Thesis for the degree of Master of Science

Balancing Cost and Precision of Approximate Type Inference in Python

Levin Fritz

Supervisor: prof. dr. S. D. Swierstra
Daily supervisor: dr. J. Hage
Utrecht, October 2011
ICA-3421252

Department of Information and Computing Sciences
Universiteit Utrecht
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Universiteit Utrecht

Abstract

In dynamically typed programming languages, values have types, but variables and other constructs in the source code do not. A variable can thus refer to any type at runtime, and in general, it is not possible to statically determine these types. Nonetheless, it is often useful to have some information on the types of variables, for example for interactive tools such as editors and IDEs (integrated development environments).

This thesis presents a method for approximate type inference for the dynamically typed language Python. It focuses on balancing precision and cost to find a method that is fast enough to be used in interactive tools while still yielding useful type information. To this end, several different variants of the basic method were developed, and their speed and precision were evaluated using a set of real-world Python codebases.

Contents

1 Introduction **3**
- 1.1 Type inference . . . 3
- 1.2 Approximate Type Inference . . . 4

2 Related Work **5**
- 2.1 Combining static and dynamic typing . . . 5
- 2.2 Type inference for functional languages . . . 6
- 2.3 Type inference for dynamic languages . . . 6
  - 2.3.1 Self . . . 6
  - 2.3.2 PHP . . . 7
  - 2.3.3 Smalltalk . . . 7
  - 2.3.4 Ruby . . . 8
- 2.4 Type inference for Python . . . 8

3 The Python Programming Language **10**
- 3.1 History . . . 10
- 3.2 Implementations . . . 11
- 3.3 Language features . . . 11
  - 3.3.1 Imperative programming . . . 11
  - 3.3.2 Names and scopes . . . 12
  - 3.3.3 Functional programming . . . 12
  - 3.3.4 Object-oriented programming . . . 13
  - 3.3.5 Types . . . 13
  - 3.3.6 Modules . . . 13

4 Data Flow Analysis for Python **15**
- 4.1 Basic data flow analysis . . . 15
  - 4.1.1 Control flow graphs . . . 15
  - 4.1.2 Lattice values . . . 16
  - 4.1.3 Monotone frameworks . . . 16
  - 4.1.4 Worklist algorithm . . . 17
  - 4.1.5 Widening . . . 18
  - 4.1.6 Interprocedural data flow analysis . . . 19
- 4.2 Extended data flow analysis . . . 20
  - 4.2.1 Adding edges dynamically . . . 20
  - 4.2.2 Optional flow-insensitive analysis . . . 21

4.3 Control flow graphs for Python . . . 23
4.3.1 Fully supported constructs . . . 23
4.3.2 Other constructs . . . 28
4.3.3 Function definitions and calls . . . 29
4.3.4 Variables and scope . . . 29

5 Type Inference for Python **31**
5.1 Basic analysis . . . 32
5.1.1 A type lattice for Python . . . 32
5.1.2 Getting information on types . . . 34
5.1.3 Modules and import statements . . . 38
5.2 Analysis variants . . . 39
5.2.1 Parameterized datatypes . . . 39
5.2.2 Context-sensitive analysis . . . 41
5.2.3 Flow-insensitive analysis . . . 42
5.2.4 Manually specified types . . . 43

6 Experimental Evaluation **45**
6.1 Method . . . 45
6.1.1 Measuring precision . . . 45
6.1.2 Measuring speed . . . 46
6.2 Example projects . . . 46
6.3 Results . . . 47
6.3.1 Variants . . . 47
6.3.2 Parameters for widening operator . . . 49
6.3.3 Evaluation . . . 49

7 Conclusions **51**
7.1 Future work . . . 51

A Implementation **53**
A.1 Data Flow Analysis . . . 53
A.2 Type Inference . . . 53
A.3 Command-line Interface . . . 54

B Results of Experiments **55**

Chapter 1

Introduction

Most programming languages entail a notion of types, where a value's type limits which operations can be applied to the value. For example, two values of a number type can be added and a value of function type can be called, but trying to add two functions or call a number would usually be considered an error. A *typing discipline* or *type system* describes the types of a given language and their properties.

Type systems are usually classified into *static* and *dynamic*. With static typing, the absence of type errors can be determined from source code, whereas with dynamic typing, checks for type errors occurs only while a program runs.

When static typing is used, types are assigned not only to values, but also to language constructs such as variables. This is done with *type annotations* provided by the programmer, with *type inference*, or a combination of both.

The main advantages of static typing are that it enables more optimizations, that it allows the compiler to ensure the absence of (certain classes of) type errors, and that it provides useful type information to tools that use the source code, including interactive tools such as text editors and refactoring tools.

The disadvantages are that a type checker must sometimes err on the side of safety and reject a program that would not cause an error at runtime, that some functions cannot be used because the type system cannot express their type, and that programmers are burdened with the additional complexity of the type system's rules.

1.1 Type inference

Type inference means that some or all types for program constructs are inferred automatically by the compiler or another tool. It was originally developed to relieve programmers of the burden of writing and maintaining type annotations. For statically typed functional languages, type inference is now standard (see Section 2.2 below), but there are also imperative programming languages that incorporate type inference, such as the BitC language [24].

Type inference is also used for dynamically typed languages, to improve optimization, error checking or tool support. However, type inference for dy-

namically typed languages is generally more difficult, because these languages were not designed to support the assignment of types to source code constructs. Such methods therefore have to make a compromise between soundness (inferred types are never wrong), precision (inferred types are not overly general) and completeness (inference works for all programs).

1.2 Approximate Type Inference

This thesis presents a method of type inference for the Python programming language, which is a dynamically typed, imperative programming language widely used for website development, scripting, and other tasks. The method is intended to support interactive tools such as programmers' text editors and IDEs (integrated development environments). These tools can use type information for features such as autocompletion (completing an identifier of which a prefix has been typed in), showing a list of methods that can be called on an object and displaying contextual help for an expression in the source code.

To be useful in these situations, the method does not have to guarantee soundness or achieve very good precision, but it is essential that it yields results quickly. The focus is therefore on balancing cost and precision, to find a method of type inference that can be included in an interactive tool without weighing it down with too much additional computation, but that still yields type information precise enough that it can provide useful hints to programmers. This is called "approximate type inference" here to distinguish it from methods of type inference that are guaranteed to either return exact types or fail.

Chapter 2

Related Work

This section discusses some previous work concerned with type inference and the idea of bringing static types in some form to dynamic languages.

2.1 Combining static and dynamic typing

Cartwright and Fagan [6] introduced the concept of *soft typing* in 1991, which is intended to combine the advantages of static and dynamic typing. They note that while static typing is useful because it can detect errors at compile time and produce faster code by removing the need for runtime checks, theses advantages come at the cost of expressiveness, generality and semantic simplicity. According to their definition, a soft type system is a type system that accepts all programs in a dynamically typed language and inserts dynamic checks in places where it cannot statically infer correct types. The programmer can then inspect the places where the checker failed and decide if the code should be changed. The authors present a soft type system for a simple functional language based on ML.

Flanagan [8] introduced *hybrid type checking* in 2006, which is a synthesis of static typing and dynamic contract checking. Dynamic contract checking can support more precise specifications than type checking, for example range checks and aliasing restrictions, but the propositions are not checked until runtime. With hybrid type checking, very precise interface specifications are possible, which are checked at compile time where possible and at runtime where necessary. This is similar to soft typing, but the emphasis is on enabling more precise specifications than are possible with static type systems.

Gradual typing [12, 21–23] is also based on the idea that static and dynamic typing have different strengths and weaknesses. It allows mixing static and dynamic typing within one program: Program elements with type annotations are checked statically, others are checked dynamically. For a fully annotated program, gradual typing is identical to conventional static typing.

2.2 Type inference for functional languages

Type inference is common for functional languages. In languages such as SML [16] and Haskell [14], types are inferred by algorithms based on Damas and Milner's Algorithm W [7]. Algorithm W assigns types to expressions by inferring a set of type equations, and solving them with a unification algorithm.

2.3 Type inference for dynamic languages

Over the past twenty years, a number of methods have been developed that infer types for dynamic languages. This section presents some of these that are particularly relevant to the project proposed here because they have similar goals or use a similar method. For clarity, the methods are organized by programming language.

2.3.1 Self

Ole Agesen introduced the *Cartesian Product Algorithm* (CPA) in his PhD thesis [1] in 1996. The algorithm infers types for programs written in the Self programming language and is intended as a basis for optimization, checking and interactive tools.

CPA is based on an algorithm introduced by Palsberg and Schwartzbach [18] that uses constraint-based analysis to determine types in object-oriented programs. This algorithm assigns a *type variable* to each variable and expression in the source program and derives a set of constraints. Each type variable stands for the set of types that the corresponding variable or expression can have during program execution. A constraint links two type variables; type variables and constraints together form a directed graph, corresponding to a data flow graph of the whole program. The system is solved by propagating types through the system until a fixpoint is reached.

One weakness of this method is that it does not work well for polymorphic methods: the types of the receiver (the object that the method is called on) and the arguments from all call sites are combined, leading to imprecise results for the method's return value. To avoid this, CPA uses the following technique: for each polymorphic method, it creates a copy of the subgraph corresponding to that method for each combination of receiver and argument types, so that each subgraph corresponds to a monomorphic version of the method. These subgraphs are created on-demand: when the analysis encounters a method call, it creates the cartesian product of the type sets of receiver and arguments, and for each combination, creates the corresponding subgraph (if it is not already present), and propagates the results. For example, if a method m is called on object r with one argument a, and the possible types of r and a are $\{\rho_1, \rho_2\}$ and $\{\alpha_1, \alpha_2, \alpha_3\}$, the analysis creates six copies of the subgraph for m, corresponding to the tuples (ρ_1, α_1), (ρ_1, α_2), $\ldots$, (ρ_2, α_3).

2.3.2 PHP

Camphuijsen et al. [4] developed a method for type inference for PHP as part of a tool to detect "suspicious" code, such as code involving a type coercion from array to string (which is legal in PHP, but not really useful). It uses constraint-based analysis: it first collects a set of constraints, then solves them using a worklist algorithm to approximate the maximal fixed point. The analysis is flow-sensitive and context-sensitive, and it supports union types and polymorphic types for arrays. A widening operator is used to ensure termination. The type system also supports polymorphic types for functions; however, these cannot be inferred by the analysis. Instead, the user may specify (polymorphic) type signatures for functions. First-class functions, classes and objects and exceptions are not supported by the analysis.

2.3.3 Smalltalk

Pluquet et al. [19] presented a method for type inference for Smalltalk, which they describe as "extremely fast [...] and reasonably precise". Their goal is an algorithm for type inference that can be used to give direct feedback in an interactive environment. Fast execution is achieved by two means: First, the analysis is local in the sense that, to infer the type of a variable, it uses only information found in the methods of the class it is defined in. Second, it uses a number of heuristics rather than a theoretical model of program execution.

The basic algorithm works as follows:

1. Infer *interface types*, which are determined according to the messages sent to the variable. The algorithm first collects the set of messages sent to the variable, then looks through all classes in the system to find the ones that understand these messages.

2. Infer *assignment types*, which are determined by looking at assignments made to a variable. The type of the right-hand side of an assignment is determined by a simple heuristic, so the algorithm does not have to recursively analyze sub-expressions.

3. Determine relations between variables. Two variables are related if one is assigned to the other. The analysis identifies groups of variables that are related to each other and merges their interface and assignment types.

4. Merge interface and assignment types according to a heuristic.

The analysis is implemented as a bytecode interpreter. This is done for practical reasons: the Smalltalk environments used by the authors make the bytecode produced by the compiler available, and taking advantage of this makes the implementation faster than an analysis that parses the source code and operates on an abstract syntax tree.

The authors validate the method by using it to infer the types of all variables in the source code of three Smalltalk applications. In their experiments, type inference took on average 3.5 ms per variable (the paper does not mention what kind of computer the experiments were done on). To evaluate precision,

they monitored the execution of the programs while running unit tests and interacting with the programs, and recorded the types stored in each variable. The (incomplete) type information retrieved this way was then compared to the inferred types. In about 75 % of the cases, the inferred types were correct or partially correct when using the recorded types as a standard for comparison.

2.3.4 Ruby

Madsen et al. [15] developed a type inference tool for Ruby based on the Cartesian Product Algorithm (see above). An interesting aspect of this tool is that it automatically extracts types for functions and methods in Ruby's standard library from the documentation.

Furr et al. [10] present a system called Diamondback Ruby (DRuby), which consists of three parts: a type language, a method for type inference, and a type annotation language. To be able to describe realistic Ruby programs, the type language includes union types, intersection types and parametric polymorphism. Types are inferred by a method based on constraint-based analysis; however, not all code can be typed with this method. Intersection types, for example, cannot be inferred automatically. For these cases, and standard library classes implemented in C, type annotations are used.

DRuby supports most of the Ruby language, but some more dynamic features such as adding and removing methods in classes are not well supported. There are also methods whose type cannot be expressed in the type system; an example is the `Array.flatten` method in the Ruby standard library, which converts an arbitrary nesting of arrays to a flat array.

PRuby [9] is an extension of DRuby that uses dynamic analysis to deal with highly dynamic constructs such as the `eval` method, which evaluates a string as program text. PRuby first instruments program code so that it records a profile of how dynamic features are used, e.g., which strings are sent to `eval`. The user then runs the program, typically with a test suite, to gather a sufficient profile. PRuby uses this profile to guide a program transformation that removes highly dynamic constructs, before applying DRuby's type inference algorithm.

An et al. [2] present a *constraint-based dynamic type inference* which takes the idea of using dynamic analysis one step further. Their system is based on a constraint-based analysis for type inference where all constraints are collected at runtime. It instruments program code so that each runtime value is wrapped to associate it with a type variable, and the wrapper generates constraints when the variable is used. This instrumented code runs a test suite to collect constraints, which are afterwards solved to find a valid typing, if one exists. For code that cannot be analyzed by the analysis, such as standard-library classes implemented in C, the user can supply type annotations manually.

2.4 Type inference for Python

John Aycock [3] developed a method called "aggressive type inference", which is motivated by the goal of writing a Python-to-Perl compiler. Unlike most

methods for type inference, it is flow-insensitive and does not use union types; this is based on the assumption that most Python code does not make use of the dynamic features of Python's type system. The algorithm consists of two phases: it first parses the source code and extracts information on variables and types, then propagates type information by repeated iteration.

Michael Salib's master thesis [20] describes a type inference method called "Starkiller" which is part of a Python-to-C++ compiler. The algorithm is "based loosely on Agesen's Cartesian Product Algorithm" (see above), but with modifications to support Python's features. It can handle first-class functions and classes and objects, and supports parametric polymorphism as well as data polymorphism (differentiating between different instances of a class with respect to the types of its members). Exceptions and generators are not supported, however, and the analysis is flow-insensitive. The system also includes an "External Type Description Language", which is used to describe types for modules not included in the analysis, such as Python extension modules written in C.

Brett Cannon's master thesis [5] presents a method to improve performance of Python programs which uses type inference and optional type annotations. His implementation modifies the CPython Python implementation, which consists of a bytecode compiler and interpreter, by adding support for type annotations for method arguments, a type inference phase and a set of type-specific bytecodes. The method for type inference is rather limited: it only infers basic built-in types such as *list* and *string*, and it only applies to local variables. It is implemented by abstract interpretation on the bytecode produced by CPython's compiler.

Gorbovitski et al. [11] describe a method for type inference for Python programs which they call "precise type analysis". The main topic of their paper is alias analysis (computing pairs of variables and fields that refer to the same object), which serves as a basis for optimization by incrementalization and specialization, but because constructing a control flow graph in a dynamic language requires information about types, their method also includes an analysis to determine types of variables and expressions. This analysis consists of two steps: type inference and refinement of the control flow graph, which are repeated until either a fixed point or a preset limit is reached.

The algorithm for type inference is based on "abstract interpretation over a domain of precise types". Precise types here means that in addition to primitive and collection types, there are types for known primitive values and ranges, and collections of known contents and lengths, so that the type system can express types such as "bool true", "int between 1 and 10" and "list of length 5". The analysis is flow-sensitive and context-sensitive and supports classes and objects, including dynamic creation and rebinding of fields and methods.

The downside of the method seems to be that it is rather slow. For example, according to experiment results included in the paper, applying it to the BitTorrent program (about 22000 lines of Python code) took about 20 minutes on a computer with a four-core Intel Core 2 CPU and 16 GiB of memory.

Chapter 3

The Python Programming Language

Python [25] is a general-purpose high-level programming language that combines an imperative, object-oriented core with features from functional and scripting languages. It is intended to be expressive and easy to learn and to support writing clean, readable code. Since its creation in 1989, it has become widely used for web development and scripting as well as other tasks such as scientific computing and development of GUI applications. It is maintained as an open-source community project guided by its creator Guido van Rossum.

3.1 History

Van Rossum started development on Python in the late 1980s while working at the *Centrum Wiskunde & Informatica* (CWI, Center for Mathematics and Computer Science) in Amsterdam to create a scripting language for the Amoeba distributed operating system developed at CWI, and to improve upon the ABC language, on which he had worked earlier. His goals for the language included platform independence, so that it could be used both on Amoeba and on the developers' Unix workstations, and support for extensibility through modules written in C.

In 1991, van Rossum first published the source code for his implementation on the Internet. This version was labeled 0.9 and already included support for modules, exceptions and classes with inheritance. In 1994, version 1.0 was released, which included functional programming features such as the *map* and *filter* functions. The next major version, 2.0, was released in 2000. It introduced Unicode strings as a basic datatype, list comprehensions and better garbage collection (earlier versions used a simple reference counting scheme that could not deal with reference cycles).

Version 3.0 was released in 2008. This version breaks compatibility with the Python 2 series in order to correct certain flaws and simplify the language. For example, Python 2.2 introduced a new class model, called "new-style classes", which unified built-in types and user-defined classes. The previous model,

called "old-style classes", was kept to preserve compatibility until it was removed in version 3.0. While developers are slowly switching to Python 3, Python 2 still appears to be the more commonly used version as of 2011.

The version of Python used for the method and implementation described in this thesis is Python 3.2, which was released in February 2011. It should also work for Python 3.0 and 3.1, since there have been no changes to the language that would break compatibility. Supporting Python 2 instead of, or in addition to, Python 3 would mean that the analysis and implementation have to deal with issues such as the distinction between old-style and new-style classes, which would detract from more interesting aspects of the method. Therefore, the method was developed with only Python 3 in mind.

3.2 Implementations

The original Python implementation, called CPython because it is written in C, is implemented as a bytecode compiler and interpreter. While CPython is still the most widely-used implementation, there are now several alternatives, including Jython, which is an implementation for the Java Virtual Machine, IronPython, which is an implementation for the .NET Command Language Runtime and PyPy, which is an experimental implementation of Python in Python.

3.3 Language features

This section gives an overview of the Python programming language, in particular those features that are relevant to the method for type inference presented below.

The following example code, which calculates and prints the factorial function for three numbers, shows Python's syntax and some of its features:

```
def factorial(x):
    if x == 0:
        return 1
    return x * factorial(x - 1)
numbers = [10, 20, 30]
for n in numbers:
    print(n, factorial(n))
```

Unlike most programming languages, Python uses line breaks to separate statements and indentation to delimit blocks of statements, where an increase in indentation indicates the beginning of a block and a decrease indicates the end of a block. Functions are defined using the keyword `def`; a new variable is introduced by assigning to it.

3.3.1 Imperative programming

Python includes the usual constructs for imperative programming: mutable variables, expressions and statements, loops, an *if* statement and function def-

initions. There are two kinds of loops: *while* loops and *for* loops, which iterate over the elements of a sequence. There is no *switch* statement; instead the *if* statement can have multiple conditions, as in the following example:

```
if x > 0:
    print("positive")
elif x < 0:
    print("negative")
else:
    print("zero")
```

Functions may be defined anywhere in the source code. There is no syntax for procedures (functions that don't return a value); instead if there is no explicit return value, the built-in value `None` is returned.

3.3.2 Names and scopes

Variables in Python are introduced not by declarations, but by *name binding operations*, which include assignments, function definitions and class definitions. The visibility of a name is limited to the block (module, function body or class definition) in which it is bound. To resolve a name, the interpreter uses the nearest enclosing scope where the name is bound. The keywords `global` and `nonlocal` can be used to change this behavior for specific variables.

Names are resolved statically, as shown by the following example:

```
x = 1
def f():
    return x
def g(flag):
    if flag:
        x = 2
    return x
```

A call to `f` returns 1, `g(True)` returns 2, but `g(False)` causes an exception: `x` here refers to the local variable `x`, which is never assigned to if `flag` is false.

What's important for the analysis is that one can determine statically which variable a name refers to; in other words, one can determine if two names will refer to the same variable at runtime.

3.3.3 Functional programming

Although Python is not a functional programming language, it does include some features that enable a functional programming style. Functions are first-class (they can be assigned to variables, passed to other functions etc.) and every function is a closure (it captures variables in the enclosing scope). Python also has anonymous functions, although these are limited to a single expression.

Python 2.0 introduced list comprehensions similar to those of Haskell, as in this example:

```
numbers = [0, -1, 1, -2, 2, -3, 3]
squares = [x*x for x in numbers if x > 0]
```

This computes the squares of the positive numbers in the list, resulting in the list `[1, 4, 9]`. Since Python 3.0, there are analogous constructs for building sets and dictionaries.

3.3.4 Object-oriented programming

Python's facility for object-oriented programming, like those of C++, Java and C#, is based on classes. The following example illustrates its main features:

```
class A:
    x = 1
    def m(self, y):
        self.x = y
a = A()
a.m(2)
print(A.x, a.x)
```

This produces "`1 2`" as output.

A class definition contains a block of code; names bound in this block become the class's attributes. Calling a class (the expression `A()` in the example) creates a new instance. The syntax `x.y` is used to access the attribute `y` in the class or instance that `x` refers to.

A method is a class attribute which is a function. When a method is called, its first argument is set to the object on which it is called. By convention, this argument is named `self`.

The class system is very dynamic: at runtime, new classes can be created, attributes (including methods) can be added to or removed from classes and attributes can be added to and removed from instances. It also supports inheritance, including multiple inheritance, and operator overloading with "special method names" such as `__add__` for overloading the `+` operator.

3.3.5 Types

Every value in Python has a type, which can be queried with the built-in function `type`. There are built-in types, such as `list` and `int`, and classes.

Since Python is dynamically typed, variables do not have types, nor do attributes of classes and objects. Instances of the built-in collection types (tuple, list, set and dictionary) can also hold objects of any type.

3.3.6 Modules

There is no syntax for declaring modules in Python; instead, each source file corresponds to a module. Thus, the declarations in file *foo.py* are contained in module *foo* at runtime. Names bound in one module can be made available in another module by an `import` statement. For example, assuming module `foo`

declares variables `x` and `y`, these can be made available as `foo.x` and `foo.y` by the statement

```
import foo
```

The following statement imports the names directly into the local namespace:

```
from foo import x, y
```

Chapter 4

Data Flow Analysis for Python

The method for type inference presented in this thesis is a data flow analysis, which is a type of program analysis based on a control flow graph for the program under analysis. This chapter describes the ideas behind data flow analysis and the techniques used to formalize it, presents two extensions to data flow analysis and finally discusses the creation of control flow graphs for Python programs.

4.1 Basic data flow analysis

Data flow analysis is one of the most common types of program analysis. This section presents an overview of the technique; for a more detailed discussion, see [17], Chapter 2.

4.1.1 Control flow graphs

The first step in a data flow analysis is the construction of a control flow graph for the program under analysis. The control flow graph is a directed graph whose nodes are *program points*. Program points correspond roughly to statements in the source code, but there also need to be program points for all points in the program where the flow of control can diverge or converge. For example, for a list comprehension in a Python program, there is a program point for each loop and each condition implied by the list comprehension. The program points are the control flow graph's nodes; edges represent possible flow of control during program execution.

Figure 4.1 shows a fragment of Python code and the corresponding control flow graph. In the source code, program points are marked with square brackets and each is labeled with a unique number. In the graph, the initial node (where program execution begins) is marked with a curved arrow and the final node is marked with a box around the number.

Figure 4.1: Control flow graph for a fragment of Python code.

4.1.2 Lattice values

A data flow analysis computes values for each program point. To formalize this idea, it is convenient to restrict these values to be elements of a *lattice*, which is defined as a partially ordered set where any two elements have a unique supremum and a unique infimum. In other words, it is a set L with a relation $\sqsubseteq$ which is reflexive, asymmetric and transitive and is defined so that any two elements $x, y \in L$ have a unique least upper bound $x \sqcup y$ and a unique lower bound $x \sqcap y$. The operators $\sqcup$ and $\sqcap$ are called *join* and *meet*. If the lattice has a greatest element, it is called *top* ($\top$); if it has a least element, it is called *bottom* ($\bot$). If $\top$ and $\bot$ exist, $\bot \sqsubseteq x \sqsubseteq \top$ for all $x \in L$.

The data flow analysis computes two values for each program point l: the *context value* $A_\circ(l)$ and the *effect value* $A_\bullet(l)$. The effect value $A_\bullet(l)$ is always computed from the context value $A_\circ(l)$ and effect values are propagated to context values along the graph's edges. There are two ways to do this: in a *forward analysis*, values are propagated in the direction of the edges; in a *backward analysis*, they are propagated in the opposite direction.

Since the analysis presented in this thesis is a forward analysis, the discussion from here on focuses on forward analyses. For the most part, the techniques are the same for backward analyses, except that the direction of the edges is reversed and initial and final nodes are swapped.

In forward analyses, the context value $A_\circ(l)$ is computed as the join of the effect values of all direct predecessors of l (nodes from which an edge leads to l). A specific analysis must provide an *initial value*, which is assigned to the initial node's context value, and specify, for each program point, how to compute the effect value from the context value.

4.1.3 Monotone frameworks

Monotone frameworks are a way to formalize these ideas; they provide a framework that abstracts the commonalities and parameterizes the differences of data flow analyses.

The program under analysis is called S; its program points are identified by the set $\mathit{label}(S)$. *Extremal labels* are those that identify program points where the program begins or ends execution: $\mathit{init}(S) \in \mathit{labels}(S)$ is the initial label and $\mathit{final}(S) \subseteq \mathit{labels}(S)$ are the final labels. The program flow $\mathit{flow}(S) \subseteq$

$labels(S) \times labels(S)$ contains possible transitions from one program point to the next; the control flow graph is given by $(labels(S), flow(S))$.

The analysis results are described by a lattice L, called the *property space* of the analysis. For each program point l, there is a *transfer function* f_l:

$$A_\bullet(l) = f_l(A_\circ(l))$$

The *extremal value* $\iota \in L$ is used as an initial value for the property space. A monotone framework can then be described by the seven-tuple

$$(L, \mathcal{F}, F, E, \iota, \lambda l.f_l)$$

where

- L is the lattice of analysis results,
- $\mathcal{F}$ is the monotone function space containing all transfer functions,
- F is the program flow,
- E is the set of extremal labels,
- ι is the extremal value, and
- $\lambda l.f_l$ is the mapping from labels to transfer functions from $\mathcal{F}$.

4.1.4 Worklist algorithm

Computing the result of an analysis expressed as a monotone framework – "solving" the framework – is done by fixpoint iteration. First, a table with context values is initialized to ι for the extremal labels and $\bot$ for the others. Then, the transfer functions are applied and their results are propagated along edges. This step is repeated until it does not result in any further changes and a fixpoint is reached. The final results are the context values computed in this manner and the effect values obtained by applying the transfer functions once more.

A variation of this method is to also store effect values; this is more efficient if the transfer functions are expensive to compute.

The simplest way to get to the fixpoint, called *chaotic iteration*, is to apply all transfer functions in every step, until none of them yield a new result. The *worklist algorithm* is a more sophisticated scheme: to avoid applying a transfer function to the same context value twice, the algorithm maintains a worklist containing all nodes whose context value changed since the last time their transfer function was computed. The worklist initially contains all nodes; the algorithm proceeds by taking the first node from the list, applying the transfer function for that node to its context value, updating effect values by propagating the result along outgoing edges, and adding those nodes whose effect value was changed to the worklist.

Algorithm 4.2 is the worklist algorithm in more formal notation. This is the variant of the algorithm that maintains both context and effect values; in consequence, the worklist (W) contains both edges and nodes and is initialized

Initialization:

$A_\circ[l] \leftarrow \begin{cases} \iota & \text{for } l \in E \\ \bot & \text{otherwise} \end{cases}$

$A_\bullet[l] \leftarrow \bot$

$W \leftarrow N$

Iteration:

while W not empty **do**
  $i \leftarrow \text{head}(W)$
  $W \leftarrow \text{tail}(W)$
  if $i = l \in N$ **then**
    if $f_l(A_\circ[l]) \not\sqsubseteq A_\bullet[l]$ **then**
      $A_\bullet[l] \leftarrow f_l(A_\circ[l])$
      for all l' **with** $(l, l') \in F$ **do**
        $W \leftarrow (l, l') : W$
  else if $i = (l, l')$ **then**
    if $A_\bullet[l] \not\sqsubseteq A_\circ[l']$ **then**
      $A_\circ[l'] \leftarrow A_\circ[l'] \sqcup A_\bullet[l]$
      $W \leftarrow l' : W$

Algorithm 4.2: Basic worklist algorithm.

with the set of all nodes (N) instead of the set of all edges. In the iteration, the test determines if the element taken from the worklist is a node ($i = l \in N$) or an edge ($i = (l, l')$).

4.1.5 Widening

The worklist algorithm is guaranteed to terminate if $(L, \sqsubseteq)$ satisfies the *ascending chain condition*: for any sequence $l_1, l_2, \ldots$ of elements of the lattice with $l_n \sqsubseteq l_{n+1}$, there is a k such that for $n \geqslant k$, $l_n = l_{n+1}$ (the sequence eventually stabilizes). If this is not the case, a *widening operator* can be used to ensure termination. The widening operator ∇ replaces the join operator $\sqcup$. Like $\sqcup$, it gives an upper bound of two elements of a lattice, but not necessarily the least upper bound, and it satisfies the following condition: given any sequence $l_1, l_2, \ldots$ of elements of the lattice, the sequence

$$l_1, l_1 \nabla l_2, (l_1 \nabla l_2) \nabla l_3, \ldots$$

eventually stabilizes.

When the worklist algorithm uses a widening operator to compute context values from effect values, it is guaranteed to terminate even if L does not satisfy the ascending chain condition, but its output is not necessarily the least fixed point anymore. The widening operator therefore needs to be carefully chosen so that the result of the algorithm with widening is similar to the least fixed point in the sense that it still represents a useful analysis result.

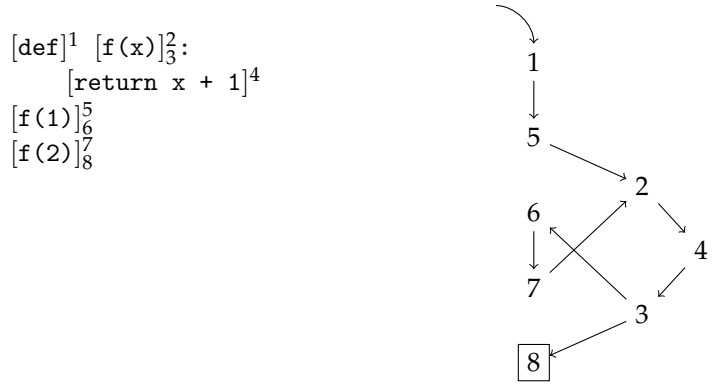

Figure 4.3: Control flow graph for function definition and function calls.

4.1.6 Interprocedural data flow analysis

If data flow analysis is to be used with any realistic programming language, it has to deal with procedures (functions). In the control flow graph, these can be represented as follows: for each procedure, there are two extra program points, the entry and exit nodes, labeled l_n and l_x. A call site (a program construct which indicates a procedure call) is also represented by two program points, the call and return nodes l_c and l_r. The call is then indicated by edges (l_c, l_n) and (l_r, l_x).

Figure 4.3 shows a fragment of Python code with one function which is called in two places.

Context-sensitive analysis

One issue with this type of interprocedural analysis is that results from different calls to the same function are combined. The example in Figure 4.4 illustrates this problem: analysis results for the two calls to function *id* are necessarily combined, because they are propagated through the context and effect values for nodes 2, 4 and 3. An analysis that determines (or approximates) the values of variables x and y would thus get the same result for both.

The Cartesian Product Algorithm [1] solves this issue by creating multiple copies of the subgraph for each function: for each combination of argument types, the nodes and edges for the function are replicated, so that the results of different calls are combined only if the argument types are the same.

Context-sensitive analysis is a more general solution. The idea here is to process analysis results separately depending on the *context*. To do so, the lattice L is replaced with a mapping from context Δ to L, so that context and effect values are of type $\Delta \to L$. When computing an effect value, the transfer function is applied to each lattice value separately.

This method is very flexible, since the context Δ can be chosen to represent any property that the analysis needs to differentiate by. To distinguish different function calls, *call strings* are used as context. A call string is a list of nodes from which function calls were made. The transfer function for a call node adds the

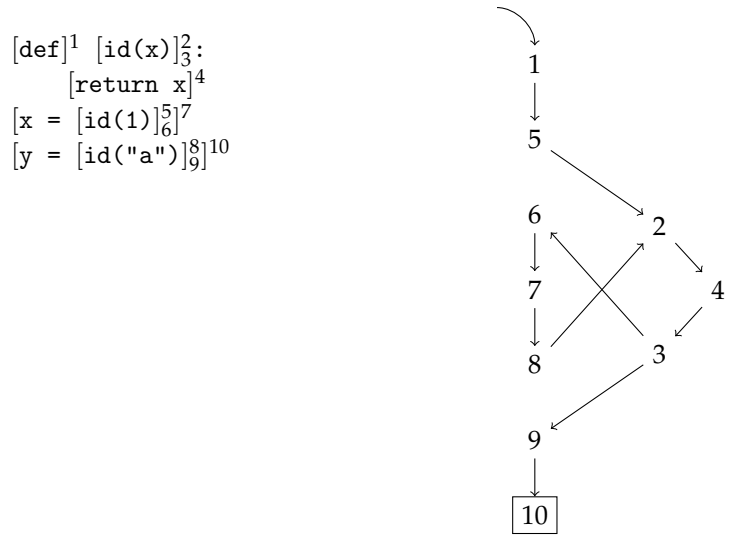

Figure 4.4: Control flow graph for the identity function.

node's label (l_c) to the context; the transfer function for a return node selects matching call strings and removes the last element of each. The call string can be seen as an abstraction of the call stack.

Returning to the example of Figure 4.4, the initial context, assuming the fragment shown is analyzed on its own, would be the empty call string []. The transfer function for node 5 would change it to [5], the one for node 6 would change it back to [] and the one for node 7 would change it to [7]. For nodes 2, 4 and 3, the context and effect values would thus contain lattice values for context [5] and context [7], enabling the analysis to keep results for the different function calls separate.

The length of the call strings has to be limited to make sure the analysis does not go into an infinite loop creating ever-longer call strings in the case of recursive function calls. The maximum length parameter can also be used to trade off precision for speed.

4.2 Extended data flow analysis

In order to handle Python code properly and to support different variants of the analysis, the basic data flow analysis described in the previous section was extended in two ways for the thesis project.

4.2.1 Adding edges dynamically

Because Python has first-class functions and uses late binding for methods, it is often not trivial to infer which function or method is called at a call site in the source code. The control flow graph constructed for the analysis is therefore incomplete at first.

Since most Python programs do use classes and methods, a realistic method for type inference has to solve this issue. The method proposed in this thesis assigns a unique identifier to each function in the program under analysis and these identifiers are included in the types inferred for functions, so which function a call can refer to does become apparent during the analysis.

To be able to use this information, the monotone framework and worklist algorithm are extended so that edges for function calls can be added to the control flow graph during execution of the worklist algorithm.

Extended monotone framework

Because the analysis has to be aware of the functions defined in the program, the monotone framework contains one more element: the function table Λ, which maps function identifiers to labels for entry and exit nodes. More formally, $\Lambda[f] = (l_n, l_x)$, where f is a function identifier and l_n and l_x are the labels of the function's entry and exit program points. The complete monotone framework then becomes the eight-tuple $(L, \mathcal{F}, F, E, \iota, \lambda l.f_l, \Lambda)$.

Extended worklist algorithm

The transfer function for a call (l_c) program point must indicate which functions may be called at that call site. Therefore, there are two kinds of transfer functions: *simple transfer functions*, which are as described above, and *call transfer functions*, which are used for function call nodes. In addition to the computed effect value, a call transfer function returns a set of function identifiers indicating which functions may be called at that point. The worklist algorithm looks up these identifier in the function table and, for each of them, adds two edges: (l_c, l_n) and (l_x, l_r).

The call transfer function also returns a flag to indicate cases where the analysis cannot identify the function called (e.g., because it is not part of the source code analyzed). If this flag is set, the worklist algorithm adds an edge (l_c, l_r), connecting the call and return nodes directly.

Each edge added to the graph is also added to the worklist, since, obviously, at that point the most recent effect value has not been propagated across the edge.

See Algorithm 4.5 for pseudocode of the extended algorithm. The transfer function f_l returns a three-tuple here, where t is the set of functions called at that program point and d is the flag indicating if there should be an edge from call to return node. For simple transfer functions, t is $\emptyset$ and d is *false*.

4.2.2 Optional flow-insensitive analysis

One of the basic properties of static analyses is whether they are flow-sensitive or flow-insensitive: flow-sensitive analyses assign different values to different program points; flow-insensitive ones only compute one global value. The method described in this thesis is flow-sensitive, but it includes variants that use flow-insensitive analysis for some of the types inferred (see Section 5.2.3).

Initialization:

$A_\circ[l] \leftarrow \begin{cases} \iota & \text{for } l \in E \\ \bot & \text{otherwise} \end{cases}$

$A_\bullet[l] \leftarrow \bot$

$W \leftarrow N$

Iteration:

while W not empty **do**
  $i \leftarrow \text{head}(W)$
  $W \leftarrow \text{tail}(W)$
  if $i = l \in N$ **then**
    $(t, d, e') \leftarrow f_l(A_\circ[l])$
    if $e' \not\sqsubseteq A_\bullet[l]$ **then**
      $A_\bullet[l] \leftarrow e'$
      for all l' **with** $(l, l') \in F$ **do**
        $W \leftarrow (l, l') : W$
    $F' \leftarrow \bigcup \{\{(l_c, l_n), (l_x, l_r)\} \mid (l_n, l_x) \leftarrow \Lambda[f], f \leftarrow t\}$
    if d **then**
      $F' \leftarrow F' \cup (l_c, l_r)$
    if $F' \not\subseteq F$ **then**
      for all $e \in F' \setminus F$ **do**
        $W \leftarrow e : W$
      $F \leftarrow F' \cup F$
  else if $i = (l, l')$ **then**
    if $A_\bullet[l] \not\sqsubseteq A_\circ[l']$ **then**
      $A_\circ[l'] \leftarrow A_\circ[l'] \sqcup A_\bullet[l]$
      $W \leftarrow l' : W$

Algorithm 4.5: Worklist algorithm extended to add edges for function calls.

To also support flow-insensitive results, the analysis maintains a global lattice value in addition to context and effect values. The monotone framework is extended to include an initial global value ι_g, so that it can be written as a nine-tuple $(L, \mathcal{F}, F, E, \iota, \iota_g, \lambda l.f_l, \Lambda)$.

The transfer functions are extended in three ways: they take the current global value as a parameter, they return the new global value, and they return a flag indicating if they used the global value or not. Whenever the global value changes, all transfer functions that use it have to be recomputed. The worklist algorithm therefore maintains a set of nodes whose transfer functions use the global value. This set is initially empty; when a transfer function indicates that it used the global value, its node is added to the set. Thus, if a transfer function returns a new global value, the global value is updated and all nodes in the set are added to the worklist.

Algorithm 4.6 is the worklist algorithm with support for adding edges dynamically and optional flow-insensitive analysis. Here, g is the global lattice value and G is the set of program points whose transfer functions use g. In the tuple returned by the transfer functions, g' is the new global lattice value and u indicates if the global lattice value was used.

4.3 Control flow graphs for Python

The first step in a data flow analysis is the creation of a control flow graph for the code under analysis. Although the language constructs of Python are, on the whole, similar to those of other imperative, object-oriented programming languages, there are differences in many details. This section describes how the analysis handles various features of the Python language, and also which features are not (completely) supported by the analysis.

4.3.1 Fully supported constructs

This section shows how the basic language constructs such as conditional statements and loops are handled.

Conditionals

The *if-else* construct, which allows a choice between two code paths depending on the value of an expression, is one of the basic features of most imperative programming languages. Figure 4.7 shows Python code with a simple *if* statement and the corresponding control flow graph. In the places where a sequence of statements is contained in the statement, the examples have a single assignment, which is represented as a single node in the control flow graph. If the assignment were replaced by more (complex) statements, the corresponding node in the graph would be replaced by the subgraph for those statements, but the rest of the graph would stay the same.

A conditional statement with an *elif* clause is show in Figure 4.8. As can be seen, the *elif* clause adds another node for the condition and the subgraph for the corresponding sequence of statements.

Initialization:

$A_\circ[l] \leftarrow \begin{cases} \iota & \text{for } l \in E \\ \bot & \text{otherwise} \end{cases}$

$A_\bullet[l] \leftarrow \bot$

$g \leftarrow \iota_g$

$W \leftarrow N$

$G \leftarrow \emptyset$

Iteration:

while W not empty **do**
  $i \leftarrow \text{head}(W)$
  $W \leftarrow \text{tail}(W)$
  if $i = l \in N$ **then**
    $(t, d, e', g', u) \leftarrow f_l(A_\circ[l])$
    if $g' \not\sqsubseteq g$ **then**
      $g \leftarrow g'$
      for all $l \in G$ **do**
        $W \leftarrow l : W$
    if u **then**
      $G \leftarrow \{l\} \cup G$
    if $e' \not\sqsubseteq A_\bullet[l]$ **then**
      $A_\bullet[l] \leftarrow e'$
      for all l' **with** $(l, l') \in F$ **do**
        $W \leftarrow (l, l') : W$
    $F' \leftarrow \bigcup \{\{(l_c, l_n), (l_x, l_r)\} \mid (l_n, l_x) \leftarrow \Lambda[f], f \leftarrow t\}$
    if d **then**
      $F' \leftarrow F' \cup (l_c, l_r)$
    if $F' \not\sqsubseteq F$ **then**
      for all $e \in F' \setminus F$ **do**
        $W \leftarrow e : W$
      $F \leftarrow F' \cup F$
  else if $i = (l, l')$ **then**
    if $A_\bullet[l] \not\sqsubseteq A_\circ[l']$ **then**
      $A_\circ[l'] \leftarrow A_\circ[l'] \sqcup A_\bullet[l]$
      $W \leftarrow l' : W$

Algorithm 4.6: Worklist algorithm extended for flow-insensitive analysis.

```
if [x]1:
    [y = "a"]2
else:
    [y = "b"]3
```

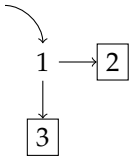

Figure 4.7: Simple *if* statement.

```
if [x]1:
    [y = "a"]2
elif [y]3:
    [y = "b"]4
else:
    [y = "c"]5
```

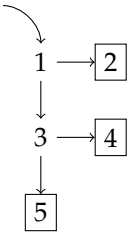

Figure 4.8: *If* statement with *elif* clause.

```
["a"]2 if [x]1 else ["b"]3
```

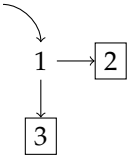

Figure 4.9: Conditional expression.

In addition to the *if* statement, Python also has a conditional expression. The control flow graph, shown in Figure 4.9, is basically the same as that for the *if* statement in Figure 4.7.

Control flow within expressions

The control flow graph in Figure 4.9 raises a question: program point 1 represents the condition `x`, but what do program points 2 and 3 stand for? More generally, how should program points that correspond to expressions, rather than statements, be dealt with? The solution used here is to introduce a generated variable whenever an expression is represented by a subgraph of the control flow graph. Each generated variable is assigned a unique number; generated variables are identified by the Greek letter ι here as well as in the analysis output.

The example in Figure 4.10 shows an assignment with a conditional expression. The program point for the assignment statement in the source code becomes the final node; the graph generated for the conditional expression precedes it in the program's control flow graph. Nodes 2 and 3 set the value of the generated variable ι_1, which is then used in node 4.

Other expressions that require multiple program points, such as list comprehensions and function calls, are handled in the same way. The subgraph for the expression is always put before the node in which it is used, and its result assigned to a unique generated variable.

Loops

There are two types of loops in Python: *while* loops, which loop as long as a condition is true, and *for* loops, which iterate through the elements of a sequence.

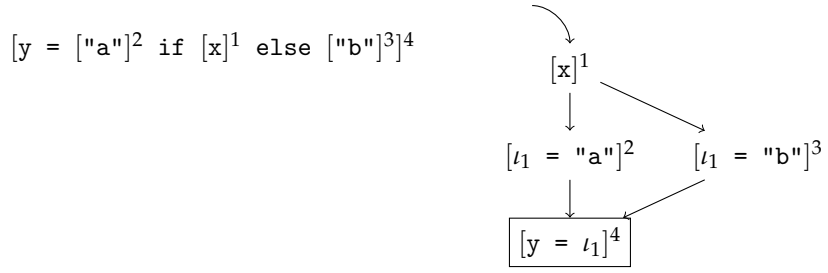

Figure 4.10: Assignment with conditional expression.

Figure 4.11: Simple *while* loop.

As in C or Java, the *break* statement can be used to break out of a loop and the *continue* statement jumps to the next loop iteration.

Figure 4.11 and Figure 4.12 show control flow graphs for simple *while* and *for* loops. In the *for* loop, the expression that the loop iterates over (`list` in the example) is evaluated before the loop is entered; in Figure 4.12, program point 1 represents this evaluation.

A rather unique feature of Python is that loops can have an *else* clause which, if present, is executed when the condition in a *while* loop is false or the iterator in a *for* loop is exhausted. Figure 4.13 shows a *while* loop with an *else* clause; Figure 4.14 shows a *while* loop with an *else* clause containing *continue* and *break* statements. Corresponding control flow graphs for *for* loops would be identical except for the additional node for the evaluation of the iterator.

List, set and dictionary comprehensions

Figure 4.15 shows an example of a list comprehension, a set comprehension and a dictionary comprehension. The control flow graph has the same structure for each of these, so only one graph is shown.

The graph contains one node (labeled 5) for the assignment, two nodes (2 and 3) for the *for* clause, one node (4) for the *if* clause and one node (1) for

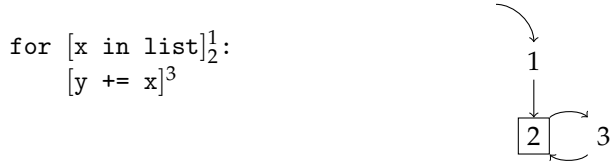

Figure 4.12: Simple *for* loop.

```
while [x]1:
    [y = "a"]2
else:
    [y = "b"]3
```

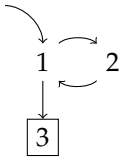

Figure 4.13: *While* loop with *else* clause.

```
while [x]1:
    if [y]2:
        [continue]3
    elif [z]4:
        [break]5
    [w = "a"]6
else:
    [w = "b"]7
```

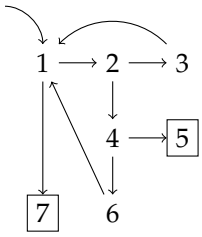

Figure 4.14: *While* loop with *continue* and *break* statements.

the accumulation of values. Note that the part of the graph for the *for* clause corresponds to that for a *for* loop in Figure 4.12 (with nodes 4 and 1 as the loop body) while the part for the *if* clause corresponds to the graph for an *if* statement without an *elif* or *else* clause.

Comprehensions can contain any number of *for* and *if* clauses; for each additional clause, there is a corresponding sub-graph nested within that for the previous clause, with the node for the accumulation of values innermost.

```
[squares_list = [[x*x]1 for [x in [0,1,2]]2_3 if [x > 0]4]]5
[squares_set  = {[x*x]1 for [x in [0,1,2]]2_3 if [x > 0]4}]5
[squares_dict = {[x : x*x]1 for [x in [0,1,2]]2_3 if [x > 0]4}]5
```

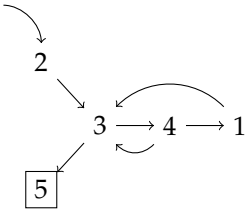

Figure 4.15: List/set/dictionary comprehension.

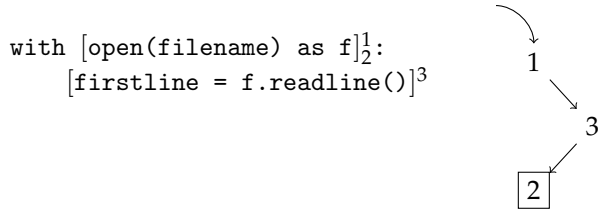

Figure 4.16: *With* statement.

The *with* statement

Python's *with* statement encapsulates the *try-finally* pattern used to ensure a resource acquired in a block of code is released before leaving it. It takes an object called a *context manager*, which provides methods called `__enter__` and `__exit__` to handle entry into and exit from an environment. Optionally, the `__enter__` method's return value can be bound to a variable.

Figure 4.16 shows a typical use of the *with* statement: opening and closing a file. The control flow graph for a *with* statement with multiple context managers is the same as that for multiple nested *with* statements.

4.3.2 Other constructs

There are two constructs in Python that the analysis does not fully support: exceptions and generators.

Exceptions

The purpose of exceptions is to break out of the normal flow of control, which, not surprisingly, is difficult to take into account in control flow graphs. To represent exception handling in the control flow graph, the analysis would have to identify the program points that could result in an exception (for instance, each expression containing a division by a non-constant divisor could result in a *ZeroDivisionError* exception), identify the *catch* clauses and which exceptions each handles, and add edges representing the flow of control from the former to the latter.

However, this would add significant complexity to the analysis, so the implementation handles exceptions in a much simpler way: it assumes that since exceptions only occur in exceptional circumstances, they can be ignored by the type inference method without changing the results too much, and therefore *catch* clauses in the source code are simply not part of the control flow graph. Figure 4.17 shows the graph for a simple *try* statement with one *catch* clause and a *finally* clause.

Generators

Python's generators are a convenient way to create iterators, which can be used either through the *for* statement or by calling their methods directly. A gener-

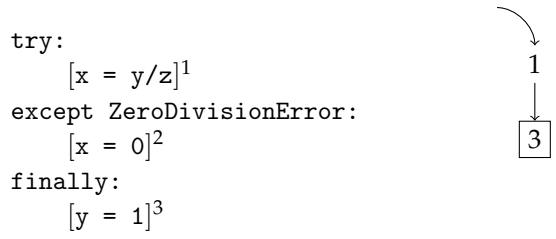

Figure 4.17: *Try* statement.

ator is written like a regular function, but it contains one or more *yield* statements, which are used to pass data to the caller. When the next element is requested from the generator, is proceeds until the next *yield* statement or until the end of the function.

To handle generators, the analysis could use a method similar to that employed for regular functions (see Section 4.2.1): a *generator table* would contain all generators defined in the program and the program points for *yield* statements contained in each; the analysis would then dynamically add edges from and to the nodes for *yield* program points.

However, because this would require yet another extension to the worklist algorithm and generators are one of the more obscure features of Python, this was not done in the thesis project.

4.3.3 Function definitions and calls

The representation of functions and function calls follows the inter-procedural analysis described above in Section 4.1.6. Figure 4.18 shows a simple example. For the function definition, program points for entry (l_n) and exit (l_x) are generated, with edges from l_n to the first program point inside the function and from the program points for *return* statements to l_x. For each function call, program points for call (l_c) and return (l_r) are generated.

There is also a program point for the function definition itself (1 in the example). During the analysis, the transfer function for this node adds an entry binding the function name to a type containing the function id, which is then propagated through the graph and used by the transfer function for l_c to tell the extended worklist algorithm which function is called at the call site. The algorithm then adds edges (l_c, l_n) and (l_x, l_r), shown as dashed lines in the graph.

4.3.4 Variables and scope

To build the control flow graph, one of the things the analysis needs to determine is which variable an identifier refers to; in other words, it needs to find out the scopes of variables. To make this information available to the next stage of the analysis, the analysis creates program points which indicate where

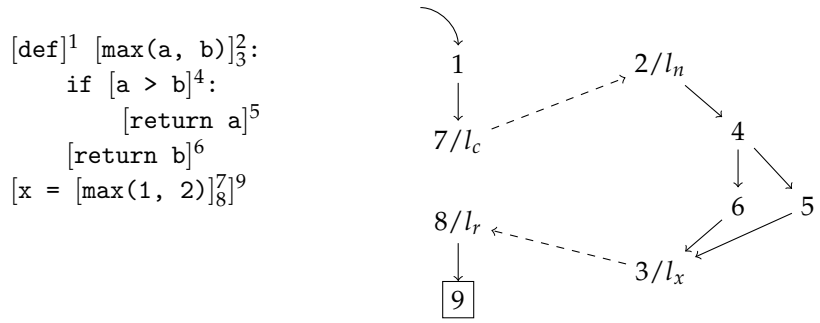

Figure 4.18: Function definition and call.

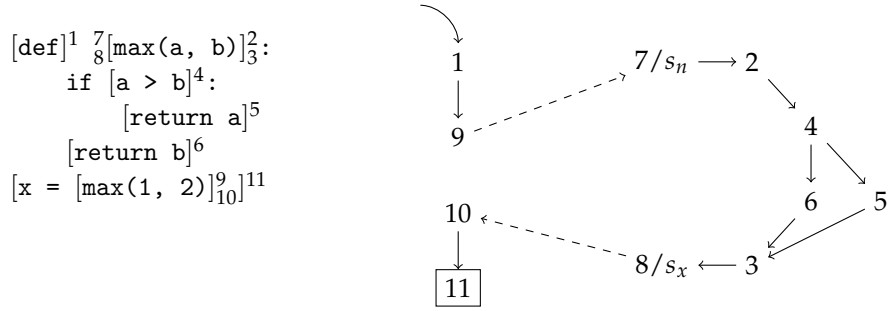

Figure 4.19: Function definition and call, with s_n and s_x nodes.

variables come into and go out of scope. For each program construct that introduces a new scope, this adds two program points: s_n for entry into the scope and s_x for exit from the scope. Each of these contains a list of the variables that come into/go out of scope.

The control flow graphs shown previously omitted these nodes to reduce visual clutter. Figure 4.19 is identical to Figure 4.18 except that it includes s_n and s_x nodes. In this example, they would indicate the scope of the parameter names a and b, since no new variables are introduced in the function.

Chapter 5

Type Inference for Python

The idea behind the method described here is to infer types for variables in Python source code. However, this formulation is not quite satisfactory: since Python is dynamically typed, there is really no notion of "types of variables" in the language. A more precise statement of the goal of the analysis is: for each variable in a Python program, try to infer the types of the values it may be bound to when the program is executed. Variable here includes parameters of functions and methods (the Python language reference [25] refers to this as "names").

In statically typed languages such as Java or Haskell, all types in a program can usually be determined statically. By contrast, the analysis described here is approximate, for several reasons:

- Fundamentally, it is impossible for static analysis of a Turing-complete language to determine the values computed or the flow of control in all cases. Since the types inferred may depend on run-time values or flow of control, it is not possible to determine all types precisely by static analysis.
- There are some very dynamic features in Python which make static analysis difficult. An example is the built-in *eval* function, which takes a string as argument, interprets it as a Python expression and returns the value it evaluates to. For instance, `eval("1 / 2")` returns the floating-point number 0.5. Since the argument string may be computed at runtime or, for example, be read from a file, it can be very difficult to statically determine the return type of a call to *eval*.
- The analysis as presented here is also limited for pragmatic reasons in some cases. For example, as described above in Section 4.3.2, exceptions and generators are not properly supported.

The method is split into a basic analysis and a number of analysis variants, because one of the goals of the thesis is to compare different variants and determine their influence on precision and speed.

$u \in$ UTy	union types	$u ::= \{v\} \mid \top$
$v \in$ ValTy	value types	$v ::= b \mid f \mid c \mid i$
$b \in$ BuiltinTy	built-in type	$b ::= int \mid bool \mid list \mid \ldots$
$f \in$ FunTy	function types	$f ::= f_l$
$c \in$ ClsTy	class types	$c ::= class\langle l, [c], \{n \mapsto u\}\rangle$
$i \in$ InstTy	instance types	$i ::= inst\langle c, \{n \mapsto u\}\rangle$
$l \in \mathbf{N}$	label	
$n \in$ String	name	

Figure 5.1: Basic type lattice for Python.

In the implementation, all variants are implemented in the same source base; a *Configuration* object is passed around which specifies the analysis variants and other options used. The user can specify these with command-line parameters.

The following sections describe first the basic analysis, then each of the analysis variants.

5.1 Basic analysis

The basic type inference method is a data flow analysis expressed as a monotone framework and solved by the worklist algorithm (see Chapter 4). It is flow-sensitive, context-insensitive and path-insensitive. (A path-sensitive analysis computes different results depending on predicates at nodes where the flow of control diverges. For example, a path-sensitive analysis could infer that in the body of the statement *if x is None: ...*, variable *x* has value *None*.)

The following sections describe the lattice used for the analysis, the way different statements provide information on types, and the method used to handle modules and import statements.

5.1.1 A type lattice for Python

Figure 5.1 defines a type lattice for Python. In the notation used on the right, $\{\nu\}$ stands for a set of zero or more elements of the form ν, and $[\nu]$ stands for an ordered sequence of zero or more elements of the form ν.

The type assigned to a variable is called a *union type*: it is either the set of types of the values that the variable may be bound to at runtime, or $\top$, which means the analysis cannot infer anything about the type.

Value types model types of values at runtime. The analysis distinguishes five kinds of value types. Built-in types are built into Python and rules to deal with them are built into the analysis. A function type refers to a function in the source code; these are assigned unique labels to avoid name clashes.

The types for classes defined in the code under analysis and instances of these are more interesting. Classes and objects are very dynamic in Python: at runtime, new classes can be created and attributes can be added to and removed from classes and objects. Classes may also have multiple superclasses.

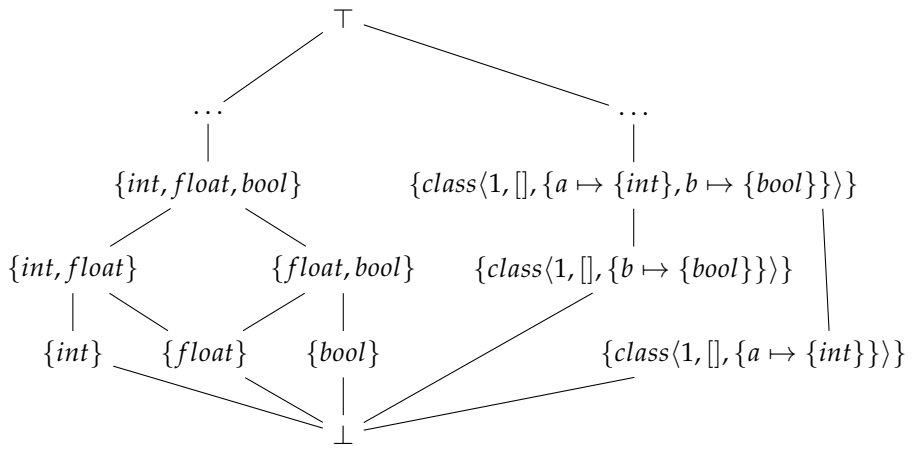

Figure 5.2: Part of the basic type lattice.

This is captured by the definitions in Figure 5.1: a class type contains a list of superclasses and a mapping from names to types for class attributes; an instance type contains the instance's class and a mapping for instance attributes. Like functions, classes are also assigned unique labels.

Join operator

Turning the set UTy into a lattice means defining a join operator $\sqcup$ and a bottom element $\bot$. (This actually defines only a join-semilattice, because there is no meet operator $\sqcap$, but this is sufficient for the data flow analysis. For brevity, UTy will be called a "lattice" here anyways.)

The bottom element is defined as $\bot = \emptyset$. The join of two union types u_1 and u_2 is $\top$ if $u_1 = \top$ or $u_2 = \top$. If neither of them is $\top$, $u_1 \sqcup u_2$ is basically the union of the sets. However, if the set union $u_1 \cup u_2$ contains multiple class types with the same class identifier, or multiple instance types whose class types have the same identifier, these are merged. When two class types are merged, the resulting class type contains the superclasses and attributes of both class types. Similarly, when two instance types are merged, the resulting instance type contains the attributes of both and their class types are merged as well.

Figure 5.2 shows some of the elements of the lattice in the form of a diagram. In the diagram, $a \sqsubseteq b$ is expressed as an edge from a to b with a being below b.

Widening operator

To limit the size of types, the basic analysis uses a widening operator $\nabla_{n,m,o}$, which is parameterized with three numbers: $n \in \mathbf{N}$ is the maximum size of a set of types, $m \in \mathbf{N}$ is the maximum number of attributes of a class or instance and $o \in \mathbf{N}$ is the maximum nesting depth. If a union type exceeds one of the limits, it is replaced by $\top$.

The following example illustrates the effect of the n parameter:

$$\begin{aligned} \{int\} \nabla_{2,2,2} \{bool\} &= \{int, bool\} \\ \{int, bool\} \nabla_{2,2,2} \{bool, str\} &= \top \end{aligned}$$

On the second line, the join of the types is $\{int, bool, str\}$, but this is too large for the widening operator $\nabla_{2,2,2}$. The next example shows the effect of the m parameter:

$$\begin{aligned} \{class\langle 1, [], a \mapsto \{int\}\rangle\} \nabla_{2,2,2} & \\ \{class\langle 1, [], b \mapsto \{str\}\rangle\} &= \{class\langle 1, [], a \mapsto \{int\}, b \mapsto \{str\}\rangle\} \\ \{class\langle 1, [], a \mapsto \{int\}\rangle\} \nabla_{2,2,2} & \\ \{class\langle 1, [], b \mapsto \{str\}, c \mapsto \{bool\}\rangle\} &= \top \end{aligned}$$

The last example show the effect of the o parameter:

$$\begin{aligned} \{class\langle 1, [], a \mapsto \{float, int\}\rangle\} \nabla_{2,2,3} \bot &= \{class\langle 1, [], a \mapsto \{float, int\}\rangle\} \\ \{class\langle 1, [], a \mapsto \{float, int\}\rangle\} \nabla_{2,2,2} \bot &= \{class\langle 1, [], a \mapsto \top\rangle\} \end{aligned}$$

The types $float$ and int are at nesting depth 3, so for $\nabla_{2,2,2}$, they are too deeply nested, and the union type is replaced with $\top$.

Map lattice

Because the goal of the analysis is to determine a type for each variable in the program, the lattice UTy defined above is not the lattice used for the data flow analysis. Instead, the value computed for each program point is a mapping from variables to union types, called a *map lattice* here. The join operator for this lattice is defined as follows: $a \sqcup b$ contains all mappings present in either a or b; if a mapping is present in both, its values are combined by the join operator of the UTy lattice.

5.1.2 Getting information on types

This section describes how various Python constructs provide information on types to the analysis. In terms of data flow analysis, this corresponds to the transfer functions generated for different program points in the control flow graph.

Assignments

Assignments are the most obvious source for information on types. For example, the statements

```
x = 1
y = z
```

signal to the analysis that, after the statements are executed, *x* has type *int* and *y* has the same type as *z*. However, assignment statements in Python can be more complex than these. There are two types of assignment statements: regular assignments such as *x = 1* and augmented assignments such as *x += 1*.

A regular assignment statement has one or more targets on the left-hand side and an expression on the right-hand side, as in the following example:

```
x = y = 2*z+1
```

Here, *x* and *y* are the targets, which are assigned the value of the expression *2*z+1*. Note that multiple targets are a special case in the syntax; assignments are not expressions as in C or Java.

The analysis determines the type of the expression and uses the result to modify the type of each of the targets. Types of expressions are determined recursively using rules built into the analysis. For the most part, these are obvious; only the rules for arithmetic expressions are somewhat involved (these are described in Chapter 5 of [25]).

Targets in assignments can have multiple forms, which are handled differently by the analysis. A target can be:

- A variable: in this case, the type of the variable is set to the type determined for the right-hand side.
- A target list, as in the following example:

  ```
  x, y, *z = [1,2,3,4,5]
  ```

  This assigns the first element of the list to *x*, the second to *y* and all further elements to *z*. For a star target, such as *z in the example, the analysis filters out all sequence types (from the elements of the union type determined for the left-hand side) and assigns these to the variable. The other targets are assigned $\top$ by the basic analysis.
- An attribute reference, as in the following example:

  ```
  x.a = 1
  ```

  Here, the analysis looks up the type of *x*, selects class and instance types and modifies the type of attribute *a*.
- A subscription, such as *x[1]*, or a slicing such as *x[1:3]*: the basic analysis filters out the sequence types (strings, tuples, lists) from the type of *x*. Better support for these is part of the analysis variant for parameterized datatypes (Section 5.2.1).

An augmented assignment is an assignment of the form *lhs* α= *rhs*, where α is an operator such as $+$ or $-$. Its effect is that of the statement *lhs* = *lhs* α *rhs*, except that *lhs* is evaluated only once. The analysis thus determines the type of *lhs* α *rhs* and uses the rules described above for regular assignments to modify the type for *lhs*.

Del statements

The *del* statement is used to remove an identifier binding, to remove an attribute from a class or object or to remove one or several elements from a collection type. In the analysis, if the target of a *del* statement is a variable, its type is set to $\bot$; if it is an attribute reference, the attribute is removed from class and instance types; if it is a subscription or slicing, its type is not changed.

Functions

For a function definition, the analysis creates three nodes in the control flow graph in addition to those for the function body: entry and exit nodes (l_n and l_x) and a node for the function definition itself. It also generates a unique identifier i for each function.

The transfer function for the function definition is straightforward: it assigns a function type f_i, containing the function's unique identifier, to a variable corresponding to the function's name.

The program points for l_n and l_x are best considered together with those generated for function calls: the call (l_c) and return (l_r) nodes. The analysis adds edges from l_c to l_n and from l_x to l_r to represent the effect of function calls.

The transfer functions for l_c and l_n need to represent argument passing in terms of the analysis, i.e., they have to transfer the types of arguments. Arguments are represented as special variables α_0, α_1, etc. for the first, second, etc. argument. The transfer function for l_c determines the types of the expressions in the function call and sets the types for the argument variables accordingly; the transfer function for l_n reads out the types of argument variables, transfers them to variables for the function arguments and removes the argument variables.

This system works correctly for cases where one function call can refer to multiple functions (an l_c node has multiple outgoing edges) or a function is called from multiple call sites (an l_n node has multiple incoming edges). In the first case, the same argument variables are propagated along all the outgoing edges; in the second, argument types from different call sites are automatically combined with the join operator.

The transfer functions for l_x and l_r, together those for *return* statements, handle passing the types of return values. A function's return value is represented by the special variable ρ, just like arguments are represented by the variables α_0, α_1, etc. The transfer function for a *return* statement sets ρ to the type of its argument expression and the one for l_r transfers the type from ρ to a generated variable and removes ρ from the map lattice. The l_x node ensures

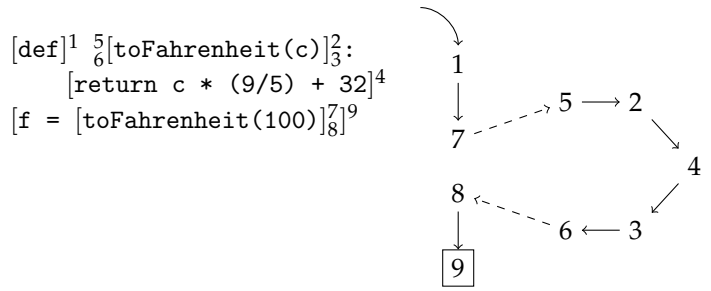

Figure 5.3: Control flow graph for function definition and function call.

that values for ρ from different *return* statements are joined and it also sets ρ to *NoneType* if it is not already set for the case where there is no *return* statement.

Scopes

As explained in Section 4.3.4, for each scope in the program code, two nodes s_n and s_x are created for entry to and exit from the scope. These program points are parameterized with the list of variables local to the scope. The transfer function for s_n adds each of these to the map lattice and sets its type to $\bot$; the transfer function for s_x removes these entries.

An Example

An example makes it clear how the various transfer functions for functions and scopes work together. Figure 5.3 shows a three-line program that defines a function to convert from degrees Celsius to degrees Fahrenheit and uses it to convert 100°C. When the analysis is applied to this program, the effect of the various transfer functions, in the order implied by the control flow graph, is as follows:

- For node 1: the type of variable *toFahrenheit* is set to $\{f_1\}$, 1 being the identifier assigned to the function.
- For node 7 (l_c): the argument variable α_0 is set to $\{int\}$, which is the type of the expression 100.
- For node 5 (s_n): variable c is set to $\bot$.
- For node 2 (l_n): the type for variable c is set to that of variable α_0, then α_0 is removed from the lattice.
- For node 4: using the current type lattice, the type of the expression *c * (9/5) + 32* is determined to be $\{float\}$ (the division *9/5* yields a floating-point value); this is assigned as the type of variable ρ.
- For node 3 (l_x): the transfer function checks if ρ is set, and since it is, does not change anything.

- For node 6 (s_x): the transfer function removes variable c.
- For node 8 (l_r): the type for the generated variable ι_1 is set to that of variable ρ, then ρ is removed from the lattice.
- For node 9: this is a simple assignment; it sets the type of f to that of ι_1.

In the end, the analysis correctly infers that the type of f is *float*.

5.1.3 Modules and import statements

Section 4.3 describes how control flow graphs for Python modules are created, but if the program under analysis consists of multiple modules, the analysis has to do two more things: create a single control flow graph for the program from the graphs for its modules, and let analysis results flow between modules according to the *import* statements in the code.

To create a control flow graph for the whole program, the analysis first creates a graph for each module, then combines the graphs. The combined graph contains all nodes and edges from the module graphs, and one additional node with label 0. Edges are added from node 0 to the initial nodes of all module graphs, and node 0 becomes the initial node of the combined graph. This ensures that the initial node has no incoming edges, which simplifies solving the monotone framework.

The contents of one module can be made available to another by an *import* statement. When an *import* statement is encountered at runtime, the module it refers to is loaded and executed (if it has not been executed already) and the names it defines are made available to the local namespace. An *import* statement thus implies flow of control to the module imported and back, similar to the way a function call implies flow of control to the function called and back. The representation of *import* statements in the analysis is therefore analogous to that of function calls: for each *import* statement, two nodes are generated, i_c and i_r, and edges are added from i_c to the initial node of the imported module and from its final nodes to i_r.

The transfer function for i_c does not need to do anything, but the transfer function for i_r has to rename variables as specified by the *import* statement. For example, for the statement

```
import module1 as module2
```

it needs to go through the map lattice and rename all variables of the form *module1.x* to *module2.x*. Similarly, for the statement

```
from module1 import x, y
```

it needs to look up variables *module1.x* and *module1.y* and turn them into variables in the local scope.

5.2 Analysis variants

This section presents several extensions and modifications of the basic analysis which are intended to make it faster or more precise. Each of these can be enabled independently from the others, so that any combination of variants is possible.

5.2.1 Parameterized datatypes

One limitation of the basic analysis is that it does not track the contents of the built-in collection types (lists, sets, dictionaries and tuples). Whenever values are stored in a collection, their type is thus lost to the analysis, so that when the contents of a collection are accessed, e.g. by a *for* loop or by a subscription, the analysis has to assign type $\top$ to the result.

The solution is to introduce parameterized datatypes such as $list\langle int\rangle$ meaning "list containing values of type *int*", or, more concisely, "list of int". The notation with angle brackets was chosen to resemble that of generic types in Java and C#.

Extended type lattice

To support parameterized datatypes, the type lattice is extended by adding to the definition of basic types:

$$b ::= \ldots \mid list\langle u\rangle \mid set\langle u\rangle \mid frozenset\langle u\rangle \mid dict\langle u; u\rangle \mid tuple\langle[u]\rangle$$

Each of the new types is parameterized with one or more union types. The *list* and *set* types take one parameter, as does the *frozenset* type, which is essentially the same as the *set* type except that *frozenset* objects are immutable. The *dict* (dictionary) type takes two parameters: one for the type of keys and one for the type of values.

The tuple type takes a list of parameters, so that each position is assigned a separate type. A parameterized tuple type could be defined more simply with only one parameter for all positions, but the form chosen here should give better precision in many cases.

To avoid visual clutter in the type expressions, when a type is parameterized with a set of types, it is written without the curly brackets, for instance, $list\langle int, float\rangle$ instead of $list\langle\{int, float\}\rangle$. Where a type has multiple parameters, they are separated by semicolons, while the elements of a union type are separated by commas, so that no ambiguity arises.

The join operator treats parameterized types specially, similar to the way it treats class and instance types. For example, if union types a and b contain types $list\langle u_a\rangle$ and $list\langle u_b\rangle$, respectively, then $a \sqcup b$ contains only one parameterized list type $list\langle u_a \sqcup u_b\rangle$. Set and dictionary types are handled analogously. For parameterized tuple types, only those of the same length are combined.

Figure 5.4 shows some of the elements of the lattice with parameterized datatypes in a diagram similar to Figure 5.2.

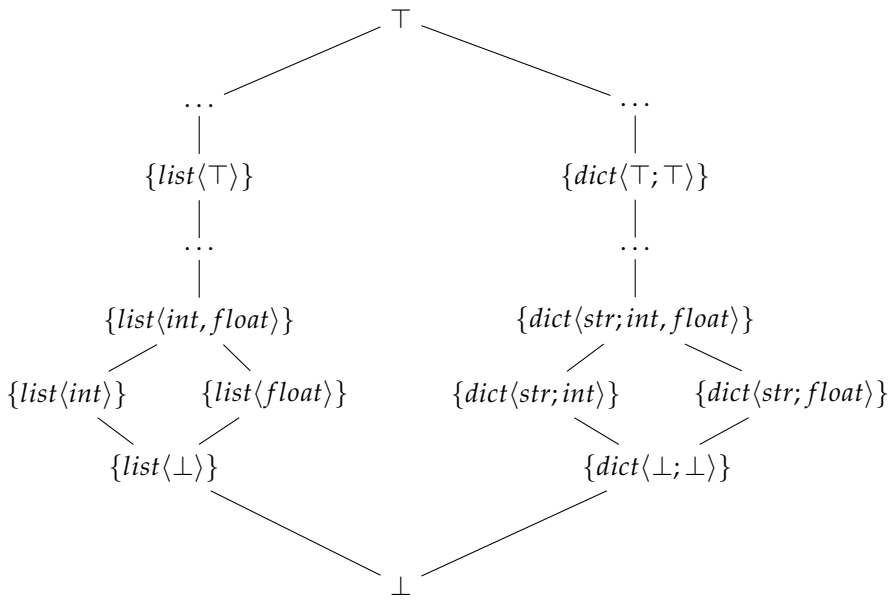

Figure 5.4: Part of the type lattice with parameterized datatypes.

The widening operator $\nabla_{n,m,o}$ described in Section 5.1.1 can be used unchanged for parameterized datatypes. Its parameter n was defined to limit the size of sets of types, so that it also applies to type parameters, as in the following examples:

$$\begin{aligned} set\langle int\rangle \ \nabla_{2,2,2} \ set\langle bool\rangle &= set\langle int, bool\rangle \\ set\langle int, bool\rangle \ \nabla_{2,2,2} \ set\langle bool, str\rangle &= set\langle \top\rangle \end{aligned}$$

Use of parameterized datatypes

Parameterized datatypes are used in a number of circumstances to improve analysis results:

- List, set, dictionary and tuple literals are assigned parameterized types. For example, the expression *(1, 1.5)* is assigned type $tuple\langle int; float\rangle$.
- Expressions that involve a subscription or slicing make use of parameterized types. For example, assuming type $\{dict\langle int; str\rangle\}$ has been inferred for variable *a*, type $\{str\}$ is inferred for the expression *a[1]*.
- Where the target of an assignment is a subscription or slicing, the parameters of parameterized types are modified correctly.
- Parameterized types are assigned to the results of list, set and dictionary comprehensions.

- In *for* loops, parameterized types are used to assign the most precise type possible to the loop variable.

The following example code illustrates several of these uses:

```
def g(x):
    return ("square", x*x)

def h(x):
    return ("half", x / 2)

functions = [g, h]
results = [f(1) for f in functions]
```

The basic analysis assigns type $\{list\}$ to both *functions* and *results*. Using the variant with parameterized datatypes, however, the analysis infers type $\{list\langle f_1, f_2\rangle\}$ for *functions* (1 and 2 being the identifiers assigned to g and h), which enables it to add the edges for the function call inside the list expression and infer the precise type $\{list\langle tuple\langle str; int, float\rangle\rangle\}$ for *results*.

5.2.2 Context-sensitive analysis

The second analysis variant implements context-sensitive analysis as described in Section 4.1.6. The type inference method uses call strings containing the labels of function calls as the context Δ.

In the implementation, context-sensitive analysis is done entirely in the worklist algorithm, so that no changes are necessary in the type lattice or the transfer functions. This helps keep the analysis implementation maintainable by keeping different aspects separate.

Because function call and return edges are already maintained by the worklist algorithm (see Section 4.2.1), it does not need any additional information for context-sensitive analysis. The worklist algorithm tags the edges in the control flow graph according to their function: edges in the monotone framework are tagged "regular", while those added for function calls are tagged "call" and "return".

Instead of the lattice L specified by the monotone framework, the mapping $\Delta \to L$ is used for context and effect values (thought not for the global value for flow-insensitive analysis). The complete lattice for the analysis is thus $\Delta \to (Var \to UTy)$, where *Var* means variables in Python code. However, because the $Var \to UTy$ mapping is not visible to the worklist algorithm (it only deals with the opaque lattice type) and the call strings are not visible to its users, no part of the implementation actually has to deal with this "double mapping" directly.

The empty call string [] is used as initial value for Δ. When processing a call edge (taken from the worklist), the label l_c of the call program point is added to the end of each call string; when processing a return edge, the worklist algorithm selects only those results where the last element of the call string matches the l_c label of the call site, and removes this element from the call string.

To ensure termination, the worklist algorithm takes a parameter k, which is the maximum length of a call string. This parameter can also be used to trade off precision for speed: depending on the patterns of function calls in the program code, a small value of k can lead to imprecise results, but it also reduces the number of separate results that are maintained by the analysis, which should make it faster. In the modified worklist algorithm, context-insensitive analysis is treated as context-sensitive analysis with $k = 0$.

5.2.3 Flow-insensitive analysis

Data flow analysis is basically flow-sensitive, but in some cases flow-insensitive analysis may be more logical. The analysis therefore includes three variants that use the extension for flow-insensitive analysis described in Section 4.2.2, which adds a "global value" with flow-insensitive results to the worklist algorithm's state. In each of these variants, the types for a certain class of variables are put in the global value, while for other types it still uses context and effect values.

Support for flow-insensitive analysis affects practically all transfer functions. When a transfer function looks up the type inferred for a variable, it first determines if flow-insensitive analysis should be used for that variable and, if so, looks it up in the global value instead of the context value and signals to the worklist algorithm that it used the global value. Similarly, when a transfer function modifies the type for a variable for which flow-insensitive analysis is used, it modifies the variable's entry in the global value and returns the new global value. As described above, the worklist algorithm then ensures transfer functions that use the global value are recomputed.

The three analysis variants for flow-insensitive analysis are described in the following. Each of them selects a different class of variables for flow-insensitive analysis, so any combination of the variants can be selected.

Flow-insensitive analysis for module-scope variables

The first variant uses flow-insensitive analysis for module-scope variables, that is, variables whose scope is not limited to a function or class definition. A module-scope variable can be modified by every function in its module, as well as other modules in which it is imported. Unlike variables with function scope, which are reset every time the function is executed, module-scope variables are essentially global variables. It makes sense, then, for the analysis to also treat their types as global.

Flow-insensitive analysis for class types

Classes in Python are very flexible. It is possible, for example, for a function to add an attribute to a class defined elsewhere, carry out its task using the extended class and remove the attribute afterwards. However, this would be seen as poor programming style by Python programmers. Because there is only one class object for each class, a modification of a class (adding, removing or

changing an attribute) affects all of the class's users. Therefore, this variant treats classes as global.

This is not quite as simple as for module-scope variables, however, because class types occur not only as the types of variables, but more often as part of other types, in particular as part of instance types.

The solution is to use a two-step process for looking up or modifying class types. Where a class type would be used in the basic analysis, a *class reference type*, which contains only the class identifier, is used instead. The actual class type is put in the global value as the type for a special *class identifier variable* containing the identifier of the class. When a transfer function looks up a type (in the context value or the global value) and finds a class reference type, it looks up the corresponding class identifier variable and uses or sets its type instead.

Writing class reference types as $class\langle l \rangle$, the type lattice defined in Figure 5.1 can be adapted for this variant by adding an alternative to the definition of *ClsTy*:

$$c ::= class\langle l, [c], \{n \mapsto u\} \rangle \mid class\langle l \rangle$$

Flow-insensitive analysis for instance types

Unlike classes, class instances are not global, and each instance has its own set of attributes independent of other instances. However, in a well-designed program, the instances of a class will tend to have the same attributes with the same types – otherwise the class's methods will not be able to make use of the instance attributes. The analysis therefore contains a variant that assigns the same type to all instances of a class.

The method used for flow-insensitive analysis of instance types is similar to the one described above for class types. *Instance reference types* are used, which only contain the class identifier, and the actual instance types are stored in the global value under an *instance identifier variable*. When an instance reference type is encountered, the instance identifier variable is looked up and its type is used instead.

Just like the previous variant modifies the definition of *ClsTy*, this one adds a clause for *InstTy*:

$$i ::= inst\langle c, \{n \mapsto u\} \rangle \mid inst\langle l \rangle$$

5.2.4 Manually specified types

For some modules, type inference cannot be used, because their source code is not available, because they are implemented in C or, for modules in the standard library, because they are implemented as part of the Python interpreter. For these cases, it is possible to specify their types manually using a plain-text format.

The following example gives types for two identifiers from the standard-library *math* module:

```
math.pi : {float}
math.sqrt : {lambda {bool, int, float} -> {float}}
```

Each line contains an identifier and a union type, separated by a colon. The second type is for a function that takes one argument of type *bool*, *int* or *float* and returns a *float*; the syntax used here is based on Python's syntax for anonymous functions.

Manually specified types are always treated as global (flow-insensitive). There is a special syntax for classes and instances: an identifier of the form

`class`*l*`.`*x* `:` *type*

specifies the type of an attribute of the class with identifier *l*. The syntax `class<`*l*`>` is then used to assign the corresponding class reference type to a variable, as in the following example:

```
class1.write : {lambda {bytes} -> {int}}
class1.flush : {lambda -> {NoneType}}
io.FileIO : {class<1>}
```

The syntax for instance types is the same, with the keyword *instance* instead of *class*.

Polymorphic function types

The syntax for manually specified types also allows for polymorphic function types. For example, the identity function:

```
def id(x):
    return x
```

can be provided with suitable type by the following specification:

```
m.id : {lambda !a -> !a}
```

The exclamation mark indicates a type variable. During the analysis, type variables are replaced with the types of function arguments.

Chapter 6

Experimental Evaluation

An important part of the thesis project was an experimental evaluation of the method using real-world Python code. This was done by applying the implementation described in Appendix A to the source code of five projects written in Python and measuring the precision and speed of the analysis. The goal of the experiments was to compare different analysis variants, evaluate the effect of its parameters and to assess its suitability for the intended uses.

This chapter first describes the method used to measure precision and speed and the projects the analysis was applied to before presenting the results of the experiments. A table with raw data for the results can be found in Appendix B.

6.1 Method

The evaluation was carried out by applying the implementation to all of a project's Python source code and measuring the precision of the results as well as the time needed for type inference. This was repeated for different variants and parameter settings, for each of five projects.

6.1.1 Measuring precision

The output of the method consists of mappings from identifiers to union types. For each program point, there are two such mappings, for context and effect, and there is one global mapping for variables inferred with flow-insensitive analysis. Ideally, an evaluation of this output would compare it to a *ground truth*, which means results known to be correct. Such a ground truth could be obtained by careful manual inspection of the source code, but because this would take a long time for all but the simplest programs, it would restrict the evaluation to a small sample of Python source code. Therefore, an algorithm was developed that automatically judges precision.

In order to focus on those analysis results that are relevant to a user, the algorithm starts from the control flow graph. For each node in the graph, it determines the identifiers used in the statement corresponding to the node. For example, for the statement

```
a = f(x + 1)
```

this would yield the identifiers f and x (but not a). The types for these identifiers are, presumably, the ones that a user would be interested in, since they determine the effect of the statement. The algorithm then looks up the type inferred for each of the variables in the context or global lattice value and adds the types to a list.

In the next step, the types in this list are classified in two groups: $\bot$ and $\top$ types are classified as "not useful", all others are classified as "useful". The final number is calculated as the ratio of "useful" types to all types in the list.

This method has the advantage that it selects, from the large number of types contained in the analysis results, those that are most likely to be relevant to a user. It also gives appropriate weights to types of identifiers for which flow-insensitive analysis is used: because the analysis infers only one type for each of these, but one type per program point for others, they might have a disproportionally small influence on the results of a simpler algorithm. The method also disregards the types inferred for generated identifiers, which are not in the original program.

6.1.2 Measuring speed

To measure speed, the implementation records the time just before and just after running the analysis, and prints the difference in microseconds (μs). The time measured is CPU time (the amount of time that the program has run on the CPU), which makes it less likely that the results are influenced by other factors such as the operating system's activities.

Haskell, the language that the implementation is written in, uses lazy evaluation: expressions are not evaluated before they are needed. This makes it difficult to measure the runtime of part of a program, because execution of different parts can be finely meshed together at runtime. To avoid this, the implementation uses the DeepSeq library[1] to force evaluation of the analysis's input before taking the start time and of its output before taking the end time.

The experiments were done on a MacBook with 2.4 GHz Intel Core 2 Duo processor and 2 GiB of main memory.

6.2 Example projects

In order to have a variety of Python source code represented in the experiments, five project were used, which are briefly presented here. These projects in particular were selected because they are compatible with Python 3.2 and they do not use modules written in C, which the analysis would not be able to process.

The projects are ordered here from most to least self-contained, so the later ones are likely to be more problematic for type inference:

[1] `http://hackage.haskell.org/package/deepseq`

	Modules	Lines of code
euler	5	110
adventure	6	2211
bitstring	6	4299
feedparser	2	4454
twitter	13	1868

Table 6.1: Projects used as input for the evaluation.

euler This codebase consists of solutions to five mathematical problems from the Project Euler website[2] written by the author. This is very straightforward code, which does not use features such as classes or exceptions.

adventure The Adventure[3] project is a port of the text-based *Colossal Cave Adventure* game from Fortran to Python 3. This is a fairly self-contained interactive program; having a text-based interface means it is not depended on a large GUI library.

bitstring The bitstring library[4] provides a convenient interface in Python for the creation and manipulation of binary data.

feedparser The Universal Feed Parser[5] is a library for parsing RSS and Atom feeds implemented in Python. It consists of only two modules, but there is quite a bit of nested code to handle all the details of various versions of the two standards.

twitter The Python Twitter Tools[6] contains a library, a command-line program, and an IRC bot to access the Twitter web site's public API.

Table 6.1 lists the number of modules and lines of code for each of the projects.

6.3 Results

The results of the experiments, in the form printed by the implementation program, can be found in Appendix B. This section presents various aspects of the results and uses them to evaluate the analysis variants, the influence of parameters and the general suitability of the method.

6.3.1 Variants

Tables 6.2–6.5 show the effect that the analysis variants have on precision and time. Results of each variant are compared here to the analysis with default parameters; the numbers shown are differences in percent. Thus, in the *precision*

[2] `http://projecteuler.net/`
[3] `https://bitbucket.org/brandon/adventure/overview`
[4] `http://code.google.com/p/python-bitstring/`
[5] `http://feedparser.org/`
[6] `http://mike.verdone.ca/twitter/`

	euler	adventure	bitstring	feedparser	twitter	*mean*
precision	11.54	0.00	0.00	1.01	0.00	2.51
time	-9.63	0.64	31.58	3.88	2.55	5.80

Table 6.2: Effects of parameterized datatypes on experiment results.

k		euler	adventure	bitstring	feedparser	twitter	*mean*
1	p	0.00	0.00	0.00	0.25	0.00	0.05
	t	36.91	67.92	0.26	117.14	0.16	44.48
2	p	0.00	3.21	0.00	0.25	0.00	0.69
	t	102.61	236.74	0.47	429.76	0.14	153.95
4	p	0.00	-9.55	0.00	0.25	0.00	-1.86
	t	295.61	505.51	0.47	420.43	0.17	244.44
8	p	0.00	-10.00	0.00	0.25	0.00	-1.95
	t	971.09	1395.62	0.46	420.33	0.16	557.53
16	p	0.00	-10.00	0.00	0.25	0.00	-1.95
	t	3577.37	4123.39	0.52	420.40	0.16	1624.37

Table 6.3: Effects of context-sensitive analysis on experiment results.

rows, positive numbers indicate better results; in the *time* rows, positive numbers indicate slower operation. The *mean* column contains the arithmetic mean of the values.

As can be seen in Table 6.2 and Table 6.3, parameterized datatypes and context-sensitive analysis did not improve the results by much. Parameterized datatypes did improve results significantly for the *euler* example code, but not for the larger projects. Context-sensitive analysis, which was tested for different values of the parameter k (maximum length of call strings) did not significantly improve the results in any of the cases.

The results of flow-insensitive analysis, shown in Table 6.4, are more encouraging. The first column of the table indicates what flow-insensitive analysis was used for. It contains the parameters passed to the implementation's command-line interface (see Section A.3): f means flow-insensitive analysis is used for module-scope variables, g means it is used for class types, and h means it is used for instance types. All seven possible combinations were used.

The best way to use flow-insensitive analysis appears to be to use it only for module-scope variables. Enabling it also for class or instance types or both improves the results in some cases, but not by much and at a large cost in speed.

For the last set of experiments, types were specified manually for identifiers not in the code under analysis (from the standard library or third-party libraries). Because of time constraints, this was only done for the two smallest projects. Table 6.5 shows the results; not surprisingly, they indicate that specifying types manually improves precision at a moderate cost in runtimes.

		euler	adventure	bitstring	feedparser	twitter	*mean*
f	p	5.05	73.13	48.23	10.37	-11.13	25.13
	t	-42.76	18.14	-22.92	12.70	-65.07	-19.98
g	p	0.00	20.90	55.77	0.17	0.00	15.37
	t	-0.01	1.87	-8.91	1356.03	-24.58	264.88
h	p	0.00	0.00	0.00	0.00	0.00	0.00
	t	-2.35	-0.09	0.10	-1.62	-0.02	-0.79
fg	p	5.05	78.19	48.23	7.58	-11.13	25.58
	t	-42.87	10.57	-46.50	938.19	-68.37	158.20
fh	p	5.05	78.19	48.23	6.58	-11.13	25.38
	t	-42.80	78.11	-22.35	339.55	-63.57	57.79
gh	p	0.00	20.90	55.77	0.17	0.00	15.37
	t	0.68	13.43	-9.55	796.72	-24.71	155.31
fgh	p	5.05	78.19	48.23	7.58	-11.13	25.58
	t	-42.45	31.86	-46.48	536.35	-67.02	82.45

Table 6.4: Effects of flow-insensitive analysis on experiment results.

	euler	adventure
precision	70.16	39.01
time	1.81	6.81

Table 6.5: Effects of manually specified types on experiment results.

6.3.2 Parameters for widening operator

Table 6.6 shows the results of different values for the parameters of the widening operator $\nabla_{n,m,o}$ (see Section 5.1.1), compared to the default settings $n = 3$, $m = 3, o = 20$. As may be expected, very small values for any of the parameters lead to poor precision and very large values lead to long runtimes. Other than that, the results indicate that the default values are actually good choices.

6.3.3 Evaluation

Table 6.7 shows the results in absolute numbers for the configuration that, according to the experiments, works best: using flow-insensitive analysis for module-scope variables, but none of the other variants. The analysis inferred a useful type for variables in the source code in between 45 and 91 percent of cases. When it was supplied with types for identifiers in external libraries, this number increased further (by 6 and 27 percent for the two projects used).

To do this, it needed between 0.014 and 23 seconds. The differences are in part explained by the size of the projects, but not entirely. For example, the analysis took 9.1 times longer for the *feedparser* project than for the *bitstring* project, but the difference in lines of code is only about 4 percent.

		euler	adventure	bitstring	feedparser	twitter	*mean*
$n = 1$	p	-23.08	0.00	-5.77	-0.25	0.00	-5.82
	t	-0.43	-0.12	0.35	0.14	0.06	-0.00
$n = 2$	p	0.00	0.00	0.00	0.00	0.00	0.00
	t	0.36	-0.16	0.28	0.25	-0.05	0.13
$n = 4$	p	0.00	0.00	0.00	0.00	0.00	0.00
	t	0.16	-0.33	0.17	0.18	-0.01	0.03
$n = 8$	p	0.00	0.00	0.00	0.00	0.00	0.00
	t	0.08	-0.12	0.26	-0.03	-0.00	0.04
$n = 16$	p	0.00	0.00	0.00	0.25	0.00	0.05
	t	-0.00	-0.09	0.23	-0.30	-0.03	-0.04
$m = 1$	p	0.00	0.00	0.00	0.00	0.00	0.00
	t	0.26	-0.25	0.20	-0.28	0.04	-0.01
$m = 2$	p	0.00	0.00	0.00	0.00	0.00	0.00
	t	0.28	-0.23	0.32	-0.34	-0.02	0.00
$m = 4$	p	0.00	0.00	0.00	0.00	0.00	0.00
	t	0.05	-0.32	0.25	-0.33	0.06	-0.06
$m = 8$	p	0.00	0.00	0.00	0.00	0.00	0.00
	t	0.42	-0.26	0.42	-0.22	-0.05	0.06
$m = 16$	p	0.00	0.00	0.00	0.00	0.00	0.00
	t	-0.08	0.01	0.25	-0.39	0.05	-0.03
$o = 4$	p	0.00	0.00	-1.92	-1.14	-21.62	-4.94
	t	0.00	-15.10	-9.61	-4.61	-12.47	-8.36
$o = 8$	p	0.00	0.00	0.00	-0.25	0.00	-0.05
	t	0.16	-6.27	-5.39	-2.68	0.09	-2.82
$o = 16$	p	0.00	0.00	0.00	0.00	0.00	0.00
	t	0.51	-0.08	0.32	-1.88	-0.08	-0.24
$o = 32$	p	0.00	0.00	1.92	1.06	0.00	0.60
	t	0.34	-0.10	9.58	13.97	-0.09	4.74
$o = 64$	p	0.00	0.00	3.85	-2.20	0.00	0.33
	t	1.00	-0.25	30.14	2267.87	0.25	459.80

Table 6.6: Effects of parameters of widening operator on experiment results.

		euler	adventure	bitstring	feedparser	twitter
	precision	0.45	0.75	0.91	0.53	0.75
	time	14	981	2,531	22,929	3,114
with user-defined types	precision	0.72	0.81			
	time	18	1,075			

Table 6.7: Precision and runtime (in *ms*) using flow-insensitive analysis for module-scope variables.

Chapter 7

Conclusions

This thesis presents a method for type inference based on data flow analysis for the dynamically typed Python programming language. To account for the language's dynamic nature, the basic data flow analysis was extended to support adding edges for function calls to the control flow graph during the computation of types. The resulting analysis is able to deal with first-class functions and Python's dynamic class system. It also supports (multiple) inheritance, modules and *import* statements, and most of the statements, expressions and built-in types of Python.

The method was implemented in a proof-of-concept implementation, which includes six variants that extend or modify the basic method and which was used to do an experimental evaluation of the method. The results of the evaluation show that the method can infer types with reasonable precision fairly quickly (on the order of milliseconds or seconds).

7.1 Future work

There are several directions in which future research could continue. One obvious way to improve the method would be to add proper support for those features of the Python language that the method described here does not handle well: exceptions, generators, and the *with* statement. Of these, exceptions are the most challenging, since exceptions circumvent the normal flow of control, so having support for exceptions would complicate construction of control flow graphs significantly.

One way to better support the use in editors and IDEs, which was the original motivation for the method, would be to enable it to infer types incrementally for different program parts. Ideally, it should be possible to apply type inference once to a whole program and then, whenever the user modifies the source code, apply it only to the part that was changed. The challenge here is to reconcile type inference for part of a program with the underlying data flow analysis, which normally operates on the control flow graph for an entire program.

Since the analysis already supports user-supplied type specifications, one

way to improve precision would be to ask the user to supply types for certain problematic definitions in the source code. To do this, the analysis would need a way to go backwards from inferred types to definitions in the source code and determine which definitions were the cause of poor results.

Appendix A

Implementation

Part of the thesis project was an implementation of the type inference method. This implementation was developed to show that the method is practicable and to make the experimental evaluation described in Chapter 6 possible.

The implementation is written in the functional programming language Haskell [14] and built using the Cabal system [13]. It is structured into three separate Cabal packages: the *data-flow-analysis* package is a library for data flow analysis, the *python-type-inference* package is an implementation of the type inference method, and the *infer-python-types* package provides a command-line interface to the implementation.

A.1 Data Flow Analysis

The *data-flow-analysis* package implements a library for data flow analysis. It includes features to support dynamic languages (as described in Section 4.2), but is not specific to Python. It also implements context-sensitive analysis using call strings; library users only need to specify the maximum length of call strings to use this.

A.2 Type Inference

The *python-type-inference* package is the main part of the implementation; it contains the type inference method including the six analysis variants. It has four main tasks: parsing Python source code, creating control flow graphs, creating monotone frameworks for type inference and handling manually specified types. The *TypeInference* module serves as a high-level interface for these.

To parse Python source code, the implementation uses the *language-python* library[1]. It also uses the *data-flow-analysis* library.

[1] `http://hackage.haskell.org/package/language-python`

A.3 Command-line Interface

The *infer-python-types* package provides a command-line program, which is also called *infer-python-types*. It takes a list of Python source files as command-line arguments as well as parameters that specify the analysis variants and parameters to be used. When called without any parameters, the program prints a usage message explaining possible parameters.

Appendix B

Results of Experiments

The numbers gathered by the experiments described in Chapter 6 are listed in two tables in this appendix. Table B.2 contains the measurements of precision, while Table B.3 contains the runtime in microseconds. In the tables, each column corresponds to one of the projects used as input, and each row corresponds to a different configuration. The configurations are specified by the parameters passed to the command-line interface of the implementation; the first row contains the results for the default configuration. The relevant parameters are shown in Table B.1.

Parameter	Effect
-p	Use parameterized types.
-k*k*	Use context-sensitive analysis; parameter *k* gives the maximum length for call strings.
-f	Use flow-insensitive analysis for module-scope variables.
-g	Use flow-insensitive analysis for class types.
-h	Use flow-insensitive analysis for instance types.
-n*n*	Set parameter *n* of widening operator (default: 3).
-m*m*	Set parameter *m* of widening operator (default: 3).
-o*o*	Set parameter *o* of widening operator (default: 20).
-u *file*	Pass a file with manually specified types.

Table B.1: Command-line parameters used in the experiments.

	euler	adventure	bitstring	feedparser	twitter
	0.42623	0.43478	0.61176	0.47716	0.84091
-p	0.47541	0.43478	0.61176	0.48197	0.84091
-k1	0.42623	0.43478	0.61176	0.47837	0.84091
-k2	0.42623	0.44872	0.61176	0.47837	0.84091
-k4	0.42623	0.39326	0.61176	0.47837	0.84091
-k8	0.42623	0.39130	0.61176	0.47837	0.84091
-k16	0.42623	0.39130	0.61176	0.47837	0.84091
-f	0.44776	0.75275	0.90681	0.52664	0.74729
-g	0.42623	0.52564	0.95294	0.47798	0.84091
-h	0.42623	0.43478	0.61176	0.47716	0.84091
-fg	0.44776	0.77473	0.90681	0.51332	0.74729
-fh	0.44776	0.77473	0.90681	0.50858	0.74729
-gh	0.42623	0.52564	0.95294	0.47798	0.84091
-fgh	0.44776	0.77473	0.90681	0.51332	0.74729
-n1	0.32787	0.43478	0.57647	0.47596	0.84091
-n2	0.42623	0.43478	0.61176	0.47716	0.84091
-n4	0.42623	0.43478	0.61176	0.47716	0.84091
-n8	0.42623	0.43478	0.61176	0.47716	0.84091
-n16	0.42623	0.43478	0.61176	0.47837	0.84091
-m1	0.42623	0.43478	0.61176	0.47716	0.84091
-m2	0.42623	0.43478	0.61176	0.47716	0.84091
-m4	0.42623	0.43478	0.61176	0.47716	0.84091
-m8	0.42623	0.43478	0.61176	0.47716	0.84091
-m16	0.42623	0.43478	0.61176	0.47716	0.84091
-o4	0.42623	0.43478	0.60000	0.47172	0.65909
-o8	0.42623	0.43478	0.61176	0.47596	0.84091
-o16	0.42623	0.43478	0.61176	0.47716	0.84091
-o32	0.42623	0.43478	0.62353	0.48220	0.84091
-o64	0.42623	0.43478	0.63529	0.46667	0.84091
-u types	0.72527	0.60440			
-f -u types	0.72165	0.81463			

Table B.2: Results for precision with different options.

	euler	adventure	bitstring	feedparser	twitter
	23,874	830,750	3,283,706	20,345,443	8,913,312
-p	21,574	836,070	4,320,555	21,135,129	9,140,365
-k1	32,685	1,394,964	3,292,313	44,177,870	8,927,858
-k2	48,371	2,797,499	3,299,257	107,782,824	8,926,053
-k4	94,448	5,030,276	3,299,123	105,884,334	8,928,088
-k8	255,713	12,424,888	3,298,955	105,862,736	8,927,310
-k16	877,936	35,085,835	3,300,638	105,877,870	8,927,659
-f	13,665	981,447	2,531,216	22,929,003	3,113,694
-g	23,872	846,304	2,991,282	296,235,225	6,722,823
-h	23,314	830,017	3,286,904	20,016,764	8,911,420
-fg	13,640	918,530	1,756,741	211,224,706	2,818,935
-fh	13,656	1,479,688	2,549,782	89,428,219	3,246,991
-gh	24,036	942,327	2,970,272	182,441,534	6,710,903
-fgh	13,739	1,095,464	1,757,497	129,468,212	2,939,887
-n1	23,771	829,746	3,295,219	20,373,761	8,918,274
-n2	23,959	829,416	3,292,825	20,395,370	8,909,148
-n4	23,913	827,967	3,289,136	20,381,050	8,912,085
-n8	23,892	829,786	3,292,363	20,340,253	8,913,138
-n16	23,873	830,014	3,291,249	20,285,058	8,911,015
-m1	23,936	828,697	3,290,213	20,289,249	8,916,918
-m2	23,941	828,812	3,294,320	20,276,008	8,911,845
-m4	23,887	828,061	3,291,808	20,277,605	8,918,676
-m8	23,974	828,629	3,297,660	20,300,891	8,909,041
-m16	23,854	830,858	3,291,867	20,266,762	8,917,586
-o4	23,874	705,337	2,968,097	19,407,861	7,801,578
-o8	23,912	778,636	3,106,767	19,800,853	8,921,605
-o16	23,996	830,091	3,294,161	19,963,922	8,906,573
-o32	23,956	829,904	3,598,158	23,187,853	8,905,257
-o64	24,112	828,677	4,273,258	481,753,372	8,935,829
-u types	24,306	887,305			
-f -u types	17,951	1,074,882			

Table B.3: Results for runtime with different options, in μs.

Bibliography

[1] Ole Agesen. Concrete type inference: Delivering object-oriented applications. *Dissertation*, Jan 1996.

[2] Jong-hoon An, Avik Chaudhuri, Jeffrey S Foster, and Michael Hicks. Dynamic inference of static types for Ruby. *Technical Report, University of Maryland*, Jul 2010.

[3] John Aycock. Aggressive type inference. *8th International Python Conference*, Jan 2000.

[4] Patrick Camphuijsen, Jurriaan Hage, and Stefan Holdermans. Soft typing PHP with PHP-validator. *Technical Report, Utrecht University*, Feb 2009.

[5] Brett Cannon. Localized type inference of atomic types in Python. *Master Thesis*, Jun 2005.

[6] Robert Cartwright and Mike Fagan. Soft typing. *PLDI '91: 1991 Conference on Programming Language Design and Implementation*, Jun 1991.

[7] Luis Damas and Robin Milner. Principal type-schemes for functional programs. *POPL '82: 9th Annual Symposium on Principles of Programming Languages*, Jan 1982.

[8] Cormac Flanagan. Hybrid type checking. *POPL '06: 33rd Annual Symposium on Principles of Programming Languages*, Jan 2006.

[9] Michael Furr, Jong-hoon An, and Jeffrey S Foster. Profile-guided static typing for dynamic scripting languages. *OOPSLA '09: 24th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct 2009.

[10] Michael Furr, Jong-hoon An, Jeffrey S Foster, and Michael Hicks. Static type inference for Ruby. *SAC '09: 24th Annual ACM Symposium on Applied Computing*, Mar 2009.

[11] Michael Gorbovitski, Yanhong A Liu, Scott D Stoller, Tom Rothamel, and K Tuncay Tekle. Alias analysis for optimization of dynamic languages. *DLS '10: 6th Symposium on Dynamic Languages*, Oct 2010.

[12] Lintaro Ina and Atsushi Igarashi. Towards gradual typing for generics. *STOP '09: 1st International Workshop on Script to Program Evolution*, Jul 2009.

[13] Isaac Jones. The Haskell Cabal, a common architecture for building applications and libraries. 2005.

[14] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Apr 2003.

[15] Martin Madsen, Peter Sørensen, and Kristian Kristensen. Ecstatic – type inference for Ruby using the cartesian product algorithm. *Master Thesis*, Jun 2007.

[16] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. The definition of standard ML (revised). May 1997.

[17] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. 2005.

[18] Jens Palsberg and Michael Schwartzbach. Object-oriented type inference. *OOPSLA '91: 6th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Nov 1991.

[19] Frédéric Pluquet, Antoine Marot, and Roel Wuyts. Fast type reconstruction for dynamically typed programming languages. *DLS '09: 2009 Symposium on Dynamic Languages*, Oct 2009.

[20] Michael Salib. Starkiller: A static type inferencer and compiler for Python. *Master Thesis*, May 2004.

[21] Jeremy Siek and Walid Taha. Gradual typing for functional languages. *Scheme and Functional Programming Workshop 2006*, Sep 2006.

[22] Jeremy Siek and Walid Taha. Gradual typing for objects. *ECOOP '07: 21st European Conference on Object-Oriented Programming*, Jul 2007.

[23] Jeremy Siek and Manish Vachharajani. Gradual typing with unification-based inference. *DLS '08: 2008 Symposium on Dynamic Languages*, Jul 2008.

[24] Swaroop Sridhar, Jonathan S Shapiro, and Scott F Smith. Sound and complete type inference for a systems programming language. *APLAS '08: 6th Asian Symposium on Programming Languages and Systems*, Dec 2008.

[25] Guido van Rossum and Fred L Drake. The Python language reference, release 3.2. Mar 2011.