyn}) \qquad \frac{\Gamma;\Sigma \vdash e_1 : n{\to} \qquad \Gamma;\Sigma \vdash \overline{e_2 : \mathsf{pyobj}} \qquad |\overline{e_2}| = n}{\Gamma;\Sigma \vdash e_1(\overline{e_2})^{\circ} : \mathsf{pyobj}}\ (\textsc{TApp})$$

$$\frac{\Gamma;\Sigma \vdash e : \mathsf{pyobj}}{\Gamma;\Sigma \vdash e.x^{\bullet} : \mathsf{pyobj}}\ (\textsc{TGet-Dyn}) \qquad \frac{\Gamma;\Sigma \vdash e : \mathsf{object}\ (\!|x|\!)}{\Gamma;\Sigma \vdash e.x^{\circ} : \mathsf{pyobj}}\ (\textsc{TGet}) \qquad \frac{\Gamma;\Sigma \vdash e_1 : \mathsf{pyobj} \qquad \Gamma;\Sigma \vdash e_2 : \mathsf{pyobj}}{\Gamma;\Sigma \vdash e_1.x^{\bullet} = e_2 : \mathsf{int}}\ (\textsc{TSet-Dyn})$$

$$\frac{\Gamma;\Sigma \vdash e_1 : \mathsf{object}\ (\!|\emptyset|\!) \qquad \Gamma;\Sigma \vdash e_2 : \mathsf{pyobj}}{\Gamma;\Sigma \vdash e_1.x^{\circ} = e_2 : \mathsf{int}}\ (\textsc{TSet})$$

$$\frac{\Gamma;\Sigma \vdash \overline{e_s : \mathsf{pyobj}} \qquad \Gamma;\Sigma \vdash \overline{e_m : \mathsf{pyobj}} \qquad \Gamma;\Sigma \vdash e_c : \mathsf{pyobj}}{\Gamma;\Sigma \vdash \mathtt{class}(\overline{e_s}){:}^{\bullet}\ \mathtt{init} = e_c; \overline{x = e_m} : \mathsf{class}\ (\!|\overline{x}, \mathsf{Any}|\!)}\ (\textsc{TClass-Dyn})$$

$$\frac{\Gamma;\Sigma \vdash \overline{e_s : \mathsf{class}\ (\!|\delta, \mathsf{Any}|\!)} \qquad \delta' = \bigcup \overline{\delta} \qquad \Gamma;\Sigma \vdash \overline{e_m : \mathsf{pyobj}} \qquad \Gamma;\Sigma \vdash e_c : (n{+}1){\to}}{\Gamma;\Sigma \vdash \mathtt{class}(\overline{e_s}){:}^{\circ}\ \mathtt{init} = e_c; \overline{x = e_m} : \mathsf{class}\ (\!|\overline{x} \cup \delta', n|\!)}\ (\textsc{TClass})$$

$$\frac{\Gamma, \overline{x{:}\mathsf{pyobj}}; \Sigma \vdash e : \mathsf{pyobj}}{\Gamma;\Sigma \vdash \lambda\overline{x}.\ e : |\overline{x}|{\to}}\ (\textsc{TFun}) \qquad \frac{\Gamma;\Sigma \vdash e : \mathsf{pyobj}}{\Gamma;\Sigma \vdash e \Downarrow_S : S}\ (\textsc{TCheck}) \qquad \frac{\Gamma;\Sigma \vdash e_1 : S_1 \qquad \Gamma, x{:}S_1;\Sigma \vdash e_2 : S_2}{\Gamma;\Sigma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : S_2}\ (\textsc{TLet})$$

$\boxed{\Sigma;\mu \vdash a : S}$

$$\frac{\mu(a) = \mathtt{Class}(\overline{a'})\{\mathtt{init} = v; \overline{x_f = v_f}\} \qquad \mathit{hasattrs}(a, \delta, \mu) \qquad \mathit{param\text{-}match}(a, \mu, C) \qquad \overline{\Sigma(a') = \mathsf{class}\ (\!|\delta', C'|\!)} \qquad \overline{\emptyset;\Sigma \vdash v_f : \mathsf{pyobj}}}{\Sigma;\mu \vdash a : \mathsf{class}\ (\!|\delta, C|\!)}\ (\textsc{THClass})$$

$$\frac{\mu(a) = \mathtt{Object}(a')\{\overline{x_f = v_f}\} \qquad \mathit{hasattrs}(a, \delta, \mu) \qquad \Sigma(a') = \mathsf{class}\ (\!|\delta', C'|\!) \qquad \overline{\emptyset;\Sigma \vdash v_f : \mathsf{pyobj}}}{\Sigma;\mu \vdash a : \mathsf{object}\ (\!|\delta|\!)}\ (\textsc{THObject})$$

$\boxed{\Sigma \vdash \mu}$

$$\frac{\mathit{dom}(\Sigma) = \mathit{dom}(\mu) \qquad \forall a \in \mathit{dom}(\Sigma).\ \Sigma;\mu \vdash a : \Sigma(a)}{\Sigma \vdash \mu}$$

**Figure 17** Type system for µPython (including origin)

$\boxed{lookup(a, h, \ell, \mu, p) = r}$

$$\frac{M(\ell) = v}{lookup(a, \texttt{Object}(a')\{M\}, \ell, \mu, p) = v \mid \mu} \qquad \frac{\ell \notin dom(M) \quad getattr(a, \ell, \mu) = v \quad v \neq \lambda\bar{\ell}.e}{lookup(a, \texttt{Object}(a')\{M\}, \ell, \mu, p) = v \mid \mu}$$

$$\frac{\ell \notin dom(M) \quad getattr(a, \ell, \mu) = v \quad v = \lambda\overline{x}.e \quad |\overline{y}| = |\overline{x}| - 1}{lookup(a, \texttt{Object}(a')\{M\}, \ell, \mu, p) = \lambda\overline{y}.v(a, \overline{y})^p \mid \mu}$$

$$\frac{\ell \notin dom(M) \quad getattr(a, \ell, \mu) = v \quad v = \lambda\overline{x}.e \quad |\overline{x}| = 0}{lookup(a, \texttt{Object}(a')\{M\}, \ell, \mu, p) = \texttt{casterror}}$$

$$\frac{getattr(a, \ell, \mu) = v}{lookup(a, \texttt{Class}(\overline{a'})\{\texttt{init} = v'; M\}, \ell, \mu, p) = v \mid \mu}$$

**Figure 18** Relations used in µPython evaluation, with origin labels

▶ Technical Lemma 4 (Heap weakening). If $\Gamma;\Sigma \vdash e : S$ and $\Sigma \sqsubseteq \Sigma'$ then $\Gamma;\Sigma' \vdash e : S$.

*Proof Sketch.* By induction on $\Gamma;\Sigma \vdash e : S$, subsuming class types to object types when necessary. ◀

▶ Technical Lemma 5. If $\Sigma \vdash \mu$ and $\Sigma; \mu \vdash a : S$ and $hasattrs(a, \delta', \mu)$, then

- If $S = \mathsf{class}\ (\!|\delta, C|\!)$, then $\delta' \subseteq \delta$.
- If $S = \mathsf{object}\ (\!|\delta|\!)$, then $\delta' \subseteq \delta$.

*Proof Sketch.* By induction on $\Sigma; \mu \vdash a : S$. ◀

▶ Technical Lemma 6. If $\Sigma \vdash \mu$ and $\emptyset; \Sigma \vdash v : S$ and $param\text{-}match(v, \mu, n)$, then $\emptyset; \Sigma \vdash v : n\!\rightarrow$

*Proof Sketch.* By cases on $v$. ◀

▶ Technical Lemma 7 (Substitution). If $\Gamma, x : S_1;\Sigma \vdash e : S_2$ and $\emptyset;\Sigma \vdash v : S_1$ then $\Gamma;\Sigma \vdash e[x/v] : S_2$.

*Proof Sketch.* Straightforward induction on $\Gamma, x : S_1;\Sigma \vdash e : S_2$. ◀

▶ Main Lemma 2 (Preservation). If $\emptyset;\Sigma \vdash e : S$, $\Sigma \vdash \mu$, and $e \mid \mu \longrightarrow e' \mid \mu'$, then $\emptyset;\Sigma' \vdash e' : S$ and $\Sigma' \vdash \mu'$ and $\Sigma \sqsubseteq \Sigma'$.

*Proof Sketch.* By induction on $e \mid \mu \longrightarrow e' \mid \mu'$. Uses Technical Lemmas 2, 3, 4, 5, 6, and 7. ◀

▶ **Corollary 1** (Iterated preservation). *If* $\emptyset;\Sigma \vdash e : S$, $\Sigma \vdash \mu$, *and* $e \mid \mu \longrightarrow^* e' \mid \mu'$, *then* $\emptyset;\Sigma' \vdash e' : S$ *and* $\Sigma' \vdash \mu'$ *and* $\Sigma \sqsubseteq \Sigma'$.

*Proof Sketch.* Straightforward induction on $e \mid \mu \longrightarrow e' \mid \mu'$. In case MCHAIN, apply Main Lemma 2. ◀

▶ Main Lemma 3 (Progress). If $\emptyset;\Sigma \vdash e : S$ and $\Sigma \vdash \mu$, then either $e$ is a value or $e \mid \mu \longrightarrow r$ and either:

$\boxed{e \mid \mu \longrightarrow r}$

$$\text{(EStep)}\ \frac{e \mid \mu \longrightarrow e' \mid \mu'}{E[e] \mid \mu \longmapsto E[e'] \mid \mu'} \qquad \text{(EPyError)}\ \frac{e \mid \mu \longrightarrow \texttt{pyerror}(p)}{E[e] \mid \mu \longmapsto \texttt{pyerror}(p)} \qquad \text{(ECastError)}\ \frac{e \mid \mu \longrightarrow \texttt{casterror}}{E[e] \mid \mu \longmapsto \texttt{casterror}}$$

$$\text{(ECheck1,2)}\quad v \Downarrow_S \mid \mu \longrightarrow \begin{cases} v \mid \mu & \text{if } \mathit{check}(v, \mu, S) \\ \texttt{casterror} & \text{otherwise} \end{cases}$$

$$\text{(EApp1,2,3)}\quad v_1(\overline{v_2})^p \mid \mu \longrightarrow \begin{cases} e[\overline{x/v_2}] \mid \mu & \text{if } v_1 = \lambda\overline{x}.e \\ & \text{and } |\overline{x}| = |\overline{v_2}| \\ \texttt{let \_ =} & \text{if } v_1 = a \\ v'(a', \overline{v_2})^p & \text{and } \mu(a) = \texttt{Class}(\overline{a''})\{\texttt{init} = v'; M\} \\ \texttt{in } a' \mid \mu' & \text{and } a' \text{ fresh} \\ & \text{and } \mu' = \mu[a' \mapsto \texttt{Object}(a)\{\emptyset\}] \\ \texttt{pyerror}(p) & \text{otherwise} \end{cases}$$

$$\text{(ELet)}\quad \texttt{let } x = v \texttt{ in } e \longrightarrow e[x/v]$$

$$\text{(EClass1,2)}\quad \texttt{class}(\overline{a})\texttt{:}^p\ \texttt{init} = v; M \mid \mu \longrightarrow \begin{cases} a' \mid \mu[a' \mapsto h] & \text{if } \overline{\mu(a) = \texttt{Class}(\overline{a''})\{\texttt{init} = v'; M'\}} \\ & \text{and } \mathit{param\text{-}match}(v, \mu, \mathsf{Any}) \\ & \text{and } h = \texttt{Class}(\overline{a})\{\texttt{init} = v; M\} \\ & \text{and } a' \text{ fresh} \\ \texttt{pyerror}(p) & \text{otherwise} \end{cases}$$

$$\text{(EClass3)}\quad \texttt{class}(\overline{v_1})\texttt{:}^p\ \texttt{init} = v_2; M \mid \mu \longrightarrow \texttt{pyerror}(p) \quad \text{if } \overline{v_1 \neq a}$$

$$\text{(EGet1,2)}\quad a.\ell^p \mid \mu \longrightarrow \begin{cases} r & \text{if } \mathit{lookup}(a, \mu(a), \ell, \mu) = r \\ \texttt{pyerror}(p) & \text{otherwise} \end{cases}$$

$$\text{(EGet3)}\quad v.\ell^p \mid \mu \longrightarrow \texttt{pyerror}(p) \quad \text{if } v \neq a$$

$$\text{(ESet1,2,3)}\quad a.\ell^p = v \mid \mu \longrightarrow \begin{cases} 0 \mid \mu[a \mapsto h'] & \text{if } \mu(a) = \texttt{Object}(a')\{M\} \\ & \text{and } h' = \texttt{Object}(a')\{M[\ell = v]\} \\ 0 \mid \mu[a \mapsto h'] & \text{if } \mu(a) = \texttt{Class}(\overline{a'})\{\texttt{init} = v'; M\} \\ & \text{and } h' = \texttt{Class}(\overline{a'})\{\texttt{init} = v'; M[\ell = v]\} \\ \texttt{pyerror}(p) & \text{otherwise} \end{cases}$$

$$\text{(ESet4)}\quad v_1.\ell^p = v_2 \mid \mu \longrightarrow \texttt{pyerror}(p) \quad \text{if } v_1 \neq a$$

$\boxed{e \mid \mu \longrightarrow^* r}$

$$\text{(MRefl)}\ \frac{}{e \mid \mu \longrightarrow^* e \mid \mu} \qquad \text{(MPyErr)}\ \frac{e \mid \mu \longmapsto \texttt{pyerror}(p)}{e \mid \mu \longrightarrow^* \texttt{pyerror}(p)}$$

$$\text{(MCastErr)}\ \frac{e \mid \mu \longmapsto \texttt{casterror}}{e \mid \mu \longrightarrow^* \texttt{casterror}} \qquad \text{(MChain)}\ \frac{e \mid \mu \longmapsto e' \mid \mu' \qquad e' \mid \mu' \longrightarrow^* r}{e \mid \mu \longrightarrow^* r}$$

**Figure 19** µPython evaluation rules, with origin labels

$\boxed{\mathcal{C} : \Gamma;S \Rightarrow \Gamma;S}$

$$\frac{}{\Box : \Gamma;S \Rightarrow \Gamma;S} \qquad \frac{\mathcal{C} : \Gamma;S_1 \Rightarrow \Gamma';S_3 \qquad S_3 <: S_2}{\mathcal{C} : \Gamma;S_1 \Rightarrow \Gamma';S_2} \qquad \frac{\mathcal{C} : \Gamma;S \Rightarrow \Gamma';\mathsf{pyobj} \qquad \Gamma';\emptyset \vdash \overline{e : \mathsf{pyobj}}}{\mathcal{C}(\overline{e})^\bullet : \Gamma;S \Rightarrow \Gamma';\mathsf{pyobj}}$$

$$\frac{\Gamma';\emptyset \vdash e : \mathsf{pyobj} \qquad \Gamma';\emptyset \vdash \overline{e : \mathsf{pyobj}} \qquad \mathcal{C} : \Gamma;S \Rightarrow \Gamma';\mathsf{pyobj}}{e(\overline{e}, \mathcal{C}, \overline{e})^\bullet : \Gamma;S \Rightarrow \Gamma';\mathsf{pyobj}} \qquad \frac{\mathcal{C} : \Gamma;S \Rightarrow \Gamma';\mathsf{pyobj}}{\mathcal{C}.x^\bullet : \Gamma;S \Rightarrow \Gamma';\mathsf{pyobj}}$$

$$\frac{\mathcal{C} : \Gamma;S \Rightarrow \Gamma';\mathsf{pyobj} \qquad \Gamma';\emptyset \vdash e : \mathsf{pyobj}}{\mathcal{C}.x^\bullet = e : \Gamma;S \Rightarrow \Gamma';\mathsf{int}} \qquad \frac{\Gamma';\emptyset \vdash e : \mathsf{pyobj} \qquad e : \Gamma;S \Rightarrow \Gamma';\mathsf{pyobj}}{\Gamma;\Sigma \vdash e.x^\bullet = \mathcal{C} : \Gamma;S \Rightarrow \Gamma';\mathsf{int}}$$

$$\frac{\Gamma';\emptyset \vdash \overline{e_s : \mathsf{pyobj}} \qquad \Gamma';\emptyset \vdash \overline{e_m : \mathsf{pyobj}} \qquad \Gamma';\emptyset \vdash e_c : \mathsf{pyobj} \qquad \mathcal{C} : \Gamma;S \Rightarrow \Gamma';\mathsf{pyobj}}{\texttt{class}(\overline{e_s}, \mathcal{C}, \overline{e_s})\texttt{:}^\bullet\ \texttt{init} = e_c; \overline{x = e_m} : \Gamma;S \Rightarrow \Gamma';\mathsf{pyobj}}$$

$$\frac{\Gamma';\emptyset \vdash \overline{e_s : \mathsf{pyobj}} \qquad \Gamma';\emptyset \vdash \overline{e_m : \mathsf{pyobj}} \qquad \Gamma';\emptyset \vdash e_c : \mathsf{pyobj} \qquad \mathcal{C} : \Gamma;S \Rightarrow \Gamma';\mathsf{pyobj}}{\texttt{class}(\overline{e_s})\texttt{:}^\bullet\ \texttt{init} = e_c; \overline{x = e_m}, x = \mathcal{C}, \overline{x = e_m} : \Gamma;S \Rightarrow \Gamma';\mathsf{pyobj}}$$

$$\frac{\Gamma';\emptyset \vdash \overline{e_s : \mathsf{pyobj}} \qquad \Gamma';\emptyset \vdash \overline{e_m : \mathsf{pyobj}} \qquad \mathcal{C} : \Gamma;S \Rightarrow \Gamma';\mathsf{pyobj}}{\texttt{class}(\overline{e_s})\texttt{:}^\bullet\ \texttt{init} = \mathcal{C}; \overline{x = e_m} : \Gamma;S \Rightarrow \Gamma';\mathsf{pyobj}} \qquad \frac{\mathcal{C} : \Gamma;S \Rightarrow \Gamma', \overline{x{:}\mathsf{pyobj}};\mathsf{pyobj}}{\lambda\overline{x}.\ \mathcal{C} : \Gamma;S \Rightarrow \Gamma';|\overline{x}|\rightarrow}$$

$$\frac{\mathcal{C} : \Gamma;S \Rightarrow \Gamma';\mathsf{pyobj}}{\mathcal{C} \Downarrow_{S'} : \Gamma;S \Rightarrow \Gamma';S'}$$

$$\frac{\mathcal{C} : \Gamma;S_1 \Rightarrow \Gamma';S_2 \qquad \Gamma', x{:}S_2;\emptyset \vdash e : S_3}{\texttt{let}\ x = \mathcal{C}\ \texttt{in}\ e : \Gamma;S_1 \Rightarrow \Gamma';S_3} \qquad \frac{\Gamma';\emptyset \vdash e : S_2 \qquad \mathcal{C} : \Gamma;S_1 \Rightarrow \Gamma', x{:}S_2;S_3}{\texttt{let}\ x = e\ \texttt{in}\ \mathcal{C} : \Gamma;S_1 \Rightarrow \Gamma';S_3}$$

**Figure 20** Type system for μPython contexts

- $r = \texttt{pyerror}(\bullet)$, or
- $r = \texttt{casterror}$, or
- $r = e' \mid \mu'$.

*Proof Sketch.* Induction on $\emptyset;\Sigma \vdash e : S$. In the -Dyn cases, this lemma is trivial, since the catch-all evaluation rules that lead to $\texttt{pyerror}(\bullet)$ satisfy the lemma. ◀

▶ Main Lemma 4 (Composition). If $\mathcal{C} : \Gamma;S \Rightarrow \Gamma';S'$ and $\Gamma;\emptyset \vdash e : S$, then $\Gamma';\emptyset \vdash \mathcal{C}[e] : S'$.

*Proof Sketch.* By induction on $\mathcal{C} : \Gamma;S \Rightarrow \Gamma';S'$. ◀

▶ Technical Lemma 8 (Typed code blames $\bullet$). If $e \mid \mu \longrightarrow \texttt{pyerror}(p)$ and $\emptyset;\Sigma \vdash e : S$ and $\Sigma \vdash \mu$, then $p = \bullet$.

*Proof Sketch.* Straightforward induction on $e \mid \mu \longrightarrow \texttt{pyerror}(p)$, noting that $\circ$-labeled code cannot take steps that lead to `pyerror`. ◀

▶ **Corollary 2.** *If $e \mid \mu \longrightarrow^* \texttt{pyerror}(p)$ and $\emptyset;\Sigma \vdash e : S$ and $\Sigma \vdash \mu$, then $p = \bullet$.*

*Proof Sketch.* Induction on $e \mid \mu \longrightarrow^* \texttt{pyerror}(p)$, applying Technical Lemma 8 in the case for EChain. ◀

▶ **Theorem 1** (Open world soundness)**.** *If $\Gamma \vdash t \rightsquigarrow e : A$, then for any context $\mathcal{C}$ such that $\mathcal{C} : \lfloor\Gamma\rfloor;\lfloor A\rfloor \Rightarrow \emptyset;S$, then $\emptyset;\emptyset \vdash \mathcal{C}[e] : A$ and either:*

- *for some $v, \Sigma, \mu$, $\mathcal{C}[e] \mid \emptyset \longrightarrow^* v \mid \mu$ and $\emptyset;\Sigma \vdash v : S$ and $\Sigma \vdash \mu$, or*
- *$\mathcal{C}[e] \mid \emptyset \longrightarrow^* \texttt{pyerror}(\bullet)$, or*
- *$\mathcal{C}[e] \mid \emptyset \longrightarrow^* \texttt{casterror}$, or*
- *for all $r$ such that $\mathcal{C}[e] \mid \emptyset \longrightarrow^* r$, have $r = e' \mid \mu'$ and there exists $r'$ such that $e' \mid \mu' \longrightarrow r'$.*

**Proof.** From Main Lemma 1, $\lfloor\Gamma\rfloor;\emptyset \vdash e : \lfloor A\rfloor$. Then from Main Lemma 4, $\emptyset;\emptyset \vdash \mathcal{C}[e] : S$. Either $\mathcal{C}[e] \mid \emptyset$ diverges or it does not. If it does, the theorem is satisfied immediately; otherwise, we have some $r$, $\mathcal{C}[e] \mid \emptyset \longrightarrow^* r$, such that either $r \neq e' \mid \mu'$ or $\nexists r', e' \mid \mu' \longrightarrow r'$. In the latter case, by Main Lemma 3, either $e'$ is a value or there exists some $r'$, $e' \mid \mu' \longrightarrow r'$. The latter is a contradiction, and so $e'$ is a value. By Corollary 1, $\emptyset;\Sigma' \vdash e : S$ and $\Sigma \vdash \mu'$ for some $\Sigma'$.

If $r = \texttt{casterror}$, the theorem is satisfied. If $r = \texttt{pyerror}(p)$, by Corollary 2, the theorem is satisfied. ◀

$\boxed{\mathit{param\text{-}match}(v, \mu, C)}$

$$\mathit{param\text{-}match}(\lambda\overline{x}.e, \mu, \mathsf{Any}) \qquad \frac{|\overline{x}| = n}{\mathit{param\text{-}match}(\lambda\overline{x}.e, \mu, n)} \qquad \frac{\mu(a) = \mathtt{Class}(\overline{a})\{\mathtt{init} = v; M\}}{\mathit{param\text{-}match}(a, \mu, \mathsf{Any})}$$

$$\frac{\mu(a) = \mathtt{Class}(\overline{a})\{\mathtt{init} = v; M\} \qquad \mathit{param\text{-}match}(v, \mu, n+1)}{\mathit{param\text{-}match}(a, \mu, n)}$$

$\boxed{\mathit{getattr}(a, x, \mu) = v}$

$$\mathit{getattr}(a, x, \mu) = \begin{cases} M(x) & \text{if } \mu(a') = \mathtt{Object}(a')\{M\} \\ & \text{and } x \in \mathit{dom}(M) \\ \mathit{getattr}(a', x, \mu) & \text{if } \mu(a) = \mathtt{Object}(a')\{M\} \\ & \text{and } x \notin \mathit{dom}(M) \\ M(x) & \text{if } \mu(a') = \mathtt{Class}(\overline{a'})\{\mathtt{init} = v; M\} \\ & \text{and } x \in \mathit{dom}(M) \\ \mathit{getattr}(a'_k, x, \mu) & \text{if } \mu(a) = \mathtt{Class}(a'_0, \ldots, a'_k, \ldots, a'_n)\{\mathtt{init} = v; M\} \\ & \text{and } x \notin \mathit{dom}(M) \\ & \text{and } \forall i, 0 \le i < k, \mathit{getattr}(a'_i, x, \mu) \neq v \end{cases}$$

$\boxed{\mathit{hasattr}(a, x, \mu)}$ $\qquad$ $\boxed{\mathit{hasattrs}(a, \delta, \mu)}$

$$\frac{\mathit{getattr}(a, x, \mu) = v}{\mathit{hasattr}(a, x, \mu)} \qquad \frac{\forall x \in \delta.\ \mathit{getattr}(a, x, \mu)}{\mathit{hasattrs}(a, \delta, \mu)}$$

$\boxed{\Sigma \sqsubseteq \Sigma}$

$$\frac{\forall a \in \mathit{dom}(\Sigma_1).\ \Sigma_2(a) <: \Sigma_1(a)}{\Sigma_1 \sqsubseteq \Sigma_2}$$

$\boxed{A \sim A}$

$$\star \sim A \qquad A \sim \star \qquad \mathtt{int} \sim \mathtt{int} \qquad \frac{\Delta_1 \sim \Delta_2}{\mathsf{object}\ (\!|q_1, \Delta_1|\!) \sim \mathsf{object}\ (\!|q_2, \Delta_2|\!)}$$

$$\frac{\Delta_1 \sim \Delta_3 \qquad \Delta_2 \sim \Delta_4 \qquad |A_1| = |A_3| \qquad \overline{A_1 \sim A_3}}{\mathsf{class}\ (\!|q_1, \Delta_1, \Delta_2, \overline{A_1}|\!) \sim \mathsf{class}\ (\!|q_2, \Delta_3, \Delta_4, \overline{A_2}|\!)}$$

$$\frac{|A_1| = |A_3| \qquad \overline{A_1 \sim A_3} \qquad A_2 \sim A_4}{\overline{A_1} \to A_2 \sim \overline{A_3} \to A_4}$$

**Figure 21** Other metafunctions