Wesleyan University The Honors College

Charming Python with Static Typechecking

by

Jeff Ruberg
Class of 2012

A thesis submitted to the
faculty of Wesleyan University
in partial fulfillment of the requirements for the
Degree of Bachelor of Arts
with Departmental Honors in Mathematics and Computer Science

Middletown, Connecticut April 22, 2012

Acknowledgements

Thank you to my advisor, Norman Danner, for introducing me to a fascinating subject, guiding me through interesting work, and always driving me to achieve more.

Thank you to my parents, Lewis Ruberg and Stephanie Phillips, for supporting me through the years and all the obvious reasons.

Thank you to my girlfriend, Ashley McDonnell, for encouraging me to do more, for helping me escape when I've tried to do too much, and for putting up with my frustration while dealing with bugs.

Abstract

Pyty is a static typechecker for the Python programming language. Pyty imposes a static type system on users, for which they provide type declarations via comments. The type system developed is modeled on the Hindley-Milner type inference algorithm for the lambda calculus and its application to ML. Much of the implementation challenge is in applying this type system to a language not designed with typechecking in mind. The type system describes a system of deriving type assignments, whereby we can prove that a Python term can be assigned a specific type; we do this by searching for a valid tree of type assignment rules. Pyty also employs type inference for a restricted subset of the Python language in order to facilitate the typechecking algorithm.

Contents

CHAPTER 1

Introduction

1. Motivation

Python is a dynamic language which strongly emphasizes flexibility and readability. Many Python developers find the language's dynamic typing desirable, since it offers fewer restrictions and less verbose code than more static languages like C or Java. This makes the language appealing to developers of a recent movement that place developer time and ease as the highest priorities (over, for example, security or enterprise support) in determining a choice of programming environment, and has garnered more support for dynamic scripting languages such as Python, Ruby, and Perl.

However, Python's flexibility comes at a serious cost that is often overlooked: long-running Python programs can reach critical bugs (that may be as trivial as a typo, yet terminate execution) several hours or days into execution or after partial manipulation of sensitive data. Wrigstad [2010], in an announcement for a conference focused on improving type systems in scripting languages, argues that these compromises made to optimize development time make it harder to reason about correctness and navigate code and often result in eventual necessity to port code to other languages. The Python development community has created various static analysis tools to mitigate this issue (though none detect errors through type analysis), and we have surveyed some of the most popular tools in Chapter 2.

Although Python emphasizes flexibility, the language's community tends to stabilize on certain idioms and behavior as the "pythonic" way, and discourage "unpythonic" code. In fact, as we will discuss in Chapter 2, Python has a recommended style guide which much of the community follows (although one of the points of the style guide is to disregard the style guide if necessary). So, although Python allows very nonstandard usage of types, it is not out of the question to develop a tool which requires a sane type usage—many

```
1  def fib_r(n, i, mem): #: fib_r : (int, int, (int, int)) -> int
2      mem1 = mem[0]     #: mem1 : int
3      mem2 = mem[1]     #: mem2 : int
4
5      if i == n == 0:
6          return 0
7      elif i == n:
8          return mem1
9      else:
10         return fib_r(n, i + 1, (mem2, mem1 + mem2))
11
12 def fib(n):            #: fib : int -> int
13     return fib_rec(n, 0, (0,1))
```

FIGURE 1. Program which has been annotated for analysis by Pyty.

developers likely already write code in this way for the sake of clarity and are also familiar with adhering to a standardized style to detect errors and ensure readability of code to other developers.

As we will discuss in Chapter 2, there have been previous attempts at type inference in Python, but these have modeled types by simulating the program's data flow—with the ultimate goal to improve compiler efficiency. Our goal, however, is to detect bugs and instances where code does not meet developers' expectations, so a typechecking system modeled after Hindley-Milner type inference with explicit type declarations is most fitting.

2. Pyty Description

Pyty is a tool which developers run on source code which has been annotated with type declarations; Pyty then determines whether the provided module typechecks correctly. The type grammar specifiable to the user in type declarations is mostly straightforward, and outlined in detail in Chapter 3, Section 2.2. Figure 1 shows an example Python module which has been annotated with type declarations and which typechecks correctly.

3. Overview

In Chapter 2, we survey some of the existing literature on typechecking and type inference in the Python domain. In Chapter 3, we lay out the type system designed for Pyty. In

Chapter 4, we discuss the implementation behind the typechecking algorithm. Finally, in Chapter 5, we provide a summary of the results of the research and possible avenues for further research.

CHAPTER 2

Literature Review

1. Adding Optional Static Typing to Python

In 2004, van Rossum [2004], creator and maintainer of Python, described his thoughts on optional static typing in Python through a series of four blog posts. In these posts directed towards the Python development community, van Rossum publicly shared his ideas on design and implementation of the feature. Many Python users admire the language for its dynamic nature and the flexibility that allows. But van Rossum showed frustration at the large class of errors that are easily caught at compile-time in statically typed languages but may not be caught until hours or days of execution in Python. By 2004, third parties had explored the implications of mixing static typing and Python; however, these blog posts were van Rossum's attempt to think through the details of building such support into the language. Van Rossum released his thoughts in blog form to receive feedback and iterate on his ideas in future posts.

The series of blog posts received a large quantity of negative feedback. Because so many users love Python for the dynamism, many were equally upset by the idea of adding static typing—even if the feature were completely optional. The first blog post surveyed several issues and proposed scattered ideas, but the negative feedback caused van Rossum to return with a more coherent and cohesive proposal in his second post. At this point, despite the negative feedback, van Rossum insisted that optional static typing was a necessary feature. However, the negative feedback kept flowing, so van Rossum conceded in his third post with a much less ambitious proposal that coincided with PEP 246 (Martelli and Evans [2005]). PEP 246 was a proposal to create a uniform `adapt()` function to convert an object of one type to any other type, which provided some of the same benefits of static typing at runtime; van Rossum hoped to offend fewer people by making optional static typing an extension of

this `adapt()` functionality. But PEP 246 was eventually rejected, and it is unclear exactly where its rejection fits chronologically after van Rossum's discussion of typing. It appears that the effort to follow through with van Rossum's proposals and form an official PEP on optional static typing was reliant on the success of PEP 246, which was rejected, and so the efforts of implementation fell by the wayside.

We will now describe the challenges inherent to the problem of static typing of Python pointed out by van Rossum's four blog posts, many of which are relevant to Pyty.

In the first blog post, van Rossum [2004] mostly raised a list of issues he foresaw—including container types, type variables, and method signature overloading—without proposals of how to deal with them.

In the second blog post, van Rossum [2005a] discussed in depth a proposal for interfaces that would essentially mirror class definition syntax, but with methods that would validate pre- and post-conditions. In his proposal, when a class is defined as implementing an interface, its methods are wrapped in the interface's validation mechanism to ensure types could at least be verified at runtime if static analysis isn't possible. He also briefly discussed how classes implicitly define interfaces, but typical subclasses usually do not define subtypes of that interface (because subclass methods generally have a confined, not expanded, domain). He discussed the issue of extreme dynamic behavior, and how any reasonable typing engine would have to detect extreme behavior and then give up. This is analogous to our approach with Pyty, except that we are considering a much more confined use case to be normal and a much larger set of operations to be "extreme dynamic behavior."

The third and fourth blog posts did not present any noteworthy details on challenges. In the third post, van Rossum [2005b] trimmed his previous proposals to keep only the details about interfaces. In the fourth post, van Rossum [2005c] explained that the community was strongly in favor of PEP 246's `adapt()` and he supported reducing the power and complexity of static typing by having it live side-by-side with `adapt()`.

2. Starkiller: A Static Type Inferencer and Compiler for Python

Salib [2004] adapts the Cartesian Product Algorithm introduced by Agesen [1995] to Python to create a faster Python compiler. Salib's Starkiller program performs full type inference in order to compile Python programs into equivalent C++ programs.

The Cartesian Product Algorithm used for type inference determines types by modeling data flow instead of the constraint and unification algorithm pioneered by Hindley and Milner. Hindley-Milner style typing uses the operations performed on expressions to generate restrictions on the types of those expressions. Traditional Hindley-Milner type inference determines the most general type assignable to an expression given the usage restrictions, through unification. This approach assumes an expression can be assigned any type, and narrows down the set of possibilities based on restrictions that arise; this makes it incompatible for the purposes of compiler optimization, where the relevant information is the most specific type possible. The Cartesian Product Algorithm models data flow, and, rather than narrowing down a set of possible types by eliminating impossible types, builds a set of possible types for an expression by adding types the algorithm knows the expression can take on. For example, if a variable is set to 5 in one branch of an if statement and to 5.0 in another, the type inferred for the expression is $\{\textbf{int}, \textbf{float}\}$, meaning that the data stored in the variable is guaranteed to be one of the specified types. Whereas the Hindley-Milner approach that we will be using makes more sense in the context of spotting errors, modeling data flow makes more sense in the context of compiling Python code into C++.

The Cartesian Product Algorithm gets its name from the algorithm's treatment of functions. As we have stated, the algorithm determines the smallest set of possible types for an expression. When a function call is encountered, the tuple of actual types that will get passed to the function is a member of the Cartesian product of the possible type sets of each parameter. For example, if `a` has type $\{\textbf{int}, \textbf{float}\}$, and `b` has type $\{\textbf{str}, \textbf{unicode}\}$, then $\{\textbf{int}, \textbf{float}\} \times \{\textbf{str}, \textbf{unicode}\} = \{(\textbf{int}, \textbf{str}), (\textbf{int}, \textbf{unicode}), (\textbf{float}, \textbf{str}), (\textbf{float}, \textbf{unicode})\}$, and each of these tuples corresponds to a possible set of inputs handed to a function call

`f(a, b)`. For each member of the Cartesian product, the algorithm re-evaluates the function body with the parameters set to those types; the algorithm then stores a function template for that instance of the parameter types, which is a pairing of the (monomorphic) parameter types and the effected result type set. All templates for a function are stored so that, if a function is called multiple times with argument lists that have overlapping Cartesian products, extra computation is not needed for the repeated argument types lists (if a new argument type list is encountered, a template is stored; if an old argument type list is encountered, the result type set is looked up). This treatment handles parametric polymorphism over specific type sets, as opposed to universal quantification.

3. Localized Type Inference of Atomic Types in Python

Cannon [2005] similarly attempts to perform Python type inference to improve performance, but embeds the type inference engine into the Python compiler. Cannon uses an algorithm similar to the Cartesian Product Algorithm, but, whereas Salib's Starkiller program replaces the standard Python compilation step, Cannon's inference engine intercepts bytecode emitted by the Python compiler and injects additional type-specific bytecodes to improve performance. Cannon's implementation contains a switch to control whether the algorithm is flow-sensitive or -insensitive; the flow-sensitive version of the algorithm adapts the data flow constraint graph after each statement. Unfortunately, Cannon's goal of reaching a 5% performance increase is not reached.

4. Static Analysis Tools

To accommodate Python's lack of built-in static analysis, there are a few static analysis tools that support the Python development ecosystem. In addition to checking for instances of specific common errors, these tools also typically perform analysis on coding style.

4.1. PEP 8. van Rossum and Warsaw [2001], in Python Enhancement Proposal (PEP) 8, outline a style guide for Python to "improve readability of code and make it more consistent across the wide spectrum of Python code." The majority of Python developers attempt to adhere closely to this standard, though individual companies often maintain their own

stricter style guides (for example, Patel et al. [2012] maintain a style guide for Python use at Google). Roholl [2006] provides a tool which checks Python code against some of the conventions of PEP 8, and which many developers run on their code before releasing to open-source communities. This tool is strictly limited to style analysis.

4.2. PyChecker. PyChecker, written by Norwitz [2008], is perhaps the most commonly used tool today, and detects common errors in Python source code, most of which are "typically caught by a compiler for less dynamic languages, like C and C++." Some of the error cases that PyChecker detects include passing the wrong number of parameters to functions (both built-in and user-defined), calling class attributes that do not exist, passing incorrect arguments to format strings, redefining functions, defining unused arguments and variables, and using a variable before initializing it. These error cases are built around implementing checks for common Python errors, and not on a theoretical structure for safe programs.

4.3. Pylint. Pylint, written by Thenault [2006], is fundamentally similar to PyChecker, but includes a wider variety of error checks and customization (allowing a user-specified coding style to check against). Generally, Pylint also reports many more warnings than PyChecker.

CHAPTER 3

PT Type System

To define Pyty's type system (henceforth named PT), we will describe the object language, types, typing judgments, and then further explore some specific language features.

1. TPython

Before describing PT, we need to formally describe our object language—a proper subset of Python which we will henceforth refer to as TPython. In Figure 1, we have presented the abstract grammar for TPython, which has been adapted from the complete Python grammar [Python development team, 2012a]. In Chapter 4, Section 7, we will discuss the complete Python grammar further and some of the reasoning behind limiting TPython's scope.

The abstract grammar is specified in the Zephyr Abstract Syntax Description Language (ASDL) as outlined by Wang et al. [1997]. The syntax of ASDL was designed to mimic standard syntax for context-free grammars. The grammar consists of a series of node type definitions, which assign a node type name (e.g., mod, expr, and stmt) to a node type, which is a node constructor or a disjoint sum of node constructors. Node constructors consist of a constructor name and a series of named fields. Named fields have a node type and a name—the names of fields are structurally insignificant to the grammar specification, but are helpful for readability. The field names are also important because they correspond directly to the names of AST node attributes in Python's AST module (discussed further in Chapter 4). Field node types can also be specified with a ? (indicating an optional argument) or * (indicating a sequence, possibly empty, of such nodes). The majority of node types are defined by a single node constructor or a disjoint sum of multiple constructors (e.g., mod, expr, and stmt), but two special nodes (arguments and keyword) are defined as

products, which are essentially "unnamed" constructors. ASDL also contains five primitive node types—int (not present in our TPython presentation), identifier, string, object, and bool. We also introduce a ptype primitive to represent the type object which we store in a type declaration.

We will describe the form of the TPython abstract grammar constructors in the following sections. We specify a *term* to be any module, statement list, statement, or expression node of TPython—these will be correspond to nodes which can be assigned types in PT. When we describe TPython terms in subsequent discussions, we will use these node constructors, so it is important to understand the structure of each node. Since writing out a tree of embedded node constructors inline becomes verbose very quickly, we have decided to describe fields positionally in such descriptions instead of including field names (e.g., Name("b", Load) instead of Name(id = "b", ctx = Load). The following discussion and Figure 1 should be used as references to interpret positional fields. Information about the behavior of the Python language constructs has been gathered from the Python language reference [Python development team, 2012b]. We will specify certain fields that Pyty always expects to be empty—or to be an empty list—and this forces us to clarify an important subtlety. TPython is the language set on which Pyty will run as expected, whereas Pyty may crash for language constructs not contained in this grammar. But fields that are included in the grammar and specified to be empty will not cause Pyty to crash—they will just cause Pyty to decide the program does not typecheck correctly.

We provide a brief note about contexts, since they appear in the specification of several nodes. The Python abstract grammar includes these context fields because of a fundamentally different approach to using AST nodes. They are provided so that a node can be examined without knowing specifically where it sits in the parse tree. However, Pyty always implicitly examines nodes based on the context of their parent nodes, so this information is redundant for our purposes.

```
mod = Module(stmt* body)
stmt = TypeDec(expr* targets, ptype t)
     | FunctionDef(identifier name, arguments args, stmt* body
                   expr* decorator_list)
     | Return(expr? value)
     | Assign(expr* targets, expr value)
     | AugAssign(expr target, operator op, expr value)
     | Print(expr? dest, expr* values, bool nl)
     | For(expr target, expr iter, stmt* body, stmt* orelse)
     | While(expr test, stmt* body, stmt* orelse)
     | If(expr test, stmt* body, stmt* orelse)
     | Expr(expr value)
     | Pass | Break | Continue
expr = BoolOp(boolop op, expr* values)
     | BinOp(expr left, operator op, expr right)
     | UnaryOp(unaryop op, expr operand)
     | Lambda(arguments args, expr body)
     | IfExp(expr test, expr body, expr orelse)
     | Compare(expr left, cmpop* ops, expr* comparators)
     | Call(expr func, expr* args, keyword* keywords, expr? starargs,
            expr? kwargs)
     | Num(object n)
     | Str(string s)
     | Subscript(expr value, slice slice, expr_context ctx)
     | Name(identifier id, expr_context ctx)
     | List(expr* elts, expr_context ctx)
     | Tuple(expr* elts, expr_context ctx)
```

```
expr_context = Load | Store | Del | AugLoad | AugStore | Param
       slice = Slice(expr? lower, expr? upper, expr? step)
             | Index(expr value)
      boolop = And | Or
    operator = Add | Sub | Mult | Div | Pow | Mod | LShift
             | RShift | BitOr | BitXor | BitAnd | FloorDiv
     unaryop = Invert | Not | UAdd | USub
       cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn
   arguments = (expr* args, identifier? vararg, identifier? kwarg, expr* defaults)
     keyword = (identifier arg, expr value)
```

FIGURE 1. TPython abstract grammar. Nodes above the break can be assigned a type as will be discussed in Section 3.

1.1. Module Node Constructors.

1.1.1. *Module.* Module(stmt* body)

body: The list of statements contained in the module.

1.2. Statement Node Constructors.

1.2.1. *TypeDec.* TypeDec(expr* targets, ptype t)

targets: The expression being given a type. Pyty will only typecheck correctly if this is an identifier.

t: A PType object representing the type specified in the type declaration comment.

Type declarations are specified in TPython source code in comments of the form

```
#: identifier : type
```

and are analogous to type declarations made in statically typed languages like Java. Type declarations must appear before, or on the same line as, the declared identifier appears in use. For example,

```
#: i : int
i = 0
```

and

```
is_done = True  #: is_done : bool
```

are valid, but

```
f = 3.14
#: f : float
```

is not a valid type declaration.

1.2.2. *FunctionDef.* FunctionDef(identifier name, arguments args, stmt* body, expr* decorator_list)

name: The name of the function. This is represented by a string in the Python AST module.

args: The parameters to the function. Further explained in Section 1.4.2.

`def f(a):` `  return`	FunctionDef("f", ([Name("a", Param)], , []), [Return()], [])

FIGURE 2. FunctionDef example.

`x = 5`	Assign([Name("x")], Num(5))
`x = y = z`	Assign([Name("x"), Name("y")], Name("z"))
`x, y = y, x`	Assign([Tuple([Name("x", Store), Name("y", Store)], Store)], Tuple([Name("y", Load), Name("x", Load)], Load))

FIGURE 3. Assign examples.

body: The body of the function (i.e., the statements to be run when the function is called).

decorator_list: A list of decorators applied to the function. Decorators are callable expressions which are passed the function object when the function is defined. Pyty does not support decorator expressions, so any FunctionDef with decorators present will not typecheck.

Figure 2 helps to illustrate the structure of common FunctionDef nodes in TPython.

1.2.3. *Return.* Return(expr? value)

value: The expression being returned. This will be empty for a void return statement.

1.2.4. *Assign.* Assign(expr* targets, expr value)

targets: Sequence of expressions being assigned to.

value: Value being assigned.

Figure 3 helps to illustrate the structure of common Assign nodes in TPython.

1.2.5. *AugAssign.* AugAssign(expr target, operator op, expr value)

target: The expression being augmented.

op: The binary operator.

value: The other operand.

AugAssign(tar, op, val) has the same evaluation semantics as Assign([tar], BinOp(tar, op, val)).

1.2.6. *Print.* Print(expr? dest, expr* values, bool nl)

dest: The file-like destination for the extended print syntax. The extended print syntax (referred to as "print chevron" in the Python language reference [Python development team, 2012b]) is `print >> dest, ...` which is equivalent to printing the output of `print ...` to the `dest` file object. This extended syntax is not supported by Pyty, so a Print node with a nonempty dest will not typecheck in Pyty.

values: The expressions to be printed.

nl: Whether the Print statement will include a trailing newline character; Python only leaves off the newline if the Print statement ends with a comma (like `print a, b,`).

1.2.7. *For.* For(expr target, expr item, stmt* body, stmt* orelse)

target: The target assigned to on each iteration of the loop. Note that this can be any assignable expression and not just an identifier; for example, `for x,y in pts:` is a valid For loop header.

iter: The collection to iterate through. Pyty only supports for loops iterating over lists, so a For node will only typecheck correctly if this is an expression of list type.

body: The list of statements to run on each iteration.

orelse: The list of statements to run at the end of the loop.

1.2.8. *While.* While(expr test, stmt* body, stmt* orelse)

test: The expression tested to determine when to terminate the loop.

body: The list of statements to run on each iteration.

orelse: The list of statements to run at the end of the loop.

1.2.9. *If.* If(expr test, stmt* body, stmt* orelse)

test: The expression tested to determine which branch to follow.

body: The list of statements to run if test is true.

orelse: The list of statements to run if test is false.

Note that in standard Python, non-booleans are allowed as the test of conditionals, but this is not true in TPython.

1.2.10. *Expr.* Expr(expr value)

value: The expression which is sitting inside the statement.

1.2.11. *Nullary Statement Constructors.* Pass, Break, and Continue are nullary node constructors that do not contain any fields.

1.3. Expression Node Constructors. In subsequent discussions, we will often use shorthand notation mimicking concrete Python syntax to succinctly describe an expression in TPython, foregoing the inline tree of embedded node constructors necessary to describe a statement. The idea is that the shorthand notation for an expression AST should look very much like the Python code that would generate that tree. For example, we will often simplify BinOp(Num(1), Add, Num(2)) as `1 + 2`. We will now describe the fields of expression constructors and explain how each constructor is represented in the shorthand style.

1.3.1. *BoolOp.* BoolOp(boolop op, expr* values)

op: The Boolean operator performed. Either And or Or.

values: The operands.

We repeat the operator with infix notation to condense BoolOp($op, [x_1, x_2, \ldots, x_n]$) to $x_1\, op\, x_2\, op \cdots op\, x_n$ where we represent And as `and` and Or as `or`. For example, BoolOp(And, [Name("x", Load), Name("y", Load)]) is shortened to `x and y`.

1.3.2. *BinOp.* BinOp(expr left, operator op, expr right)

left: The left operand.

op: The binary operator. One of the following: Add, Sub, Mult, Div, Pow, Mod, LShift, RShift, BitOr, BitXor, BitAnd, or FloorDiv.

right: The right operand.

We use infix notation to condense BinOp(x, op, y) to $x\ op\ y$, where we represent Add as +, Sub as -, Mult as *, Div as /, Pow as **, Mod as %, LShift as <<, RShift as >>, BitOr as |, BitXor as ^, BitAnd as &, and FloorDiv as //. For example, BinOp(Name("x", Load), Add, Num(5)) is shortened to x + 5.

1.3.3. *UnaryOp.* UnaryOp(unaryop op, expr operand)

op: The unary operator. One of the following: Invert (bitwise inversion), Not (Boolean negation), UAdd, or USub.

operand: The single operand.

We use prefix notation to condense UnaryOp(op, x) to $op\ x$, where we represent Invert as ∼, Not as not, UAdd as +, and USub as -. For example, UnaryOp(Invert, Num(3)) is shortened to ∼ 3.

1.3.4. *Lambda.* Lambda(arguments args, expr body)

args: The parameters to the function. Further explained in Section 1.4.

body: The body of the function (i.e., the expression the function evaluates to when called).

We condense Lambda(a, b) to lambda a: b. For example, Lambda(([Name("a", Param), Name("b", Param)], , []), Name("b", Load)) becomes lambda a, b: b.

1.3.5. *IfExp.* IfExp(expr test, expr body, expr orelse)

test: The expression tested.

body: The expression to evaluate to if test is true.

orelse: The expression to evaluate to if test is false.

We condense IfExp(x, a, b) to a if x else b. For example, IfExp(Name("True", Load), Num(1), Num(0)) becomes 1 if True else 0.

1.3.6. *Compare.* Compare(expr left, cmpop* ops, expr* comparators)

left: The left-most expression in the chain of comparisons.

ops: The list of comparison operators. Each operator is one of the following: Eq, NotEq, Lt, LtE, Gt, GtE, Is, IsNot, In, NotIn.

comparators: The rest of the subsequent expressions.

The shorthand syntax for Compare also helps illustrate how the chain of comparisons is stored in left, ops, and comparators. We represent Eq as =, NotEq as `!=`, Lt as <, Lte as <=, Gt as >, GtE as >=, Is as `is`, IsNot as `is not`, In as `in`, and NotIn as `not in`. We condense Compare($x_0, [o_1, o_2, \ldots, o_n], [x_1, x_2, \ldots, x_n]$) to $x_0\ o_1\ x_1\ o_2\ x_2\ \cdots\ o_n\ x_n$. For example, Compare(Num(1), [Lt, Lte], [Num(2), Num(3)]) becomes `1 < 2 <= 3`.

1.3.7. *Call.* Call(expr func, expr* args, keyword* keywords, expr? starargs, expr? kwargs)

func: The function being called. Pyty only supports Call nodes with identifiers as the function being called, so the application of a non-identifier will not typecheck in Python.

args: Sequence of expressions used as arguments.

keywords: Sequence of keywords used along with arguments. Pyty does not support keyword arguments, so a Call node will only typecheck correctly if this field is an empty list.

starargs: An iterable collection treated as additional positional arguments. Pyty does not support starargs, so a Call node will only typecheck correctly if this field is empty.

kwargs: A mapping treated as additional keyword arguments. Pyty does not support kwargs, so a Call node will only typecheck correctly if this field is empty.

We condense Call($f, [x_1, x_2, \ldots, x_n], [],,$) to $f(x_1, x_2, \ldots, x_n)$.

1.3.8. *Num.* Num(object n)

n: The number value. This is represented by a numeric literal in the Python AST module.

We condense Num(`n`) to `n`.

1.3.9. *Str.* Str(string s)

s: The string value. This is represented by a string literal (`str` or `unicode`) in the Python AST module.

We condense Str(`s`) to `s`. For example, Str(`"Hi"`) becomes `"Hi"`.

1.3.10. *Subscript.* Subscript(expr value, slice slice, expr_context ctx)

value: The collection being subscripted.

slice: The slice. Further explained in Section 1.4.

ctx: The context the subscription is being used in. One of the following: Load, Store, Del.

We condense Subscript(col, s, ctx) to $col\,[s]$, though we will elaborate further on how to condense the slice term s in Section 1.4.

1.3.11. *Name.* Name(identifier id, expr_context ctx)

id: The name of the identifier. This is represented by a string in the Python AST module.

ctx: The context the identifier is being used in. One of the following: Load, Store, Del, or Param.

Note that Python (and thus TPython) treat certain values (`True`, `False`, and `None`) as Name nodes with specific id fields. `True` is Name("`True`", Load), `False` is Name("`False`", Load), and `None` is Name("`None`", Load). We condense Name("`x`", ctx) to `x`.

1.3.12. *List.* List(expr* elts, expr_context ctx)

elts: The elements in the list.

ctx: The context the list is being used in. One of the following: Load, Store, or Del.

We condense List($[x_1, x_2, \ldots, x_n]$, ctx) to $[x_1, x_2, \ldots, x_n]$.

1.3.13. *Tuple.* Tuple(expr* elts, expr_context ctx)

elts: The elements in the tuple.

ctx: The context the tuple is being used in. One of the following: Load, Store, or Del.

We condense Tuple($[x_1, x_2, \ldots, x_n]$, ctx) to $(x_1, x_2, \ldots, x_n)$.

1.4. Miscellaneous Node Constructors.

1.4.1. *Slice.* Slice(expr? lower, expr? upper, expr? step)

lower: The lower bound of the slice.

upper: The upper bound of the slice.

step: The step of the slice.

We condense Slice(l, u, s) to l:u:s, where the last colon is optional if s is empty. For example, Subscript(Name(l, Load), Slice(Num(1), ,), Load) becomes `l[1::]` or `l[1:]` and Subscript(Name(l, Load), Slice(Num(1), Num(10), Num(2)), Load) becomes `l[1:10:2]`.

Index(expr value)

value: The position being indexed.

We condense Index(x) to x. For example, Subscript(Name(l, Load), Index(Num(0)), Load) becomes `l[0]`.

1.4.2. *Arguments.* arguments = (expr* args, identifier? vararg, identifier? kwarg, expr* defaults)

args: Sequence of positional arguments.

vararg: A parameter of the form `*identifier` that receives a tuple when the function is executed containing excess positional arguments. Pyty does not support varargs, so this field will always be empty in TPython.

kwarg: A parameter of the form `**identifier` that receives a dictionary when the function is executed containing excess keyword arguments. Pyty does not support kwargs, so this field will always be empty in TPython.

defaults: Sequence of default values for positional arguments. Pyty does not support default argument values, so this field will always be empty.

We condense ($[x_1, x_2, \ldots, x_n]$, , , []) to $x_1, x_2, \ldots, x_n$. For example, Lambda(([Name("a", Param), Name("b", Param)], , []), Name("b", Load)) becomes `lambda a, b: b`.

Note that, as specified in the ASDL grammar, an argument node is a product type, but in practice this makes it very similar to a constructor node with the elements of the product as constructor fields. This is, in fact, how the Python AST module handles it; in further discussion, we will assume that argument nodes are created by argument constructors.

$$
\begin{array}{rll}
\tau, \tau_i ::= & \textbf{int} \mid \textbf{float} \mid \textbf{str} \mid \textbf{unicode} \mid \textbf{bool} \mid \textbf{unit} \mid \textbf{stmt} \mid \textbf{stmt*} \mid \textbf{mod} & \text{literals} \\
\mid & \tau\ \textbf{list} & \text{list} \\
\mid & \langle \tau_1, \ldots, \tau_n \rangle & \text{tuple} \\
\mid & \tau_1 \to \tau_2 & \text{arrow} \\
\mid & \alpha & \text{type variable} \\
\gamma ::= & \forall \bar{\alpha}.\tau & \text{type scheme}
\end{array}
$$

FIGURE 4. Grammar for types and type schemes in PT. The definitions for τ and τ_i define types, and the definition for γ defines a type scheme.

2. Types

2.1. Abstract Types. We begin our specification of PT by defining a *type* to be a type variable or a term built out of type constructors as presented in the first definition of Figure 4. Note that type literals (e.g., **int** or **str**) are nullary type constructors.

We will formally describe the use of these constructors in Section 3, but for now we will present a brief, intuitive description of what we intend the constructors to represent.

Most of the type literals are immediately clear, but a few deserve special attention. The **unit** type has two meanings: the type of the Python `None` literal and the domain type of a function which takes no parameters. Note that **unit** is also the range type of a function which does not return a value, but in Python a function with no—or an empty—return statement returns the value `None`, so this meaning is already implicit in **unit**'s use as the type of `None`. The **stmt** type represents the type of any valid statement, the **stmt*** type represents the type of a sequence of valid statements, and the **mod** type represents the type of a valid module.

The list constructor corresponds to its Python counterpart; a value of type α **list** is a Python `list` object whose elements all have type α. Note that Python allows heterogeneous lists; restricting to homogeneous lists is a design decision of PT—it reflects common statically-typed languages like Java or ML and the common practices of programmers used to these languages. If we have a type $\tau = \alpha$ **list**, then we refer to α as the element type of τ.

The tuple constructor corresponds to its Python counterpart; an element with type $\langle \tau_1, \ldots, \tau_n \rangle$ is a tuple whose first element has type τ_1, whose second element has type

τ_2, etc., and whose last element has type τ_n. Note that, while the list constructor only supports homogeneous lists, the tuple constructor allows heterogeneous tuples. We have the following alternative syntax for tuple constructors, which will allow us to succinctly express certain forms of tuple types: $\langle \tau_i{}^{i\in 1\ldots n} \rangle = \langle \tau_1, \ldots, \tau_n \rangle$. If we have a type $\tau = \langle \alpha_1, \ldots, \alpha_n \rangle$, then we refer to α_i as the i^{th} element type of τ.

The arrow constructor corresponds to Python function objects (which includes functions defined with the `lambda` construct); an element with type $\tau_1 \to \tau_2$ is a function which takes input of type τ_1 and returns a result of type τ_2. If we have a type $\tau = \alpha_1 \to \alpha_2$, then we refer to α_1 as the domain of τ and to α_2 as the range of τ. A function with multiple inputs has an arrow type whose domain is a tuple of the function's inputs. The arrow constructor associates to the right, and has lower precedence than the list constructor; for example, α **list** $\to \beta$ **list** $\to \alpha$ **list** is equivalent to α **list** $\to (\beta$ **list** $\to (\alpha$ **list**$))$.

A type variable is a name that stands for a type, which can be instantiated as a type through type substitution. We denote replacing all instances of type variable α with type σ in the type τ as $\tau[\alpha \mapsto \sigma]$. We also define multiple simultaneous type substitutions as $\tau[\bar{\alpha} \mapsto \bar{\sigma}] = \tau[\alpha_1 \mapsto \sigma_1, \alpha_2 \mapsto \sigma_2, \ldots, \alpha_n \mapsto \sigma_n]$, where the $\bar{\alpha}$ notations signifies a class of variables (here $\bar{\alpha} = \{\alpha_1, \ldots, \alpha_n\}$).

We define a *type scheme* to be a type universally quantified over a (potentially empty) set of type variables as presented by the second definition in Figure 4. If $\bar{\alpha} = \alpha_1, \ldots, \alpha_n$, then $\forall\bar{\alpha}.\sigma = \forall\alpha_1, \ldots, \alpha_n.\sigma = \forall\alpha_1.\cdots\forall\alpha_n.\sigma$. A type variable that is quantified is *bound* in the type scheme, and type variables that are not bound are *free*. We denote the free type variables that appear free in a type (scheme) σ as $ftv(\sigma)$. We will often want to quantify a type (scheme) over its all of its free variables, and we express this as $\forall\sigma = \forall\bar{\alpha}.\sigma$ where $\bar{\alpha} = ftv(\sigma)$. When we defined type substitution for types, we claimed that the notation signifies replacing *all* instances of the specified type variables, not just the free instances; this is because a type cannot be built out of type schemes and so cannot contain any bound variables. We define type substitution for type schemes similarly, except that it replaces all *free* instances of the specified type variables.

$$
\begin{array}{lr}
\tau, \tau_i ::= \texttt{int} \mid \texttt{float} \mid \texttt{str} \mid \texttt{unicode} \mid \texttt{bool} \mid \texttt{unit} & \text{literals} \\
\quad \mid \ \texttt{[}\tau\texttt{]} & \text{list} \\
\quad \mid \ \texttt{()} \mid \texttt{(}\tau_1\texttt{,)} \mid \texttt{(}\tau_1\texttt{,}\tau_2\texttt{)} \mid \texttt{(}\tau_1\texttt{,}\tau_2\texttt{,)} \mid \ldots \mid \texttt{(}\tau_1\texttt{,}\ldots\tau_n\texttt{,)} & \text{tuple} \\
\quad \mid \ \tau_1 \texttt{ -> } \tau_2 & \text{arrow} \\
\quad \mid \ \texttt{'a} & \text{type variable} \\
\quad \mid \ \texttt{(}\tau\texttt{)} & \text{grouping}
\end{array}
$$

FIGURE 5. Concrete grammar for types in PT.

2.2. Concrete Types. The types presented so far are our abstract, mathematical representation of PT types that will be used to perform typing judgments. Chapter 4 will go into more depth about the implementation of the type system presented to the user, but the main distinction is in the syntax of the concrete type grammar, presented in Figure 5.

One important distinction is that the user-facing concrete type grammar does not contain equivalents of the abstract **stmt**, **stmt***, and **mod** type literals; this is because the user will only declare types for identifiers (which can only be assigned expressions). The syntax for type variables is matched by the regular expression `'[a-zA-Z0-9]+`. $\texttt{(}\tau\texttt{)}$ signifies grouping (which means $\texttt{(}\tau\texttt{)} = \tau$), not a single-element tuple, so single element tuples must be specified with a trailing comma. Note that `()` signifies the type of an empty tuple and is not equivalent to `unit`. Although the type of the empty tuple and `unit` are considered isomorphic in many type systems, we have chosen to maintain a distinction because in Python the `None` value and the empty tuple value do act differently.

3. Typing Derivations

3.1. Introduction. A *typing derivation* is a proof that we can assign a type τ to a term e under a type environment Γ—such a claim is written as $\Gamma \vdash e : \tau$, read as "Γ proves that e can be assigned type τ." Our typing derivation system contains a series of assignment rules—each is a rule that allows us to conclude that a certain form of expression can be assigned a certain type under a certain environment, given that a series of conditions is satisfied. A proof in this system thus takes the form of a tree of these rules.

The type environment maps program identifiers (technically names of identifiers, stored as strings) to the type that the identifier has been stated to have. This allows us to form derivations to assign types to expressions which contain identifiers, like `f(a, b)`, assuming we have already stored the types of `f`, `a`, and `b` in the environment. Some other type systems include the types of object language literals in an initial environment; since there are an infinite number of such literals (every integer, every floating point number, every string, etc.), we decided to instead model this with specific assignment rules for each type of literal. Our type system also stores a special `"return"` identifier that, if examining a function definition's body, is used to store the expected return type. This is a safe place to store such information because type environments map program identifiers to types, and `"return"` is not a valid identifier in Python because it conflicts with the obvious keyword.

Some conventions (mostly typographic) for type assignment rules follow:

(1) Sans-serif typeface denotes the node of the Python abstract syntax tree.
(2) **Bold font weight** denotes a type in PT.
(3) `Monospace typeface` denotes verbatim Python code. When present as the condition of a type assignment rule, the code will be a Python expression, and the condition is satisfied if and only if the expression evaluates to `True`.
(4) The lowercase italicized Greek letters τ and σ (and their subscripts) are metavariables spanning over PT types.
(5) The lowercase italicized Greek letter γ (and its subscripts) is a metavariable spanning over PT type schemes.
(6) Unless otherwise specified, italicized letters (e.g., e, e_0) signify metavariables spanning over TPython terms.
(7) Rule variants condense formally similar assignment rules together.
    (a) A forward slash (/) indicates a global rule variant, and represents one dependent branching taken consistently with other global rule variants throughout the rest of the rule. For example, a rule containing $a/b * c/d$ is syntactic sugar for one rule containing $a * c$ and another rule containing $b * d$.

(b) A vertical pipe (|) indicates a local rule variant, and represents a single independent branching point—the effect is that a rule with multiple local variants will have several independent branching points. For example, a rule containing $a \mid b * c \mid d$ is syntactic sugar for the four rules respectively containing $a * c$, $a * d$, $b * c$, and $b * d$.

Note that both types of rule variants can be nested to achieve arity greater than two—for example, $a/b/c$ nests global rule variants and acts like a ternary global rule variant.

(8) Underscores as fields of AST nodes (e.g., Subscript(c, _, _)) denote a wildcard. If present as a condition, the condition is satisfied if there exists a derivation satisfying the condition with any node taking the place of the wildcard.

3.2. Module Typing Rules.

$$\frac{\vdash ss : \textbf{stmt*}}{\vdash \mathsf{Module}(ss) : \textbf{mod}} \text{ (Mod)}$$

3.3. Statement List Typing Rules.

$$\frac{}{\Gamma \vdash \mathsf{StmtList}() : \textbf{stmt*}} \text{ (Stmts-Base)}$$

$$\frac{s_0 \text{ not TypeDec node} \qquad \Gamma \vdash s_0 : \textbf{stmt} \qquad \Gamma \vdash \mathsf{StmtList}(s_1, \ldots, s_{n-1}) : \textbf{stmt*}}{\Gamma \vdash \mathsf{StmtList}(s_0, s_1, \ldots, s_{n-1}) : \textbf{stmt*}} \text{ (Stmts)}$$

$$\frac{s_1 \text{ not Assign or FunctionDef node} \qquad x \notin \Gamma \qquad \Gamma, x : \tau \vdash \mathsf{StmtList}(s_1, \ldots, s_{n-1}) : \textbf{stmt*}}{\Gamma \vdash \mathsf{StmtList}(\mathsf{TypeDec}(x, \tau), s_1, \ldots, s_{n-1}) : \textbf{stmt*}} \text{ (StmtsT)}$$

$$\frac{x \notin \Gamma \qquad \Gamma \vdash e : \tau \qquad \Gamma, x : \forall \tau \vdash \mathsf{StmtList}(s_2, \ldots, s_{n-1}) : \textbf{stmt*}}{\Gamma \vdash \mathsf{StmtList}(\mathsf{TypeDec}(x, \tau), \mathsf{Assign}(e, [x]), s_2, \ldots, s_{n-1}) : \textbf{stmt*}} \text{ (Stmts-LetA)}$$

$$\frac{f \notin \Gamma \qquad \Gamma, f : \sigma \to \tau \vdash \mathsf{FunctionDef}(f, a, b, d) : \textbf{stmt} \qquad \Gamma, f : \forall(\sigma \to \tau) \vdash \mathsf{StmtList}(s_2, \ldots, s_{n-1}) : \textbf{stmt*}}{\Gamma \vdash \mathsf{StmtList}(\mathsf{TypeDec}(f, \sigma \to \tau), \mathsf{FunctionDef}(f, a, b, d), s_2, \ldots, s_{n-1}) : \textbf{stmt*}} \text{ (Stmts-LetF)}$$

Note that there is not a StmtList abstract syntax tree node in TPython; in Section 1 we described statement lists with brackets because it was clear based on field positioning what brackets were to be interpreted as. For example, in FunctionDef(f, $args$, [Return(), Return(), Return()], []), it is clear that [Return(), Return(), Return()] is a list of statements because the third field of FunctionDef is a list of statements and it is clear that [] is a list of expressions because the fourth field of FunctionDef is a list of expressions. Here we

use here we use notation mimicking a StmtList node to distinguish it from other possible interpretations for the bracket notation.

3.4. Statement Typing Rules.

3.4.1. *Function Definition – FunctionDef.*

$$\frac{\Gamma \vdash f : \textbf{unit} \to \tau \qquad \Gamma, \texttt{return} : \tau \vdash b : \textbf{stmt*}}{\Gamma \vdash \mathsf{FunctionDef}(f, ([], , , []), b, []) : \textbf{stmt}} \text{ (Fn-Def1)}$$

$$\frac{\Gamma \vdash f : \sigma \to \tau \qquad \sigma \neq \textbf{unit} \qquad \Gamma, x : \sigma, \texttt{return} : \tau \vdash b : \textbf{stmt*}}{\Gamma \vdash \mathsf{FunctionDef}(f, ([\mathsf{Name}(x)], , , []), b, []) : \textbf{stmt}} \text{ (Fn-Def2)}$$

$$\frac{n \geq 2 \qquad \Gamma \vdash f : \langle \sigma_1, \ldots, \sigma_n \rangle \to \tau \qquad \Gamma, x_i : \sigma_i, \texttt{return} : \tau \vdash b : \textbf{stmt*}}{\Gamma \vdash \mathsf{FunctionDef}(f, ([\mathsf{Name}(x_1), \mathsf{Name}(x_2), \ldots, \mathsf{Name}(x_n)], , , []), b, []) : \textbf{stmt}} \text{ (Fn-Def3)}$$

3.4.2. *Return Statement – Return.*

$$\frac{\Gamma(\texttt{return}) = \textbf{unit}}{\Gamma \vdash \mathsf{Return}() : \textbf{stmt}} \text{ (RetU)} \qquad \frac{\Gamma \vdash e : \Gamma(\texttt{return})}{\Gamma \vdash \mathsf{Return}(e) : \textbf{stmt}} \text{ (Ret)}$$

3.4.3. *Assignment – Assign.*

$$\frac{\Gamma \vdash t_i : \tau_i \qquad \Gamma \vdash v : \tau_i}{\Gamma \vdash \mathsf{Assign}([t_1, \ldots, t_n], v) : \textbf{stmt}} \text{ (Assmt)}$$

Side conditions: $n \geq 1$ and no t_i is a subscript of a tuple.

3.4.4. *Augmented Assignment – AugAssign.*

$$\frac{\Gamma \vdash e_0 : \tau \qquad \Gamma \vdash \mathsf{BinOp}(e_0, op, e_1) : \tau}{\Gamma \vdash \mathsf{AugAssign}(e_0, op, e_1) : \textbf{stmt}} \text{ (Aug-Assmt)}$$

3.4.5. *Print Statement – Print.*

$$\frac{}{\Gamma \vdash \mathsf{Print}(, v, nl) : \textbf{stmt}} \text{ (Print)}$$

3.4.6. *For Loop – For.*

$$\frac{\Gamma \vdash x : \tau \qquad \Gamma \vdash e : \tau \textbf{ list} \qquad \Gamma \vdash b_0 : \textbf{stmt*} \qquad \Gamma \vdash b_1 : \textbf{stmt*}}{\Gamma \vdash \mathsf{For}(x, e, b_0, b_1) : \textbf{stmt}} \text{ (For)}$$

Note that For requires the type of the loop variable to be defined before (and thus outside the scope of) the loop.

3.4.7. *While Loop – While.*

$$\frac{\Gamma \vdash e : \textbf{bool} \qquad \Gamma \vdash b_0 : \textbf{stmt*} \qquad \Gamma \vdash b_1 : \textbf{stmt*}}{\Gamma \vdash \mathsf{While}(e, b_0, b_1) : \textbf{stmt}} \text{ (While)}$$

3.4.8. *Conditional Block – If.*

$$\frac{\Gamma \vdash e : \textbf{bool} \qquad \Gamma \vdash b_0 : \textbf{stmt*} \qquad \Gamma \vdash b_1 : \textbf{stmt*}}{\Gamma \vdash \mathsf{If}(e, b_0, b_1) : \textbf{stmt}} \text{ (If-Stmt)}$$

3.4.9. *Expression Statement – Expr.*

$$\frac{\Gamma \vdash f : \sigma \to \tau \qquad \Gamma \vdash \mathsf{Call}(f, a, k, sa, kw) : \tau}{\Gamma \vdash \mathsf{Expr}(\mathsf{Call}(f, a, k, sa, kw)) : \textbf{stmt}} \text{ (Expr-Stmt)}$$

See Chapter 4, Section 7.5 for an explanation of the reasoning behind this rule.

3.4.10. *Trivial Statements – Pass, Break, Continue.*

$$\frac{}{\Gamma \vdash \mathsf{Pass} \mid \mathsf{Break} \mid \mathsf{Continue} : \textbf{stmt}} \text{ (Triv)}$$

3.5. Expression Typing Rules.

3.5.1. *Boolean Operations – BoolOp.*

$$\frac{\Gamma \vdash e_i : \tau}{\Gamma \vdash e_1 \ \texttt{and|or} \ e_2 \ \texttt{and|or} \ \cdots \ e_n : \tau} \text{ (BoolOp)}$$

3.5.2. *Binary Operations – BinOp.*

Numeric Operations.

$$\frac{\Gamma \vdash e_0 : \textbf{int/float} \qquad \Gamma \vdash e_1 : \textbf{int/float}}{\Gamma \vdash e_0 \bullet e_1 : \textbf{int/float}} \text{ (Arith)}$$

$$\frac{\Gamma \vdash e_0 : \textbf{int} \qquad \Gamma \vdash e_1 : \textbf{int}}{\Gamma \vdash e_0 \circ e_1 : \textbf{int}} \text{ (BitOp)}$$

where $\bullet$ is one of `+`, `-`, `*`, `/`, `//`, `%`, or `**`, and $\circ$ is one of `<<`, `>>`, `|`, `&`, or `^`.

String and List Operations.

$$\frac{\Gamma \vdash e_0 : \textbf{str/unicode/}\tau\ \textbf{list} \qquad \Gamma \vdash e_1 : \textbf{str/unicode/}\tau\ \textbf{list}}{\Gamma \vdash e_0 \ + \ e_1 : \textbf{str/unicode/}\tau\ \textbf{list}} \text{ (Flat-Cat)}$$

$$\frac{\Gamma \vdash e_0 : \textbf{int} \qquad \Gamma \vdash e_1 : \textbf{str/unicode/}\tau\ \textbf{list}}{\Gamma \vdash e_0 * e_1 \ \mid \ e_1 * e_0 : \textbf{str/unicode/}\tau\ \textbf{list}} \text{ (Flat-Rep)}$$

$$\frac{\Gamma \vdash e_0 : \textbf{str/unicode}}{\Gamma \vdash e_0 \ \% \ e_1 : \textbf{str/unicode}} \text{ (Str-Form)}$$

See Chapter 4, Section 7.6 for a discussion of the broad nature of Str-Form.

Tuple binary operations.

$$\frac{\Gamma \vdash e_0 : \langle \tau_1, \ldots, \tau_m \rangle \qquad \Gamma \vdash e_1 : \langle \tau_{m+1}, \ldots, \tau_n \rangle}{\Gamma \vdash e_0 \texttt{ + } e_1 : \langle \tau_1, \ldots, \tau_n \rangle} \text{ (Tup-Cat)}$$

$$\frac{\Gamma \vdash e : \langle \tau_0, \ldots, \tau_{n-1} \rangle \qquad \texttt{type(}m\texttt{) is int}}{\Gamma \vdash e\texttt{*}\mathsf{Num}(m) \mid \mathsf{Num}(m)\texttt{*}e : \langle \tau_{i \bmod n}{}^{i \in 0 \ldots mn-1} \rangle} \text{ (Tup-Rep)}$$

where $i \bmod n$ is the unique j such that $j \equiv i \pmod n$ and $0 \leq j < n$. Note that any $0 \leq m \leq n+1$ is valid for Tup-Cat (where $m = 0$ and $m = n+1$ indicate empty left and right tuples respectively).

3.5.3. *Unary Operations – UnaryOp.*

$$\frac{\Gamma \vdash e : \textbf{int}}{\Gamma \vdash \sim e : \textbf{int}} \text{ (Inv)} \qquad \frac{\Gamma \vdash e : \textbf{int/float}}{\Gamma \vdash \pm e : \textbf{int/float}} \text{ (Uadd)} \qquad \frac{\Gamma \vdash e : \textbf{bool}}{\Gamma \vdash \texttt{not } e : \textbf{bool}} \text{ (Not)}$$

where $\pm$ is + or -.

3.5.4. *Abstraction – Lambda.*

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{lambda : } e : \textbf{unit} \to \tau} \text{ (Abs1)}$$

$$\frac{\sigma \neq \textbf{unit} \qquad \Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \texttt{lambda } x\texttt{: } e : \sigma \to \tau} \text{ (Abs2)}$$

$$\frac{\Gamma, x_i : \sigma_i \vdash e : \tau}{\Gamma \vdash \texttt{lambda } x_1, \ldots, x_n\texttt{: } e : \langle \sigma_1, \ldots, \sigma_n \rangle \to \tau} \text{ (Abs3)}$$

3.5.5. *Conditional Expression – IfExp.*

$$\frac{\Gamma \vdash e_0 : \tau \qquad \Gamma \vdash e_1 : \textbf{bool} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_0 \texttt{ if } e_1 \texttt{ else } e_2 : \tau} \text{ (If-Exp)}$$

3.5.6. *Comparisons – Compare.*

$$\frac{}{\Gamma \vdash e_0 \circ e_1 : \textbf{bool}} \text{ (Eqlty)}$$

where $\circ$ is ==, !=, is, or is not. See Chapter 4, Section 7.7 for a discussion of the broad nature of this rule.

$$\frac{\Gamma \vdash e_0 : \textbf{int/float/str/unicode} \qquad \Gamma \vdash e_1 : \textbf{int/float/str/unicode}}{\Gamma \vdash e_0 \bullet e_1 : \textbf{bool}} \text{ (Ineqlty)}$$

where $\bullet$ is <, <=, >, or >=.

$$\frac{n > 1 \qquad \Gamma \vdash e_0 \oplus_1 e_1 : \textbf{bool} \qquad \Gamma \vdash e_1 \oplus_2 e_2 \cdots \oplus_n e_n : \textbf{bool}}{\Gamma \vdash e_0 \oplus_1 e_1 \cdots \oplus_n e_n : \textbf{bool}} \text{ (Comp-Chain)}$$

where $\oplus$ is ==, !=, is, is not, <, <=, >, or >=. n represents the number of comparisons (which is one lower than the number of elements being compared). For example, when $n = 2$, $e_0 \oplus_1 e_1 \cdots \oplus_n e_n = e_0 \oplus_1 e_1 \oplus_2 e_2$ and $e_1 \oplus_2 e_2 \cdots \oplus_n e_n = e_1 \oplus e_2$.

3.5.7. *Application – Call.*

$$\frac{\Gamma \vdash f : \textbf{unit} \to \tau}{\Gamma \vdash f() : \tau} \text{ (App1)}$$

$$\frac{\Gamma \vdash f : \sigma \to \tau \qquad \Gamma \vdash x : \sigma}{\Gamma \vdash f(x) : \tau} \text{ (App2)}$$

$$\frac{n > 1 \qquad \Gamma \vdash f : \sigma \to \tau \qquad \Gamma \vdash \langle x_1, \ldots, x_n \rangle : \sigma}{\Gamma \vdash f(x_1, \ldots, x_n) : \tau} \text{ (App3)}$$

Note that the σ metavariable appears in the conditions but not the conclusions of App2 and App3 and so these rules are not syntax directed. As we will discuss in Chapter 4, Section 7.9, this currently restricts us to typechecking application of identifiers.

3.5.8. *Numeric Literals – Num.*

$$\frac{\texttt{type}(n)\ \texttt{is int/float}}{\Gamma \vdash \mathsf{Num}(n) : \textbf{int/float}} \text{ (Num)}$$

3.5.9. *String Literals – Str.*

$$\frac{\texttt{type}(s)\ \texttt{is str/unicode}}{\Gamma \vdash \mathsf{Str}(s) : \textbf{str/unicode}} \text{ (Str)}$$

3.5.10. *Subscription – Subscript.*

Indexing.

$$\frac{\Gamma \vdash s : \textbf{str/unicode} \qquad \Gamma \vdash e : \textbf{int}}{\Gamma \vdash s\,[e] : \textbf{str/unicode}} \text{ (Str-Idx)}$$

$$\frac{\Gamma \vdash l : \tau\ \textbf{list} \qquad \Gamma \vdash e : \textbf{int}}{\Gamma \vdash l\,[e] : \tau} \text{ (Lst-Idx)}$$

$$\frac{\Gamma \vdash t : \langle \tau_1, \ldots, \tau_n \rangle \qquad \texttt{type}(m)\ \texttt{is int} \qquad -n \le m < n \qquad \tau_{f(m)} = \tau}{\Gamma \vdash t\,[\mathsf{Num}(m)] : \tau} \text{ (Tup-Idx)}$$

$$f(m) = \begin{cases} m & m \ge 0 \\ m + n & m < 0 \end{cases}$$

Slicing.

$$\frac{\Gamma \vdash c : \textbf{str/unicode/}\tau\ \textbf{list} \qquad \Gamma \vdash e_0 : \textbf{int} \qquad \Gamma \vdash e_1 : \textbf{int} \qquad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash c\,[e_0 : e_1 : e_2] : \textbf{str/unicode/}\tau\ \textbf{list}} \text{ (Flat-Slc)}$$

where e_0, e_1, and e_2 are each optional. This means there are 8 extra variants of this rule, corresponding to each e being excluded or not. If an e_i is excluded from $s\,[e_0 : e_1 : e_2]$ (e.g., $s\,[: e_1 : e_2]$ or $s\,[e_1 : : e_2]$), then its corresponding condition is also excluded.

$$\frac{\Gamma \vdash t : \langle \tau_1, \ldots, \tau_n \rangle \qquad \texttt{type}(a)\ \texttt{is int} \quad \texttt{type}(b)\ \texttt{is int} \quad \texttt{type}(c)\ \texttt{is int}}{\Gamma \vdash t\,[\mathsf{Num}(a) : \mathsf{Num}(b) : \mathsf{Num}(c)] : \langle\, \tau_{h(a,b,c,i)}{}^{\,i \in f(a,b,c) \ldots g(a,b,c)} \,\rangle} \text{ (Tup-Slc)}$$

Like all other forms of slicing, empty and negative values are permitted for the upper, lower, and step parameters; unlike string and list subscription, tuple subscription requires a careful study of Python's slicing mechanics. These mechanics are encapsulated by the functions f, g, and h, which describe the relevant τ_i selected $\langle \tau_1, \ldots, \tau_n \rangle$. These mechanics are fairly complex, given that each of a, b, and c can be positive, negative, or absent; we describe all of these cases and how the f, g, and h functions are affected in Chapter 4, Section 5. The functions for the simple (and most common) case of this type assignment rule, where each slice parameter is present and positive, is:

$$\begin{aligned} f(a,b,c) &= 0 \\ g(a,b,c) &= \left\lfloor \frac{b-a}{|c|} \right\rfloor \\ h(a,b,c,i) &= a + ci \end{aligned}$$

The effect of absent slice parameters on f, g, and h will be discussed in Chapter 4, but it should be noted that, like STR-SLC and LSLC, the relevant `type(x) is int` condition is excluded when a parameter is absent.

3.5.11. *Identifiers – Name.*

$$\frac{}{\Gamma \vdash \mathsf{Name}(\texttt{"True"/"False"/"None"}) : \textbf{bool/bool/unit}} \text{ (BASE-ENV)}$$

$$\frac{}{\Gamma, x : \tau \vdash \mathsf{Name}(x) : \tau} \text{ (IDN)}$$

3.5.12. *List Construction – List.*

$$\frac{\Gamma \vdash e_i : \tau}{\Gamma \vdash [e_1, \ldots, e_n] : \tau \textbf{ list}} \text{ (LST)}$$

3.5.13. *Tuple construction – Tuple.*

$$\frac{\Gamma \vdash e_i : \tau_i}{\Gamma \vdash (e_1, \ldots, e_n) : \langle \tau_1, \ldots, \tau_n \rangle} \text{ (TUP)}$$

3.6. Polymorphism Typing Rules.

$$\frac{\Gamma \vdash e : \forall \alpha . \tau}{\Gamma \vdash e : \tau[\alpha \mapsto \sigma]} \text{ (INST)}$$

4. Polymorphism

4.1. Overview. Various calculi and programming languages employ many different varieties of polymorphism, and the word on its own can take on various meanings in different contexts. As Pierce [2002] describes, the two most prevalent forms of polymorphism are *ad-hoc polymorphism* and *parametric polymorphism*. Ad-hoc polymorphism allows a value of polymorphic type to evaluate differently conditional on the type it is used as; this frequently appears as function (or method) overloading in languages such as Java. Parametric polymorphism allows us to generically type a single piece of code that will perform identically for each possible type; a lack of generic typing would require us to repeat identical code with different types ascribed. Parametric polymorphism itself varies among languages; we have based Pyty's polymorphism on the variant found in ML—known as *let-polymorphism* because of its use through the ML `let` construct. We will first provide a brief description of let-polymorphism as found in ML, and then describe how Pyty's system differs.

Let-polymorphism provides the polymorphic use of terms bound through the `let` construct. As a motivational example, consider the simple identity function in ML, `fn x => x`, for which ML's type inference algorithm will compute the type `'a -> 'a`, where `'a` is a type variable as we have seen outlined in Pyty's type system. We can use this function with an argument of any type; for example,

```
(fn x => x) 5.0
```

will evaluate to `5.0` and

```
(fn x => x) true
```

will evaluate to `true`. However, the function is not being used polymorphically in these instances; instead, the ML type inference algorithm generates the type variable `'a` to represent the function domain and then restricts `'a` to the type of the argument passed. The effect of this difference becomes clear if we try to use the same function expression with two different input types. For example,

```
(fn f => (f 2, f 2.0)) (fn x => x)
```

will not type correctly because the type variable representing the input of `f` is restricted to both `int` and `real`. However, as stated before, ML provides parametric polymorphism through the `let` construct. If `fn x => x` is bound to `f` through a `let` expression, then it can be used polymorphically; the expression

```
let val f = fn x => x in (f 2, f 2.0) end
```

will type correctly. Because value bindings in ML are treated as nested `let` expressions, this allows us to easily declare functions with polymorphic type as follows:

```
1 fun f x = x;
2 val a = f 2;
3 val b = f 2.0;
4 val c = (f 2, f 2.0);
```

One restriction of the type inference algorithm employed by ML that we have not discussed is that polymorphic functions cannot be used as arguments to other functions. For example, the expression

```
let val f = x => x in (fn g => (g 2, g 2.0)) f end
```

is not valid in ML. This is because, as we will see in Section 4.2.1, traditional type assignment systems for the lambda calculus only include a rule to assign a type of form $\tau \to \tau'$ to a lambda abstraction, which means we can have an abstraction which is universally quantified, but not a domain which is quantified on its own. Following our example, we cannot assign the type $(\forall\alpha.(\alpha \to \alpha)) \to (\textbf{int} \times \textbf{real})$ to `fn g => (g 2, g 2.0)`. Second-order polymorphism refers to systems that include arbitrary nesting of arrow and quantified types; Wells [1998] has shown that typechecking for second-order polymorphic systems is undecidable.

Although Python lacks a `let` construct, we are treating function definitions in Python as value bindings are treated in ML. With appropriate type assignment rules, we can build this into Pyty without needing to directly model `let` expressions. However, practical concerns

(discussed in Section 4.2.3) restrict us to only allow this for assignment directly following type declarations. We present the Pyty equivalent of the ML example we have been following:

```
1 f = lambda x: x       #: f : 'a -> 'a
2 a = f(2)              #: a : int
3 b = f(2.0)            #: b : float
4 c = (f(2), f(2.0))    #: c : (int, float)
```

4.2. Typing Judgments for Polymorphic Types. We have explained the behavior of ML's let-polymorphism and Pyty's related system, but we will now proceed to illustrate how the typing judgment systems of ML and Pyty give rise to this behavior.

4.2.1. *Damas and Milner.* Pottier and Rémy [2005] provide a thorough definition of the ML type system originally defined by Damas and Milner [1982]. Below are the relevant typing rules found in Pottier and Rémy's discussion (with note that, like in our assignment rules, γ is a metavariable ranging over type schemes and τ over types). Note that lambda calculus notation is used, but this relates very easily with ML code.

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'} \text{ (DM-Abs)}$$

$$\frac{\Gamma \vdash e_0 : \tau \to \tau' \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0\, e_1 : \tau'} \text{ (DM-App)}$$

$$\frac{\Gamma \vdash e_0 : \gamma \qquad \Gamma, z : \gamma \vdash e_1 : \tau}{\Gamma \vdash \texttt{let } z\texttt{=}e_0 \texttt{ in } e_1 \texttt{ end} : \tau} \text{ (DM-Let)}$$

$$\frac{\Gamma \vdash e : \tau \qquad \alpha_i \text{ does not appear free in } \Gamma}{\Gamma \vdash e : \forall \alpha_0, \ldots, \alpha_{n-1}.\tau} \text{ (DM-Gen)}$$

$$\frac{\Gamma \vdash e : \forall \alpha_0, \ldots, \alpha_{n-1}.\tau}{\Gamma \vdash e : \tau[\alpha_i \mapsto \tau_i']} \text{ (DM-Inst)}$$

Now, let us explore what kind of type assignment these typing rules allow. First, consider $(\lambda x.x)(5)$. We can easily prove that we can assign integer type to this expression:

$$\frac{\dfrac{\dfrac{}{\Gamma, x : \textbf{int} \vdash x : \textbf{int}}}{\Gamma \vdash \lambda x.x : \textbf{int} \to \textbf{int}} \qquad \dfrac{\dfrac{}{\Gamma(5) = \textbf{int}}}{\Gamma \vdash 5 : \textbf{int}}}{\Gamma \vdash (\lambda x.x)(5) : \textbf{int}}$$

Note that the treatment of the numeric literal differs from Pyty's system; here we have types for literals stored in a base environment. We can also easily adapt the expression to pass an argument of any type and adapt the proof to show that the resulting expression has that type.

We can derive many types for $\lambda x.x$. For example,

$$\frac{\dfrac{}{\Gamma, x : \textbf{bool} \vdash x : \textbf{bool}}}{\Gamma \vdash \lambda x.x : \textbf{bool} \to \textbf{bool}} \qquad \frac{\dfrac{}{\Gamma, x : \textbf{int} \to \textbf{int} \vdash x : \textbf{int} \to \textbf{int}}}{\Gamma \vdash \lambda x.x : (\textbf{int} \to \textbf{int}) \to \textbf{int} \to \textbf{int}}$$

As we described before, we expect to be unable to assign a type to $(\lambda f.(f\,2, f\,2.0))(\lambda x.x)$. Our discussion before focused on type inference and the constraint system used for its algorithm, but we will now show that this typing system makes it impossible to assign a type to the expression. Let us try to form a derivation to assign $\langle \textbf{int}, \textbf{real} \rangle$ to the expression, since this is the only reasonable type we can expect.

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{f : \tau \to \tau \vdash f\ 2 : \textbf{int}} \quad \dfrac{\dots}{f : \tau \to \tau \vdash f\ 2.0 : \textbf{real}}}{f : \tau \to \tau \vdash (f\ 2, f\ 2.0) : \langle \textbf{int}, \textbf{real} \rangle}}{\vdash \lambda f.(f\ 2, f\ 2.0) : (\tau \to \tau) \to \langle \textbf{int}, \textbf{real} \rangle} \quad \dfrac{\dfrac{}{x : \tau \vdash x : \tau}}{\vdash \lambda x.x : \tau \to \tau}}{\vdash (\lambda f.(f\ 2, f\ 2.0))(\lambda x.x) : \langle \textbf{int}, \textbf{real} \rangle}$$

At this point, there is no type that we can fill in for τ that will allow us to satisfy both $f : \tau \to \tau \vdash f\ 2 : \textbf{int}$ and $f : \tau \to \tau \vdash f\ 2.0 : \textbf{real}$. In each of the steps shown already, we were restricted from introducing any type schemes. Attempting to introduce a type scheme through DM-INST at any of the intermediary steps may rename some type variables, but leaves us ultimately with the same dilemma at the top of the proof tree. Also, the proof never gets to use the **int** or **real** information, so we would run into the same problem attempting to assign any two-element tuple type. Additionally, we would run into more issues if we attempted to assign any non-tuple type or a longer or shorter tuple type. Therefore, we can conclude that no type can be assigned to the expression.

Now let us see how the situation changes if we bind $\lambda x.x$ to f through a let-binding instead of an abstraction and application.

$$\frac{\dfrac{\dfrac{\dfrac{}{x : \alpha \vdash x : \alpha}}{\vdash \lambda x.x : \alpha \to \alpha} \quad \dfrac{}{\alpha \notin \emptyset}}{\vdash \lambda x.x : \forall \alpha.\alpha \to \alpha} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash f : \forall \alpha.\alpha \to \alpha}}{\Gamma_1 \vdash f : \textbf{int} \to \textbf{int}} \quad \vdots}{\Gamma_1 \vdash f\ 2 : \textbf{int}} \quad \dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash f : \forall \alpha.\alpha \to \alpha}}{\Gamma_1 \vdash f : \textbf{real} \to \textbf{real}} \quad \vdots}{\Gamma_1 \vdash f\ 2.0 : \textbf{real}}}{\Gamma_1 \vdash (f\ 2, f\ 2.0) : \langle \textbf{int}, \textbf{real} \rangle}}{\Gamma_0 \vdash \texttt{let } f\texttt{=}\lambda x.x \texttt{ in } (f\ 2, f\ 2.0) \texttt{ end} : \langle \textbf{int}, \textbf{real} \rangle}$$

where Γ_0 is the initial environment which includes the types of literals, and $\Gamma_1 = \Gamma_0, f : \forall\alpha.\alpha \to \alpha$. We have also omitted weakening steps to hide environment information where possible to save horizontal space. The $\vdots$ indicate that it is trivial to show $\Gamma_1 \vdash 2 :$ **int** and $\Gamma_1 \vdash 2.0 :$ **real**. So we have shown how the type system outlined by Pottier and Rémy supports ML's let-polymorphism.

4.2.2. *Damas and Milner Variant.* We will now consider a variation on the Damas and Milner system, and will show how this variation has helpful properties for typechecking and retains the necessary properties of the original type system. In the following discussion, we will refer to the original Damas and Milner system laid out by Pottier and Rémy as DM, and to our modified variant of DM as DM1.

The new rules for abstraction, application, and instantiation (DM1-Abs, DM1-App, and DM1-Inst, respectively) are identical to the original Damas and Milner rules (DM-Abs, DM-App, and DM-Inst respectively). There is no DM1-Gen. We define a new version of the let rule:

$$\frac{\Gamma \vdash e_0 : \tau' \qquad \Gamma, z : \forall\tau' \vdash e_1 : \tau}{\Gamma \vdash \texttt{let } z\texttt{=}e_0 \texttt{ in } e_1 \texttt{ end} : \tau} \text{ (DM1-Let)}$$

Note that DM1 lacks any rules allowing us to assign a type scheme, since no quantified types appears in the conclusion of a rule. This is in line with the fact that quantified types are not in the grammar of types that a user can specify, as discussed in Section 2.

This system is preferable for use in a typechecking implementation, particularly because the lack of DM-Gen and the fact that we cannot assign a type scheme will have helpful properties. It is desirable for the assignment rules of a type system to be *syntax-directed*, where the form of a claim $\Gamma \vdash e : \tau$ corresponds to at most one assignment rule that can derive that claim—this allows us to build a typechecking implementation by simply reading the relevant rules from bottom up. The PT type system has many complications in this regard, and in Chapter 4 we discuss the methods we employ to circumvent these issues. DM-Inst worsens the situation because every condition of the form $\Gamma \vdash e : \tau$ can be the conclusion of DM-Inst (which itself has an infinite number of possible variants that could be applied). However, we can show that in DM1 type assignment derivations can be

arranged such that all instances of DM1-INST act on identifiers and appear directly below leaves of the proof tree, where the $\Gamma \vdash e : \forall \bar{\alpha}.\tau$ condition reduces to environment look-up. Thus, the typechecker need only consider the possibility of applying DM1-INST when checking an identifier.

Theorem 1. *$\Gamma \vdash e : \tau$ is derivable in DM if and only if it is derivable in DM1.*

PROOF. Lemma 1 proves the necessary and Lemma 3 the sufficient conditions. □

Lemma 1. *If $\Gamma \vdash e : \tau$ is derivable in DM, then $\Gamma \vdash e : \tau$ is derivable in DM1.*

PROOF. By induction on the length of the derivation.

The base case is trivial because DM and DM1 do not differ for any type assignment rules that can appear as leaves of a derivation.

The inductive hypothesis is that, if $\Gamma \vdash e : \tau$ is derivable in DM with height n, it is also derivable in DM1.

Now we consider a derivation of $\Gamma \vdash e : \tau$ in DM with height $n + 1$. The final rule in the derivation cannot be DM-GEN because τ is not a type scheme. Suppose the final rule is DM-ABS or DM-APP; then we can apply the inductive hypothesis for the premises and apply DM1-ABS or DM1-APP respectively to form a derivation in DM1.

Suppose the final rule is DM-INST (for the purposes of this case, we are trying to derive $\Gamma \vdash e : \tau[\bar{\alpha} \mapsto \bar{\tau}']$). Then the DM-INST condition (which assigns a type scheme) must be satisfied by DM-GEN since this is the only rule that can assign a type scheme, as follows:

$$\dfrac{\dfrac{\Gamma \vdash e : \tau(\bar{\alpha}) \qquad \bar{\alpha} \text{ not free in } \Gamma}{\Gamma \vdash e : \forall \bar{\alpha}.\tau}\ \text{(DM-GEN)}}{\Gamma \vdash e : \tau[\bar{\alpha} \mapsto \bar{\tau}']}\ \text{(DM-INST)}$$

And we can apply the inductive hypothesis to build a DM1 proof of the $\Gamma \vdash e : \tau(\bar{\alpha})$ condition. Since $\bar{\alpha}$ does not appear free in Γ, the variables can be replaced without effecting any behavior, so it is clear that $\Gamma \vdash e : \tau(\bar{\alpha})$ and $\bar{\alpha}$ not free in Γ imply $\Gamma \vdash e : \tau[\bar{\alpha} \mapsto \bar{\tau}']$, so we have built a DM1 derivation.

Suppose the final rule is DM-LET, which means the end of the derivation looks like:

$$\frac{\Gamma \vdash e_0 : \gamma \qquad \Gamma, z : \gamma \vdash e_1 : \tau}{\Gamma \vdash \texttt{let } z\texttt{=}e_0 \texttt{ in } e_1 \texttt{ end} : \tau}\ (\text{DM-LET})$$

We can form a DM1 derivation of both of the premises by the inductive hypothesis. We have two possible cases corresponding to whether γ is trivially quantified or not—$\gamma = \tau'$ or $\gamma = \forall\bar{\alpha}.\tau'$. If $\gamma = \tau'$, then, with the help of Lemma 2, we can form a DM1 derivation:

$$\frac{\dfrac{\text{IH}}{\Gamma \vdash e_0 : \tau'} \qquad \dfrac{\dfrac{\text{IH}}{\Gamma, z : \tau' \vdash e_1 : \tau}\ (\text{LEMMA 2})}{\Gamma, z : \forall\tau' \vdash e_1 : \tau}}{\Gamma \vdash \texttt{let } z\texttt{=}e_0 \texttt{ in } e_1 \texttt{ end} : \tau}\ (\text{DM1-LET})$$

If $\gamma = \forall\bar{\alpha}.\tau'$, then DM-GEN is the only assignment rule that could satisfy the $\Gamma \vdash e_0 : \gamma$ condition, so the end of our original DM derivation must look like:

$$\frac{\dfrac{\Gamma \vdash e_0 : \tau'(\bar{\alpha}) \qquad \bar{\alpha} \text{ not free in } \Gamma}{\Gamma \vdash e_0 : \forall\bar{\alpha}.\tau'}\ (\text{DM-GEN}) \qquad \Gamma, z : \forall\bar{\alpha}.\tau' \vdash e_1 : \tau}{\Gamma \vdash \texttt{let } z\texttt{=}e_0 \texttt{ in } e_1 \texttt{ end} : \tau}\ (\text{DM-LET})$$

And we can form a DM1 derivation of $\Gamma \vdash e_0 : \tau'(\bar{\alpha})$ and $\Gamma, z : \forall\bar{\alpha}.\tau' \vdash e_1 : \tau$ by the inductive hypothesis, so we can form a DM1 derivation:

$$\frac{\dfrac{\text{IH}}{\Gamma \vdash e_0 : \tau'(\bar{\alpha})} \qquad \dfrac{\text{IH}}{\Gamma, z : \forall\bar{\alpha}.\tau' \vdash e_1 : \tau}}{\Gamma \vdash \texttt{let } z\texttt{=}e_0 \texttt{ in } e_1 \texttt{ end} : \tau}\ (\text{DM1-LET})$$

□

Lemma 2. *If $\Gamma, x : \sigma(\bar{\alpha}) \vdash e : \tau$ is derivable in DM1, then $\Gamma, x : \forall\bar{\alpha}.\sigma \vdash e : \tau$ is also derivable in DM1.*

PROOF. By induction on the length of the derivation.

The base case would be

$$\frac{}{\Gamma, y : \tau,\ x : \sigma(\bar{\alpha}) \vdash y : \tau}$$

in which case we can clearly replace $x : \sigma(\bar{\alpha})$ with $x : \forall\bar{\alpha}.\sigma$ without effecting the type assignable to y, or

$$\frac{}{\Gamma, x : \sigma(\bar{\alpha}) \vdash x : \sigma(\bar{\alpha})}$$

in which case we can derive $\Gamma, x : \forall\bar{\alpha}.\sigma \vdash x : \sigma(\bar{\alpha})$ as follows:

$$\dfrac{\dfrac{}{\Gamma, x : \forall\bar{\alpha}.\sigma \vdash x : \forall\bar{\alpha}.\sigma}}{\Gamma, x : \forall\bar{\alpha}.\sigma \vdash x : \sigma(\bar{\alpha})}\ \text{(DM1-Inst)}$$

The inductive hypothesis is that, if $\Gamma, x : \sigma(\bar{\alpha}) \vdash e : \tau$ is derivable in DM1 with height n, then $\Gamma, x : \forall\bar{\alpha}.\sigma \vdash e : \tau$ is also derivable in DM1.

Now we consider a derivation of $\Gamma, x : \sigma(\bar{\alpha}) \vdash e : \tau$ in DM1 with height $n+1$. If e is an identifier, then we are in one of the base cases. If e is an abstraction, application, or let-expression, then the inductive hypothesis can be applied straightforwardly to the premises. For example, if $e = \texttt{let } z\texttt{=}e_0 \texttt{ in } e_1 \texttt{ end}$, then the end of the derivation looks like

$$\dfrac{\Gamma, x : \sigma(\bar{\alpha}) \vdash e_0 : \tau' \qquad \Gamma, x : \sigma(\bar{\alpha}), z : \forall\tau' \vdash e_1 : \tau}{\Gamma, x : \sigma(\bar{\alpha}) \vdash \texttt{let } z\texttt{=}e_0 \texttt{ in } e_1 \texttt{ end} : \tau}\ \text{(Dm1-Let)}$$

and the inductive hypothesis applies to both premises, so we can form a derivation:

$$\dfrac{\dfrac{\text{IH}}{\Gamma, x : \forall\bar{\alpha}.\sigma \vdash e_0 : \tau'} \qquad \dfrac{\text{IH}}{\Gamma, x : \forall\bar{\alpha}.\sigma, z : \forall\tau' \vdash e_1 : \tau}}{\Gamma, x : \forall\bar{\alpha}.\sigma \vdash \texttt{let } z\texttt{=}e_0 \texttt{ in } e_1 \texttt{ end} : \tau}$$

and the abstraction and application cases follow similarly.

By Theorem 2, DM1-Inst can only occur directly below leaves of the derivation, so if DM1-Inst occurs it must be like so (for the purposes of this case, we are trying to derive $\Gamma, x : \sigma(\bar{\alpha}) \vdash e : \tau[\bar{\beta} \mapsto \bar{\tau}']$):

$$\dfrac{\dfrac{}{\Gamma, y : \forall\bar{\beta}.\tau, x : \sigma(\bar{\alpha}) \vdash y : \forall\bar{\beta}.\tau}}{\Gamma, y : \forall\bar{\beta}.\tau, x : \sigma(\bar{\alpha}) \vdash y : \tau[\bar{\beta} \mapsto \bar{\tau}']}\ \text{(DM1-Inst)}$$

in which case we can clearly replace $x : \sigma(\bar{\alpha})$ with $x : \forall\bar{\alpha}.\sigma$ without effect. □

Lemma 3. *If $\Gamma \vdash e : \tau$ is derivable in DM1, then $\Gamma \vdash e : \tau$ is derivable in DM*

Proof. By induction on the length of the derivation.

The base case is trivial because DM1 and DM do not differ for any type assignment rules that can appear as leaves of a derivation.

The inductive hypothesis is that, if $\Gamma \vdash e : \tau$ is derivable in DM1 with height n, it is also derivable in DM.

Now we consider a derivation of $\Gamma \vdash e : \tau$ in DM1 with height $n+1$. Suppose the final rule in the derivation is DM1-Abs or DM1-App; then we can apply the inductive hypothesis for the premises and apply DM-Abs or DM-App respectively to form a derivation in DM.

Suppose the final rule is DM1-Inst (for the purposes of this case, we are trying to derive $\Gamma \vdash e : \tau[\bar{\alpha} \mapsto \bar{\tau}']$). By Theorem 2, DM1-Inst can only occur directly below leaves of a derivation, so our DM1 derivation must look like:

$$\dfrac{\dfrac{}{\Gamma, x : \forall \bar{\alpha}.\tau \vdash x : \forall \bar{\alpha}.\tau}}{\Gamma, x : \forall \bar{\alpha}.\tau \vdash x : \tau[\bar{\alpha} \mapsto \bar{\tau}']}\ \text{(DM1-Inst)}$$

and we can replace DM1-Inst with DM-Inst to get a valid DM derivation.

Suppose the final rule is DM1-Let, which means $e = \texttt{let } z\texttt{=}e_0 \texttt{ in } e_1 \texttt{ end}$ and the end of the derivation looks like:

$$\dfrac{\Gamma \vdash e_0 : \tau' \qquad \Gamma, z : \forall \tau' \vdash e_1 : \tau}{\Gamma \vdash \texttt{let } z\texttt{=}e_0 \texttt{ in } e_1 \texttt{ end} : \tau}\ \text{(Dm1-Let)}$$

We can form a DM derivation of both of the premises by the inductive hypothesis. And then we can form a DM derivation for the let-expression:

$$\dfrac{\dfrac{\dfrac{\text{IH}}{\Gamma \vdash e_0 : \tau'} \qquad \dfrac{\textit{ftv}(\tau') \text{ do not appear free in } \Gamma}{\Gamma \vdash e_0 : \forall \tau'}\ \text{(DM-Gen)} \qquad \dfrac{\text{IH}}{\Gamma, z : \forall \tau' \vdash e_1 : \tau}}{\Gamma \vdash \texttt{let } z\texttt{=}e_0 \texttt{ in } e_1 \texttt{ end} : \tau}\ \text{(DM-Let)}$$

□

Theorem 2. *Every derivation of DM1 that is concluded with an instance of* DM1-Inst *must have height 2.*

Proof. Suppose we have an instance of DM1-Inst which looks like:

$$\dfrac{\Gamma \vdash e : \forall \bar{\alpha}.\tau}{\Gamma \vdash e : \tau[\bar{\alpha} \mapsto \bar{\tau}']}\ \text{(Dm1-Inst)}$$

DM1 does not include any assignment rules that allow us to conclude that an expression can be assigned a type scheme, so this is only possible if e is an identifier and we have the following situation:

$$\dfrac{\dfrac{}{\Gamma, x : \forall \bar{\alpha}.\tau \vdash x : \forall \bar{\alpha}.\tau}}{\Gamma \vdash e : \tau[\bar{\alpha} \mapsto \bar{\tau}']}$$

So DM1-INST must occur directly below leaves of a derivation, and a derivation that concludes with an instance of DM1-INST has height 2.

□

4.2.3. *Pyty*. Pyty let-polymorphism is based on our variant of Damas and Milner (DM1). In the Pyty type system, assignments and function definitions are treated let-polymorphically if they immediately follow a type declaration. This is handled by the STMTS-LETA and STMTS-LETF assignment rules. We are only able to model assignment immediately following type declarations as let statements because this is the only situation where the information for the DM1-LET rule is immediately available. The issue arises from the fact that DM1-LET contains one condition that requires the use of τ', and one condition that requires the use of $\forall\tau'$; after we've encountered a declaration like `#: f: 'a -> 'a`, if we assign to `f` we want the value being assigned to be checked against $\alpha \to \alpha$, but when we encounter `f` in other expressions we want to look up its type as $\forall\alpha.(\alpha \to \alpha)$. One possible implementation could use parallel environment dictionaries and modified assignment and function definition rules to simulate this behavior. However, functions are the only Python terms which will have any value to be treated polymorphically. It is very rare to redefine functions in Python (in fact, static analysis tools such as PyChecker consider redefining a function to be an error), so it does not cost us much to not provide polymorphic behavior for a function defined multiple times.

The Python equivalent of $(\lambda f.(f\ 2, f\ 2.0))(\lambda x.x)$ is

```
(lambda f: (f 2, f 2.0))(lambda x: x)
```

While we can assign a type to this expression using the typing rules we have laid out, we will find that implementation details of the typechecking algorithm (as discussed in Chapter 4 Section 7) restrict Pyty from typechecking applications of abstraction expressions (that is, opposed to applications of identifiers which have been assigned an abstraction).

Figure 6 exhibits a derivation that shows we can use `f` polymorphically in

```
1 f = lambda x: x       #: f : 'a -> 'a
2 c = (f(2), f(2.0))    #: c : (int, float)
```

$$
\begin{aligned}
f &= \mathsf{Name}(\texttt{"f"}) \\
c &= \mathsf{Name}(\texttt{"c"}) \\
s_0 &= \mathsf{TypeDec}(\texttt{"f"}, \alpha \to \alpha) \\
s_1 &= \mathsf{Assign}(\texttt{lambda}\ \ x\texttt{:}\ x, [f]) \\
s_2 &= \mathsf{TypeDec}(\texttt{"c"}, \langle \mathbf{int}, \mathbf{float} \rangle) \\
s_3 &= \mathsf{Assign}((f(\mathsf{Num}(2)), f(\mathsf{Num}(2.0))), [c]) \\
\Gamma_0 &= f : \forall\alpha.(\alpha \to \alpha) \\
\Gamma_1 &= \Gamma_0, s : \langle \mathbf{int}, \mathbf{float} \rangle
\end{aligned}
$$

$$
\dfrac{\dfrac{}{f \notin \emptyset} \quad \dfrac{\dfrac{}{x : \alpha \vdash x : \alpha}}{\vdash \texttt{lambda}\ \ x\texttt{:}\ x : \alpha \to \alpha} \quad \dfrac{\dfrac{}{c \notin \emptyset} \quad \mathscr{D} \quad \dfrac{}{\Gamma_1 \vdash \mathsf{StmtList}() : \mathbf{stmt}^*}}{\Gamma_0 \vdash \mathsf{StmtList}(s_2, s_3) : \mathbf{stmt}^*}}{\vdash \mathsf{StmtList}(\mathsf{TypeDec}(f, \alpha \to \alpha), \mathsf{Assign}(\texttt{lambda}\ \ x\texttt{:}\ x, [f]), s_2, s_3) : \mathbf{stmt}^*}
$$

where we define $\mathscr{D}$:

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_0 \vdash f : \forall\alpha.(\alpha \to \alpha)}}{\Gamma_0 \vdash f : \mathbf{int} \to \mathbf{int}} \quad \dfrac{}{\vdash \mathsf{Num}(2) : \mathbf{int}}}{\Gamma_0 \vdash f(\mathsf{Num}(2)) : \mathbf{int}} \quad \dfrac{\dfrac{\dfrac{}{\Gamma_0 \vdash f : \forall\alpha.(\alpha \to \alpha)}}{\Gamma_0 \vdash f : \mathbf{float} \to \mathbf{float}} \quad \dfrac{}{\vdash \mathsf{Num}(2.0) : \mathbf{float}}}{\Gamma_0 \vdash f(\mathsf{Num}(2.0)) : \mathbf{float}}}{\Gamma_0 \vdash (f(\mathsf{Num}(2), \mathsf{Num}(2.0)) : \langle \mathbf{int}, \mathbf{float} \rangle}
$$

FIGURE 6. Typing derivation that exhibits a polymorphic function.

CHAPTER 4

Typechecking Implementation

1. Algorithm

In Chapter 3, we presented the PT type system and the system of deriving proofs for claims of the form $\Gamma \vdash t : \tau$, where t is a term in TPython and τ is a type in PT. Typechecking is the problem of determining whether a given term can be assigned a given type in a given type environment. In the context of our PT type assignment proof system, this means that `typecheck(`t`, `τ`, `Γ`)` should evaluate to true if some derivation exists for $\Gamma \vdash t : \tau$, and to false otherwise.

This naturally raises the issue of syntax-directedness. We say that the type assignment rules for a type system are *syntax directed* if at most one assignment rule exists for every form of conclusion. This means that, given a language term t, type τ, and environment Γ, only one type assignment rule could possibly allow us to conclude $\Gamma \vdash t : \tau$. For example, the simply typed lambda calculus as outlined by Pierce [2002, chap. 9] is syntax directed. A syntax directed type assignment system allows us to easily build a typechecking implementation by simply testing the conditions presented by the relevant assignment rule—this is valid because, if a derivation exists, it is built by applying the single relevant rule at every node in the derivation tree. Because the Python language (and thus TPython) overloads many operators, syntax-directedness is unachievable in designing PT. Many constructs in TPython and type forms in PT give us enough information to narrow down to one relevant assignment rule, but a few exceptions (discussed in Section 4) complicate things.

The most notable of these complications is the necessity for a limited implementation of type inference. This is because some assignment rules (e.g., ASSMT and the rules for subscription) require information about the form of types not included in the conclusion, and without type inference a typechecking implementation would be required to "guess"

types to check, which would lead the implementation to not halt when no type is assignable. In Section 6 we will discuss our implementation of limited type inference.

2. Pyty Code Overview

Figure 1 provides an overview for the structure of the Pyty implementation. The core of the typechecking implementation is contained in `check.py`. `pyty.py` is the entry point for the user, and allows the user to typecheck a provided file, to typecheck an expression written as a command-line argument against a type also provided as a command-line argument, or to infer the type of an expression written as a command-line argument. In typechecking or type inferring an expression, there can be no type declarations, so for both cases the process is fairly simple—create the relevant abstract syntax tree with Python's `ast` module, create a type object (if typechecking) using `ptype.py`, and defer to `check.py` or `infer.py`. If we are typechecking a file (the main use case), then we need to consider type declarations. Our type assignment rules, and their implementation in `check.py`, assume that type declarations exist as nodes in the TPython AST, so we need to add these nodes to the AST created by `ast` before we can pass off an AST to `check.py`—this process is described in Section 3.

2.1. Type Implementation. The typechecking implementation employs a class—`PType`—to represent types in the PT type system. `PType` instances are created by parsing string representations of types (e.g., `"int"` or `"[(int, float)]"`) and building up an abstract syntax tree of types. To perform the parsing, we use Lepl, a recursive decent parser for Python which supports grammars written directly in Python. In Chapter 3, Section 2.2, we described the types which can be parsed by our Lepl-generated type specification parser.

2.2. Typechecking Implementation Structure. As we noted in Chapter 3, Section 2.2, the **stmt** and **stmt*** abstract PT types are not exposed to the user via concrete type. A statement can only be assigned **stmt**, and a statement list can only be assigned **stmt***, so when we ask whether a statement or a statement list typechecks, it is implicit what type we want to check it against.

```
pyty/
|-- src/                        <- Source code.
|   |-- ast_extensions.py       <--- Extensions to ast.AST.
|   |-- check.py                <--- Typechecking.
|   |-- errors.py               <--- Custom errors.
|   |-- infer.py                <--- Type inference.
|   |-- logger.py               <--- Custom logging.
|   |-- parse_file.py           <--- File reading.
|   |-- ptype.py                <--- Types.
|   |-- pyty.py                 <--- Command-line interface.
|   |-- settings.py             <--- Settings.
|   |-- util.py                 <--- Multi-use utilities.
|-- test/                       <- Tests.
|   |-- generate_tests.py       <--- spec/ --> tests to core.
|   |-- ptype_unit_tests.py     <--- Unit tests for ptype.py.
|   |-- spec/                   <--- Unit test specs.
|   |   |-- expr_binop.spec     <----- expression specs.
|   |   |-- ...                 <----- expression specs.
|   |   |-- mod_if.spec         <----- module specs.
|   |   |-- ...                 <----- module specs.
|   |-- test_files/             <--- Generated test source.
|   |   |-- mod_if1.py          <----- individual modules.
|   |   |-- ...                 <----- individual modules.
|   |-- unit_tests_core.py      <--- Core test functions.
```

FIGURE 1. Pyty implementation file structure.

Broadly, the typechecking implementation exposes three functions—`check_expr`, `check_stmt`, and `check_stmt_list`. All three require a TPython AST node and an environment, and `check_expr` is the only one of the three that also requires a type. The implementation contains a function for each AST node (e.g., `check_BinOp_expr` or `check_Assign_stmt`), and `check_expr` and `check_stmt` determine the type of the AST node being considered and then defer to these specific functions. Each function then determines which rule is applicable, and applies it; in many cases, choosing and applying the relevant rule is straightforward, but in Section 4 we will discuss instances where this is particularly difficult. This strategy is only legitimate because the PT assignment rules are syntax directed up to the node type of the term—that is, we have a certain set of rules that are relevant to Subscript nodes, and we know that a Subscript node can only be assigned a type if one of those rules is applicable. The only potential caveat is the INST assignment rule, which is the only rule

with a generic *e* appearing in the conclusion and appears to be applicable for any expression. However, in Chapter 3, Section 4 we discussed how type assignment derivations in PT can always be rearranged such that applications of INST appear at leaves and applied to identifiers, so we know to only attempt to apply this rule when checking a Name node.

3. Type Declaration Preprocessing

We use Python's `ast` module to generate abstract syntax trees of Python source code, which discards information about comments. Since our type declarations sit in comments, this means we have to perform an extra set of steps to obtain an AST which contains type declaration nodes. These steps boil down to parsing the file for lines which contain `#:`, creating a list of type declarations (one from each of these lines), and then adding each of these type declarations to the AST based on the line numbers of the nodes. We define a `TypeDec` class, which is an AST node for type declarations (meeting the specification in Chapter 3), and imitates a native AST node. We also define a `TypeDecASTModule` class to store an AST module node—with the property that the tree of a `TypeDecASTModule` has been populated with the type declarations present in the module's source code. Most of these steps described are straightforward, the main exception being the step of placing a TypeDec node.

3.1. Placing Type Declarations. To place a TypeDec node into its proper place in the syntax tree, we rely on the line number information embedded in each AST node. To place a TypeDec, we traverse the AST to find the last statement node with a line number smaller than the type declaration's (which we will refer to as the *target node*), and then place the TypeDec directly after the target node. This presents an issue if the target node is the last statement in a compound statement's statement list (i.e., the last line of a block)—we do not know if the type declaration belongs directly after the target node within the same statement list, or directly after the block (either outside the compound statement, or at the beginning of the statement list of the next branch of the compound statement). In the example displayed in Listing 4.1, placing the type declarations for `f` and `x` is simple, but just given the line number of the `c` type declaration, we don't know whether it appeared within

```
#: f : [int] -> int
def f(a):
    #: x : int
    x = 0
    return x
    #: c : float

print f([1, 2, 3, 4, 5])
```

LISTING 4.1. Example of a situation which makes type declaration difficult.

the function block or not. However, since we treat type declarations with block scope, there is no value to adding a type declaration as the last line of a statement list, so we always assume the type declaration is meant to be placed after the block. This means there may be unexpected behavior if a type declaration is placed as the last line of a block. This situation could be remedied with deeper examination of column offsets of each statement to determine indentation levels, but this situation would arise so rarely in normal usage that we found it not worth the complication.

As described in Chapter 3, if a type declaration lives on the same line as a statement, then it will be placed before that statement. For example,

```
def f(a):       #: f : [int] -> int
  x = 0         #: x : int
  for y in a:   #: y : int
    x += y
  return x
```

will behave as desired, with the type of `f` declared before the function definition, the type of `x` before the assignment, and the type of `y` before the for loop.

4. Typing Rule Implementations

In this section, we will cover the nontrivial steps taken to combat a lack of syntax-directedness. In many cases, the relevant rule to apply and applying it are straightforward—the implementation in these cases should be self-explanatory translation of type assignment rules.

4.1. Statement Typing Rules.

4.1.1. *Function Definition – FunctionDef.* Determining which of FN-DEF1, FN-DEF2, or FN-DEF3 is relevant is syntax-directed, but each assignment rule has type metavariables which appear in the condition and not the conclusion (τ for FN-DEF1, σ and τ for FN-DEF2, and $\sigma_1, \ldots, \sigma_n, \tau$ for FN-DEF3). However, the name attribute of the FunctionDef is a string, and we mandate that the type for a function be declared before definition, so it is valid to look up the type of the function in the environment to determine the form of the $\Gamma \vdash f : \cdots$ condition, which is enough to uniquely determine the rest of the rule.

4.1.2. *Assignment – Assign.* Complications arise in implementing ASSMT because the τ_i type metavariables appear in the rule's premises and not in the conclusion. The only possible solution is to infer types for the t_i expressions to determine the τ_i. Fortunately, a limited subset of the Python language is allowed on the left-hand side of assignment—in TPython, this subset is Subscript, Name, List, and Tuple, and these are all nodes in our type-inferrable subset of TPython, so there are no added restrictions by employing type inference in this case. Since we already have to infer τ_i, it is fairly trivial to verify if t_i is Subscript and if the collection of the subscript has a tuple type to satisfy the side-condition.

4.1.3. *Augmented Assignment – AugAssign.* Like in the ASSMT rule, the τ type metavariable in the AUG-ASSMT rule is present in the premises and not in the conclusion, so we need to perform type inference on the expression of the left-hand side of the augmented assignment. As in typechecking an Assign node (Section 4.1.2), this is valid because all language constructs that are valid left-hand sides of assignment are handled by our type inference algorithm.

4.1.4. *For Loop – For.* We require type inference to determine the τ metavariable that only appears in the premises of FOR. But τ is the type of the loop variable, which is subject to the same left-hand side constraints discussed for typechecking Assign nodes in Section 4.1.2, so it is safe to perform type inference to determine τ.

4.1.5. *Expression Statement – Expr.* Because we restrict the function of applications to be an identifier, we can use environment look-up to determine the type of the τ type metavariable that does not appear in the EXPR conclusion.

4.2. Expression Typing Rules.

4.2.1. *Binary Operations – BinOp.* Determining which of the BinOp rules—FLAT-CAT, FLAT-REP, STR-FORM, TUP-CAT, or TUP-REP—to determine which rule to apply is straightforward, and then to apply the rule is also straightforward for most rules, except for TUP-CAT and TUP-REP. In TUP-REP, the m integer metavariable only appears in the conditions, and the only information we know about it is that $0 \leq m \leq n+1$, so we test if the conditions hold for *any* choice of m in that range. Implementing TUP-CAT is nontrivial because it requires unraveling the meaning of $\langle \tau_{i \bmod n}{}^{i \in 0 \ldots mn-1} \rangle$, which basically means that we have something that looks like $\langle \tau_0, \ldots, \tau_{n-1} \rangle + \langle \tau_0, \ldots, \tau_{n-1} \rangle + \cdots + \langle \tau_0, \ldots, \tau_{n-1} \rangle$ (m times). So we are given a tuple type τ with mn elements, and we know m, so we compute $n = mn/m$, then verify if e can be assigned the tuple consisting of the first n elements of τ, and then we check if τ looks like those first n elements repeated m times.

4.2.2. *Application – Call.* Applying APP1 is straightforward, but the σ type metavariable appears in the conditions of both APP2 and APP3 and not their conclusions. We resolve this with the restriction, discussed further in Section 7.9, that the function being called must be an identifier, which means we can look up its type in the environment. Once we restrict to identifier nodes and perform environment look-up, the rest of the implementation of APP2 and APP3 is straightforward.

4.2.3. *Subscription – Subscription.* It is trivial to determine whether we are dealing with an index or a slice, but from there we need to determine the type of the collection being subscribed to determine the relevant rule. This is the main area we run into trouble with type inference—unlike the other uses of type inference, which are all considered left-hand sides, arbitrary expressions can be subscribed. If we are dealing with an index node or a slice of a list or string, applying the relevant assignment rule once we know the type of the collection is straightforward. Things are much more complex if we have the slice of a tuple. Essentially, we have a function `slice_range` which encapsulates the logic of determining which indices will be hit by a given slice, and we then verify if the elements of our given type match the elements hit by those indices. Section 5 will go into more depth about this `slice_range` function.

5. Python Subscript Behavior

Tuples present an interesting typechecking challenge, because we need a very close understanding of certain evaluation mechanics to determine the type for expressions involving tuples. The biggest example of this is with tuple subscription—we need to understand most, if not all, of the mechanics behind slice evaluation to know what types will be swept out by a slice of a tuple.

5.1. Empty Slice Parameters in the AST. If we have an AST node representing `l[a:b:c]`, we have an `ast.Subscript` object with an `ast.Slice` attribute that has three children attributes: `lower`, `upper`, and `step`. If we exclude the lower or upper parameters (e.g., `l[:b:c]`, `l[a::c]`, or `l[::c]`), then our `ast.Slice` will contain empty attributes for those children nodes—`lower` or `upper` will contain the value `None`. However, if we exclude the step parameter (`l[a:b:]`), then our `ast.Slice`'s `step` parameter will be a reference to an AST node representing the `None` literal (i.e., an `ast.Name` node with identifier `"None"`).

If a slice attribute is empty, we want to interpret it as the default value for that kind of attribute (described in Section 5.2). However, we do not agree with the notion of `None` being a legitimate value for slice attributes (for example, `l[None:None]` is equivalent to `l[:]` in Python). Our main goal in typechecking is to catch errors before they would normally be detected; except in rare cases, providing `None` to a slice parameter seems like unintended behavior that most users would expect to throw an error. So we would like typechecking an expression like `l[None:None]` to fail. However, because of the `None` literal inserted in the step parameter described above, `l[a:b:]` and `l[a:b:None]` create identical ASTs; we would like `l[a:b:]` to be valid, so we must make a concession and also allow the `None` literal in the step parameter even if it was placed there by the user (not by the AST).

These considerations come into play in the implementation of typechecking and type inference when we are verifying that a subscript expression has a valid slice. We need to verify that each parameter is specified correctly or left blank, where "left blank" means no child

node for the upper and lower parameters and means no child node or a `None` literal child node for the slice parameter. Two functions in `util.py`, `slice_range(l, u, s, n)` and `valid_int_slice(l, u, s, env)`, handle this validation depending on the situation (`slice_range` is used to determine the actual range of values for a slice of integer literals and returns `None` if any of the parameters is not an integer literal; `valid_int_slice` is used to determine if the parameters all typecheck as integers and returns `False` if they do not).

5.2. Slice parameter interpretations. For the purposes of typechecking and type inferring lists and strings, we do not need to know the semantics of the slice operator; however, for tuple typechecking and type inferring, we need to construct a tuple type of the elements which are swept out by the slicing. So we require tuple parameters to be integer literals, and must determine the indices of the tuple which the slice hits; the `slice_range` function mentioned above handles this logic.

If a parameter is blank (as determined by the discussion above), default values are used. As the Python 2.7.2 expression reference [Python development team, 2012b] describes, 0 is the default value for the lower bound, `sys.maxint` for the upper bound, and 1 for the step. It is not an error for upper and lower bounds to be outside the range of the valid indices; we just do not select elements whose indices are outside the range. For example, if we have `t = (0, 1, 2)`, then `t[1:10]` should evaluate to `(1, 2)` and `t[10:20]` should evaluate to an empty tuple. Thus, the default upper bound (`sys.maxint`) will effectively go to the end of the list.

If either bound is negative, then the length of the collection is added to the bound and then this new bound is treated normally. For example, if `l` is a list of length 10, `l[-1:]` = `l[9:]` will refer to the list of just the tenth element, `l[-5:]` = `l[5:]` will refer to the list of the sixth element onward, and `l[-15:]` = `l[0:]` = `l[:]` will refer to the entire list because 10 is only added to -15 once (i.e., we don't get `l[-15:]` = `l[-5:]` = `l[5:]`). These semantics for treatment of negative values are also applied for simple indexing, except that indexing throws an error if the index is outside the collection's

valid indices (e.g., `l[-5]` = `l[5]` will be the 6th element of the list, but `l[-15]` will throw an `IndexError`).

If the step parameter is negative, then the indices are collected in the reverse order. `slice_range` implements this by using the absolute value of the step parameter to generate the range of indices and then reverses the list of indices if the parameter is negative.

Suppose we have a slice subscript `l[a:b:c]`, where `a`, `b`, and `c` are integer literals representing the integers a, b, and c respectively. We can think of the slice hitting all $x = a + ci$ where $a \leq x < b$ (or $x = b + ci$ if $c < 0$). To do this, we take advantage of Python's `range` built-in which uses a lower bound, upper bound, and step like slicing; however, to ensure the condition that each $a \leq x < b$, we first sanitize the lower and upper bounds which we pass to the `range` function. To sanitize each bound, we use the following logic:

function SANITIZE(a, n) ▷ a is the bound, n the length of collection
  if $a < -n$ **then**
    return 0
  else if $-n \leq a < 0$ **then**
    return $a + n$
  else if $n < a$ **then**
    return n
  else
    return a
  end if
end function

Default values for the parameters are also entered at this phase (instead of setting the upper bound to be `sys.maxint` and then seeing here that `n < sys.maxint`, the upper bound is just set to be `n` if it is blank).

In the language of the TUP-SLC assignment rule,

$$\begin{aligned} f(a,b,c) &= \text{SANITIZE}(a) \text{ if } a \text{ is not } \texttt{None} \text{ else } 0 \\ g(a,b,c) &= \text{SANITIZE}(b) \text{ if } b \text{ is not } \texttt{None} \text{ else } \texttt{sys.maxint} \\ h(a,b,c,i) &= f(a,b,c) + ci \text{ if } c > 0 \text{ else } g(a,b,c) + ci \end{aligned}$$

```
expr = Num(object n)
     | Str(string s)
     | Subscript(expr value, slice slice, expr_context ctx)
     | Name(identifier id, expr_context ctx)
     | List(expr* elts, expr_context ctx)
     | Tuple(expr* elts, expr_context ctx)

-----------------------------------------------------------

expr_context = Load | Store | Del | AugLoad | AugStore | Param
slice = Slice(expr? lower, expr? upper, expr? step)
      | Index(expr value)
```

FIGURE 2. ITPython abstract grammar. Nodes above the break can be inferred, and nodes below are informational.

6. Type Inference

6.1. Necessity for Type Inference. Section 2.1 described many instances where type-inferring expressions was necessary for typechecking, due to Python's overloaded syntax. Most uses of type inference are for expressions appearing as left-hand sides (i.e., in Assign, AugAssign, and For), and the Python abstract syntax specification [Python development team, 2012a] dictates that the only TPython nodes that can appear as left-hand sides are Subscript, Name, List, and Tuple, so this means that type inference can be useful without supporting the entire TPython language.

6.2. Type Inference Coverage. Figure 2 presents the subset of TPython for which we support type inference—which we will call inferable TPython, or ITPython.

6.3. Type Inference Algorithm. The type inference algorithm is very similar to the typechecking algorithm outlined in Section 1, with a few modifications. In Chapter 3, we presented the PT type system and the system of deriving proofs for claims of the form $\Gamma \vdash t : \tau$, where t is a term in TPython and τ is a type in PT. Given a term and a type environment, type inference is the problem of determining a type that can be assigned to

the term; in most contexts, it is considered the problem of finding the most general type, traditionally through unification. ITPython and its associated type assignment rules are simple enough that we can determine a type by building out a proof tree in "the obvious way" at each step, foregoing the constraint system and unification problem necessary for inferring types in the simply typed lambda calculus and ML.

The adaptations of assignment rules to our basic type inference algorithm is very straightforward. The biggest difference is that we are not given information about the type to check against, but within ITPython we have the power to look up the types of expressions in conditions.

7. Limitations

In this section, we will discuss some of Pyty's limitations—both in the form of language constructs which have not been included, reducing the subset of Python which Pyty supports, and in the form of restrictions imposed on the included constructs, making type assignment rules overly broad and thus limiting Pyty's error-checking power.

7.1. Missing Components. In Chapter 3, we presented the abstract grammar of the subset of the Python language which Pyty supports. Figures 3, 4, 5, and 6 present the full Python abstract grammar as specified by the Python abstract syntax tree documentation [2012a], with Pyty-supported constructs highlighted. As in Chapter 3, the abstract grammar is specified in the Zephyr Abstract Syntax Description Language as outlined by Wang et al. [1997].

Figure 3 presents the grammar of modules. Pyty only examines the abstract syntax trees of source code files, in which context Module nodes are the only relevant node (in practice, modules which solely contain an expression are still modeled as a Module node with one Expr statement, despite the ASDL specification including the Expression case). Figure 4 presents the grammar of statements, and Figure 5 presents the grammar of expressions. Figure 6 presents the grammar of the miscellaneous other language elements which we are not considering terms; we have not explicitly marked support for these constructs because

```
mod = Module(stmt* body)
    | Interactive(stmt* body)
    | Expression(expr body)
    | Suite(stmt* body)
```

FIGURE 3. Python abstract grammar for modules. Language constructs handled by Pyty are in bold.

Pyty's typechecking algorithm assigns types to terms, where these constructs are secondary attributes of terms.

7.2. Modules and Classes. Pyty does not yet include support for importing and interacting with modules. This is a very important feature because even very simple command-line scripts often rely on the `sys` module. To support modules, we will need to create a new *signature* type which represents the type of a module and acts very similarly to the labeled record type described by Pierce [2002, chap. 11].

Pyty also does not yet support classes, which is clearly an important feature of an object-oriented language. The trouble with user-defined classes arises by using class names to refer to the type of instances of that class; we also then need to store a mapping from type names to the signature type that class name stands in for. To this effect, we will need to redefine our notion of an environment to instead be the sum of a *type environment*, mapping program identifiers to types, and a *signature environment*, mapping type identifiers to signatures.

7.3. Abstract Types and `in`. Pyty currently does not support the `in` and `not in` binary operations because they raises the issue of abstract types. The correct type assignment rule for an `in` expression would be something like

$$\frac{\Gamma \vdash e_0 : \tau \qquad \Gamma \vdash e_1 : \textbf{collection of } \tau}{\Gamma \vdash e_0 \texttt{ in}|\texttt{not in } e_1 : \textbf{bool}} \text{ (IN)}$$

```
stmt = FunctionDef(identifier name, arguments args, stmt* body,
                   expr* decorator_list)
     | ClassDef(identifier name, expr* bases, stmt* body, expr* decorator_list)
     | Return(expr? value)
     | Delete(expr* targets)
     | Assign(expr* targets, expr value)
     | Print(expr? dest, expr* values, bool nl)
     | For(expr target, expr item, stmt* body, stmt* orelse)
     | While(expr test, stmt* body, stmt* orelse)
     | If(expr test, stmt* body, stmt* orelse)
     | With(expr context_expr, expr? optional_vars, stmt* body)
     | Raise(expr? type, expr? inst, expr? tback)
     | TryExpect(stmt* body, excepthandler* handlers, stmt* orelse)
     | TryFinally(stmt* body, stmt* finalbody)
     | Assert(expr test, expr? msg)
     | Import(alias* names)
     | ImportFrom(identifier? module, alias* names, int? level)
     | Exec(expr body, expr? globals, expr? locals)
     | Global(identifier* names)
     | Expr(expr value)
     | Pass | Break | Continue
```

FIGURE 4. Python abstract grammar for statements. Language constructs handled by Pyty are in bold.

but we have no simple way of formalizing what we mean by a **collection of** τ. We can ignore this concern, and implement the obvious rule for lists (where we replace collection of τ with τ **list**). But we will run into concerns when user-defined classes are supported, since a user could create a collection supporting the `in` operator by implementing a `__contains__()` or `__iter__()` method as described in the Python expression reference [Python development team, 2012b]. A signature type could possibly be used to represent collections.

We have also excluded comprehensions because they raise similar questions and issues.

```
expr = BoolOp(boolop op, expr* values)
     | BinOp(expr left, operator op, expr right)
     | UnaryOp(unaryop op, expr operand)
     | Lambda(arguments args, expr body)
     | IfExp(expr test, expr body, expr orelse)
     | Dict(expr* keys, expr* values)
     | Set(expr* elts)
     | ListComp(expr elt, comprehension* generators)
     | SetComp(expr elt, comprehension* generator)
     | DictComp(expr key, expr value, comprehension* generators)
     | Generator(expr elt, comprehension* generators)
     | Yield(expr? value)
     | Compare(expr left, cmpop* ops, expr* comparators)
     | Call(expr func, expr* args, keyword* keywords, expr? staragrs,
           expr? kwargs)
     | Repr(expr value)
     | Num(object n)
     | Str(string s)
     | Attribute(expr value, identifier attr, expr_context ctx)
     | Subscript(expr value, slice slice, expr_context ctx)
     | Name(identifier id, expr_context ctx)
     | List(expr* elts, expr_context ctx)
     | Tuple(expr* elts, expr_context ctx)
```

FIGURE 5. Python abstract grammar for expressions. Language constructs handled by Pyty are in bold.

7.4. Subscripting. As described in the discussion of their implementation, the subscription assignment rules overlap a significant amount because the operation itself is very overloaded. The only way to determine which rule is relevant, and what form of that rule to apply, in many cases is to perform type inference. This is a significant limitation, because arbitrary expressions can be subscripted in Python.

7.5. Expression Statements. To typecheck an expression, we need to have some type to check it against; since the expression in an expression statement really is not being

```
expr_context = Load | Store | Del | AugLoad | AugStore | Param
       slice = ~~Ellipsis~~ | Slice(expr? lower, expr? upper, expr? step)
             | ~~ExtSlice(slice* dims)~~
             | Index(expr value)
      boolop = And | Or
    operator = Add | Sub | Mult | Div | Pow | LShift
             | RShift | BitOr | BitXor | BitAnd | FloorDiv
     unaryop = Invert | Not | UAdd | USub
       cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | ~~In~~ | ~~NotIn~~
~~comprehension~~ = ~~(expr target, expr iter, expr* ifs)~~
~~excepthandler~~ = ~~ExceptHandler(expr? type, expr? name, stmt* body)~~
   arguments = (expr* args, ~~identifier? vararg, identifier? kwarg, expr* defaults~~)
   ~~keyword~~ = ~~(identifier arg, expr value)~~
     ~~alias~~ = ~~(identifier name, identifier? asname)~~
```

FIGURE 6. Miscellaneous non-term elements of Python abstract grammar. Elements explicitly not included in TPython are in struck out.

"used" in any specific way, we have no guess for what type to assign to it. An exception is when the expression is an application, since we store information about the function being applied. This is convenient because function application is the one practical case for writing an expression as a stand-alone statement.

7.6. String Formatting. The STR-FORM rule is very general because it requires information about the value of the first parameter—the value of the left-hand string determines what type the right-hand expression should have. So a static type assignment rule cannot possibly cover all the possible cases.

7.7. Equality Comparisons. The straightforward (and Java-like) assignment rule for equality comparisons would require assigning some type to both the left- and right-hand sides of the comparison; however, the type to assign to the sides is not syntax-directed, which makes this approach difficult for a typechecking algorithm. So we allow any expression to be compared for equality with any other expression—this is undesirable because it

means any errors within an equality comparison will not be found by Pyty because Pyty will blindly report that every equality comparison typechecks correctly.

7.8. For Loops. We do not have an assignment rule for for loops iterating over any collections other than lists. Sets and dictionaries are, obviously, missing because they are not present in the current language subset.

Tuples present a rather difficult challenge. A logical type assignment rule for for loops iterating over tuples would be as follows:

$$\frac{\Gamma \vdash x : \tau_i \qquad \Gamma \vdash e : (\tau_0, \ldots, \tau_n) \qquad \Gamma, x : \tau_i \vdash b_0 : \textbf{stmt*} \qquad \Gamma \vdash b_1 : \textbf{stmt*}}{\Gamma \vdash \mathsf{For}(x, e, b_0, b_1) : \textbf{stmt}} \text{ (TFOR)}$$

However, this assignment rule is extremely hard to apply in a typechecking algorithm. Given the components of the conclusion, Γ and $\mathsf{For}(x, e, b_0, b_1)$, there is no way to guess the several different τ_i. Even if we restrict this to uniformly typed tuples (i.e., replace the second condition with $\Gamma \vdash e : (\sigma, \ldots, \sigma)$ and replace τ_i with σ elsewhere), we still run into dead-ends guessing σ and the length of $(\sigma, \ldots, \sigma)$. There are a countable number of possible σ and lengths, so, if such a $(\sigma, \ldots, \sigma)$ exists we could find it, but this would be a recursively enumerable algorithm; we would find a solution whenever a solution exists, but could possibly run forever if a solution does not exist (i.e., if the expression does not typecheck properly).

It would be possible to implement one of these rules if we restrict the iterated expression to inferable expressions, but this is a very big restriction to make.

7.9. Functions.

7.9.1. *Special Parameters and Arguments.* Our function definition and call assignment rules and implementation only handle positional arguments, but Python allows the following special arguments in function definitions and calls:

Default Parameter Values: In a function definition, parameters can have the form `identifier = expression`. Corresponding arguments can be omitted in calls and the value of the expression (computed once when the definition is read) provided in the definition is substituted.

Variable Positional Arguments: In a function definition, one parameter can have the form `*identifier` to indicate a vararg parameter. If a function definition contains a vararg parameter, then the parameter receives a tuple containing excess positional arguments when the function is called. A function definition must contain zero or one vararg parameter, and it must be the last parameter listed (the parser will throw a syntax error otherwise).

Variable Keyword Arguments: In a function definition, one parameter can have the form `**identifier` to indicate a kwarg parameter. If a function definition contains a kwarg parameter, then the parameter receives a tuple containing excess keyword arguments (i.e., keyword arguments not corresponding to formal parameter names) when the function is called. A function definition must contain zero or one kwarg parameter, the parser will throw an error unless it is the last parameter positionally.

Keyword Arguments: In a function call, arguments can have the form `identifier = expression` to indicate a keyword argument. These are converted to positional arguments by looking at the list of parameters unfilled by positional arguments and finding matching names (a type error is raised if a name matches but was already filled or if a name doesn't match). Slots that are unfilled after this process are filled with default values from the function definition. If any slots are still unfilled, then a type error is raised.

Positional Argument Expansion: In a function call, one argument can have the form `*expression` to indicate positional argument expansion. `expression` must evaluate to an iterable, and then its elements are treated as additional positional arguments. A function call must contain zero or one argument of this form, and it must be the last positional argument (the parser will throw a syntax error otherwise), but may be followed by keyword (named) arguments. The Python expression documentation notes that, because keyword arguments are processed

after positional arguments, `*expression` syntax can appear after keyword arguments but be processed before them. The AST refers to one of these arguments as a `starargs`.

Keyword Argument Expansion: In a function call, one argument can have the form `**expression` to indicate keyword argument expansion. `expression` must evaluate to a mapping, and then its contents are treated as additional keyword arguments. A function call must contain zero or one argument of this form, and it must be the last argument (the parser will throw a syntax error otherwise). The AST refers to one of these arguments as a `kwargs`.

7.9.2. *Decorators.* Our assignment rules and implementation do not handle function decorator expressions. Decorators should be callables which return callables, and are basically function wrappers; the function identifier is bound to the result of applying the decorator to the original function definition.

7.10. Application. The App2 and App3 assignment rules include metavariables in their premises that do not occur in their conclusions, so they are not syntax-directed. These assignment rules are repeated here for reference:

$$\frac{\Gamma \vdash f : \sigma \to \tau \qquad \Gamma \vdash x : \sigma}{\Gamma \vdash f(x) : \tau} \text{ (App2)}$$

$$\frac{n > 1 \qquad \Gamma \vdash f : \sigma \to \tau \qquad \Gamma \vdash \langle x_1, \ldots, x_n \rangle : \sigma}{\Gamma \vdash f(x_1, \ldots, x_n) : \tau} \text{ (App3)}$$

Note that the App1 assignment rule does not contain parameters, so it is fully syntax-directed and does not cause any issues.

This means we can only typecheck application expressions where we can guess the form of σ. Since in practice it is almost always true that a function is defined and assigned to a variable before being called, we have restricted typechecking application expressions to cases where the function being called is an identifier.

If the type environment maps the function identifier to an arrow type, then we are in the clear, and can apply the rule. However, this is the one case where we have to worry about applying the Inst rule; if the type environment instead maps the function identifier

to a type scheme, then we need to apply INST to satisfy the $\Gamma \vdash f : \sigma \to \tau$ condition. But this means the type environment does not give us the form of $\sigma \to \tau$ since the stored type scheme could be instantiated at any type value for each type variable. Thus, we are required to also infer the type of σ by examining the arguments of the function call (and then must verify whether the computed $\sigma \to \tau$ is a valid instantiation of the stored type scheme for the function). For now, this restricts us to only passing inferrable arguments to polymorphic functions; this will probably cause additional headaches if subtyping is introduced to the type system.

Although this covers most use cases, it does exclude us from being able to apply an inline-defined lambda expression (like `(lambda x: x**2)(8)`), or to inline call the result of a function which returns a function (like `function_maker()()`).

7.10.1. *Other Callables.* We only allow applications on functions whose types have been defined by the user, which mean we lack native support for some other objects which Python provides as callable:

(1) Built-in functions
(2) Methods of built-in objects
(3) Class objects
(4) Methods of class instances
(5) Class instances

as outlined in the Python expression reference [Python development team, 2012b]. Note that built-in functions can be used in Pyty, as long as the user declares a type for the function before use.

7.10.2. *Unit Type Overloading.* Because we overload our meaning of **unit** with two distinct Python notions (it is both the type of `None` and the argument type of a parameter-less function—two separate notions in Python), we run into conflict if we want to define a function which accepts a single parameter of **unit** type. Since there is no feasible reason to construct such a function, we have allowed the reasonable restriction that no such functions will be supported.

7.10.3. *Tuple Domain Confusion.* Because of how we denote functions with multiple parameters, a function which takes one tuple argument can cause confusion. Say we want to create a function which takes an input of type $\langle \textbf{int}, \textbf{float} \rangle$. There will be no problem declaring the type of and defining such a function, but once it is defined the function can be used both ways—as a function with two arguments, or as a function with one tuple argument. This is not desired behavior, but is an edge case that will likely not arise very often.

7.11. Multiple Assign. Because we assume that an Assign statement that follows a type declaration is to be interpreted let-polymorphically and STMTS-LETA assumes one statement target, we encounter issues if an Assign statement with multiple targets follows a type declaration, even if the type declaration does not contain free type variables and is clearly not intended to be interpreted polymorphically. Possible solutions include adding a premise to STMTS-LETA checking whether the type declaration's type includes a free type variable. However, this is currently a rare problem (less so if support for declaring types of multiple variables in one type declaration), so further time has not been spent solving it.

CHAPTER 5

Conclusion

As Pierce [2002] puts it, "retrofitting a type system onto a language not designed with typechecking in mind can be tricky; ideally, language design should go hand-in-hand with type system design." While we have found ways to adapt Hindley-Milner style typing to Python, the Cartesian Product Algorithm employed by Salib [2004] and Cannon [2005] appears to have been more easily adaptable to Python, though our use case of finding errors still makes Hindley-Milner a more fitting approach.

The framework we have used has shown potential, but Pyty has a few large features missing that currently hold it back from being practical for widespread use—support for importing modules and for classes. Further development and research will be required to include these features, along with the various smaller language constructs that are trivial theoretically. We have begun to show that Hindley-Milner style typing can be used to support a full typechecking algorithm for Python, and with further development could continue to expand the type-inferrable subset of the language, allowing Pyty type declarations to be optional like in ML.

Bibliography

Ole Agesen. Concrete type inference: Delivering object-oriented applications. Technical report, 1995.

Brett Cannon. Localized type inference of atomic types in Python. Master's thesis, California Polytechnic State University, 2005.

Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM. ISBN 0-89791-065-6. doi: 10.1145/582153.582176. URL `http://doi.acm.org/10.1145/582153.582176`.

Python development team. *The Python Standard Library - Abstract Syntax Trees*. The Python Software Foundation, February 2012a. URL `http://docs.python.org/library/ast.html`.

Python development team. *The Python Language Reference*. The Python Software Foundation, February 2012b. URL `http://docs.python.org/reference/`.

Alex Martelli and Clark C. Evans. PEP 246 - object adaptation, January 2005. URL `http://www.python.org/dev/peps/pep-0246/`.

Neal Norwitz. Pychecker: a Python source code checking tool, 2008. URL `http://pychecker.sourceforge.net/`.

Amit Patel, Antoine Picard, Eugene Jhong, Jeremy Hylton, Matt Smart, and Mike Shields. Google Python style guide, 2012. URL `http://google-styleguide.googlecode.com/svn/trunk/pyguide.html`.

Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10. The MIT Press,

2005.

Johann C. Roholl. PEP8 - Python style guide checker, 2006. URL http://pypi.python.org/pypi/pep8.

Michael Salib. Starkiller: A static type inferencer and compiler for Python. Master's thesis, Massachusetts Institute of Technology, 2004.

Sylvain Thenault. Pylint (analyzes Python source code looking for bugs and signs of low quality), 2006. URL http://www.logilab.org/857.

Guido van Rossum. Adding optional static typing to Python, December 2004. URL http://www.artima.com/weblogs/viewpost.jsp?thread=85551.

Guido van Rossum. Adding optional static typing to Python – part II, January 2005a. URL http://www.artima.com/weblogs/viewpost.jsp?thread=86641.

Guido van Rossum. Adding optional static typing to Python – stop the flames, January 2005b. URL http://www.artima.com/weblogs/viewpost.jsp?thread=87182.

Guido van Rossum. Adding optional static typing to Python – redux, January 2005c. URL http://www.artima.com/weblogs/viewpost.jsp?thread=89161.

Guido van Rossum and Barry Warsaw. Style guide for Python, 2001. URL http://www.python.org/dev/peps/pep-0008/.

Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Christopher S. Serra. The Zephyr abstract syntax description language. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*, DSL'97, pages 17–17, Berkeley, CA, USA, 1997. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267950.1267967.

J. B. Wells. Typability and type checking in system f are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98:111–156, 1998.

Tobias Wrigstad. Introducing STOP, 2010. URL http://wrigstad.com/stop/.