UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING

Electronics and Computer Science

Preemptive type checking in dynamically typed programs

by

Neville Grech

Thesis for the degree of Doctor of Philosophy

November 2013

The research work disclosed in this publication is partially funded by the Strategic Educational Pathways Scholarship (Malta). This Scholarship is part-financed by the European Union – European Social Fund (ESF) under Operational Programme II – Cohesion Policy 2007-2013, "Empowering People for More Jobs and a Better Quality Of Life".

Operational Programme II – Cohesion Policy 2007-2013
Empowering People for More Jobs and a Better Quality of Life
Scholarship part-financed by the European Union
European Social Fund (ESF)
Co-financing rate: 85% EU Funds; 15%National Funds

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING
Electronics and Computer Science

Doctor of Philosophy

PREEMPTIVE TYPE CHECKING IN DYNAMICALLY TYPED PROGRAMS

by Neville Grech

With the rise of languages such as JavaScript, dynamically typed languages have gained a strong foothold in the programming language landscape. These languages are very well suited for rapid prototyping and for use with agile programming methodologies. However, programmers would benefit from the ability to detect type errors in their code early, without imposing unnecessary restrictions on their programs.

Here we describe a new type inference system that identifies potential type errors through a flow-sensitive static analysis. This analysis is invoked at a very late stage, after the compilation to bytecode and initialisation of the program. It computes for every expression the variable's present (from the values that it has last been assigned) and future (with which it is used in the further program execution) types, respectively. Using this information, our mechanism inserts type checks at strategic points in the original program. We prove that these checks, inserted as early as possible, preempt type errors earlier than existing type systems. We further show that these checks do not change the semantics of programs that do not raise type errors.

Preemptive type checking can be added to existing languages without the need to modify the existing runtime environment. We show this with an implementation for the Python language and demonstrate its effectiveness on a number of benchmarks.

Contents

List of Figures

Declaration of Authorship

I, Neville Grech , declare that the thesis entitled *Preemptive type checking in dynamically typed programs* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as: Neville Grech, Julian Rathke, Bernd Fischer. *Preemptive Type Checking in Dynamically Typed Languages*. In *Proceedings of ICTAC*, 2013.

Signed:

Date: 7th November 2013

Acknowledgements

The expert guidance and patience of both my supervisors Julian Rathke and Bernd Fischer were instrumental to the completion of this doctoral thesis, and I cannot thank them enough for this. I would also like to thank the kind people around me, both in the UK and in Malta for their help and support. My parents and friends have given me their unequivocal support throughout. Thanks goes to my fiancée Marilyn for supporting me and accompanying me in Southampton. Finally, I would not have been able to pursue a doctorate without the competitive funding provided by the School of Electronics and Computer Science and STEPS Malta.

Programming language theory is one of the most important branches in computer science, and I encourage people to learn more about this very interesting field.

Chapter 1

Introduction

In a dynamically typed language such as Python [39], the principle type of any variable or expression in a program is determined through runtime computations and can change throughout the execution and between different runs. Type checking is typically carried out as the program is executing and type errors manifest themselves as runtime errors or exceptions rather than being detected before execution. However, badly typed programs are potentially dangerous. For example, the Mars climate orbiter crashed into the atmosphere due to *metric mixup* [1], a form of type incompatibility that can be detected by some type systems [59, 60]. Type incompatibilities are an indication that the code has latent errors and therefore the earlier they are detected the earlier the code can be fixed.

Figure 1.1 shows a small example program that takes user input either from the screen or as command line arguments and calculates a result based on this and further input. In a statically typed language, compilation of this program should fail with multiple type errors. Namely, at

```
1  from sys import argv
2
3  def compute(x1=None,x2=None,x3=None):
4      global initial
5      if initial%5==0:
6          fin=int(input('enter final value: '))
7          return x1+x2+x3+fin
8      else:
9          initial-=1
10         return compute(x2,x3,initial)
11
12 def main():
13     global initial
14     if len(argv)<2:
15         initial=abs(input('enter initial value: '))
16     else:
17         initial=abs(argv[1])
18     print('outcome:',compute())
19
20 if __name__=='__main__':
21     main()
```

Figure 1.1: Dynamically typed program with type errors.

lines 15 and 17, function `abs` is given a string instead of a numeric type. Also, at line 7, the addition operations could be called with `None` and Integer arguments. In a dynamically typed language, the program will fail at either line 15 or line 17, depending on the arguments passed to the program. If this program is executed using the standard Python interpreter without passing command line arguments, we get the following interaction on the shell prompt:

```
$ python foo.py
enter initial value: 45
Traceback (most recent call last):
  File "foo.py", line 21, in <module>
    main()
  File "foo.py", line 15, in main
    initial=abs(input('enter initial value: '))
TypeError: bad operand type for abs(): 'str'
```

We can see that the program only raised a TypeError when it hit line 15, after user input has been taken.

In this thesis we introduce the concept of *type error preemption* for dynamically typed languages. Our goal is to force the termination of the program execution as soon as it can be detected that a type error is inevitable. In some cases, this can be even before the program execution starts. Our analysis, which is at the core of preemptive type checking, infers the potential types for every variable and expression and tries to find the earliest point from which a program is guaranteed to raise a TypeError. If we analyse this program *statically* with preemptive type checking enabled, we are presented with a few "potential" type errors, among which is the error described above.

Furthermore, with preemptive type checking enabled, the `main` function gets automatically transformed to:

```
def main():
    raise PreemptiveTypeError('Type mismatch ...')
    global initial
    if len(argv)<2:
        initial=abs(input('enter initial value: '))
    else:
        initial=abs(argv[1])
    print('outcome:',compute())
```

This terminates the program as soon as the `main` function is called and therefore reduces the time required for testing since no user input is needed for the error to be raised. Now, we assume that the user "fixes" this bug and rewrites the `main` function as in Figure 1.2.

```
12 def main():
13     global initial
14     if len(argv)<2:
15         initial=abs(int(input('enter initial value: ')))
16     else:
17         initial=abs(int(argv[1]))
18     print('outcome:',compute())
```

Figure 1.2: Modified main function.

When this program is run without preemptive type checking, the user notices that depending on the input, the program will either raise a TypeError or work as expected, for example:

```
$ python foo.py
enter initial value: 3
enter final value: 3
outcome: 6

$ python foo.py
enter initial value: 2
enter final value: 3
Traceback (most recent call last):
  File "foo.py", line 21, in <module>
    main()
  File "foo.py", line 18, in main
    print('outcome:',compute())
  File "foo.py", line 10, in compute
    return compute(x2,x3,initial)
  File "foo.py", line 10, in compute
    return compute(x2,x3,initial)
  File "foo.py", line 7, in compute
    return x1+x2+x3+fin
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

As we can see, not only is the manual testing process time consuming, but automated testing mechanism techniques will not necessarily produce the right combination of inputs to find these errors.

```
def compute(x1=None,x2=None,x3=None):
    global initial
    if initial%5==0:
        # begin inserted type check
        if not isinstance(x1, Number):
            raise PreemptiveTypeError(...)
        if not isinstance(x2, Number):
            raise PreemptiveTypeError(...)
        # end inserted type check
        fin=int(input('enter final value: '))
        return x1+x2+x3+fin
    else:
        initial-=1
        return compute(x2,x3,initial)
```

Figure 1.3: Transformed version of the `compute` function.

With preemptive type checking we can minimise this effort and preempt type errors much quicker. Our analysis infers that `x1` and `x2` are either of type `NoneType` or Integer, depending on the control flow taken by the program. Our analysis also concludes that `x1` and `x2` need to be integers for the program not to raise type errors. By simply *statically* analysing this program with preemptive type checking, we can get the following output:

```
Failure 1 - partial Traceback:
File "foo.py", line 18, in main
File "foo.py", line 6, in compute
Variable x1 expected Number but found NoneType

Failure 2 - partial Traceback:
File "foo.py", line 18, in main
File "foo.py", line 10, in compute
File "foo.py", line 6, in compute
Variable x1 expected Number but found NoneType

Failure 3 - partial Traceback:
File "foo.py", line 18, in main
File "foo.py", line 10, in compute
File "foo.py", line 10, in compute
File "foo.py", line 6, in compute
Variable x1 expected Number but found NoneType
```

We note that these are not the only failures that can occur, but our mechanism will only flag the failures that are guaranteed to happen first, so as not to confuse the user. We can see that for this particular example there are no false positives and all errors can occur when this program is executed. Another point is that determining the possible types of `x1` and `x2` is difficult and expensive. For example, using data flow analysis techniques, the fact that `x1` can be an integer is only discovered on a path that inlines function `compute` three times. In this thesis we will be introducing an effective technique that uses *trails* (see Section 4.4) to perform a flow sensitive type inference. Preemptive type checking can also transform the `compute` function so that any type errors are preempted (see Figure 1.3). Note that the assertions cannot be inserted any earlier (i.e., before the if-statement) because there are possible control flow paths that do not raise type errors. Moreover, the inserted assertions would contain all details to identify the source of the type error, in particular the variable causing the type error, the location where the

type error would be raised and the present type there. Hence, the user can correct the program with minimal debugging. There are no implementations of soft typing [22] or gradual typing [96, 98] that are sophisticated enough to handle this example. However, if soft typing or gradual typing were used in this case, any implementation would insert a type check right before the additions of `x1` and `x2`. This however means that the user input still has to take place before the type error can be raised.

In dynamic, gradual or soft typing, these errors are only detected at the point when a value of an incorrect type is used. In our case, we guarantee that the inserted runtime checks preempt all type errors. Secondly, we guarantee that these do not affect the semantics of programs that do not raise runtime type errors. Finally, we guarantee, using our linear optimality condition, that type errors are preempted at an optimal (early) point.

The usefulness of a type system is not just its ability to simply reject bad (and sometimes good) programs. Rather, its usefulness is the ability to prevent failures from occurring and the ability to hand enough information to the programmer so that he can debug the errors. In dynamically typed languages, errors are raised when the arguments to a function application do not match the function's signature. This is not always helpful. In contrast, with preemptive type checking, our generated runtime checks also indicate the locations where type errors can potentially originate. A programmer can therefore inspect these points to better understand where a type error can occur and where it can come from before executing the program.

1.1 Dynamically typed languages

Types are a very important concept in programming languages. A type is a set of values of a particular kind, for example, the set of all natural numbers. In this view, a type can be defined as a predicate over the universal set. Alternatively, types can also be seen as abstractions of terms in a programming language: a program is made up from operations, which are restricted by the type of the terms that they can operate upon. A program is called well-typed if for all the operations in the program, the type of the data that these operations operate upon matches the type of the operation. One way to perform type checking is to do so at runtime: just as an operation is about to be invoked at runtime, a type check is performed. If the operation was not originally defined over the type of the value passed to the operation, a type error exception is raised. This form of type checking is called dynamic type checking. Therefore, not all syntactically valid programs are well-typed. If a program is not well-typed, it is possible for an operation to be used on a value with an incompatible type. Such a program is said to be ill-typed. A *type checker* accepts well-typed programs and rejects ill-typed programs. Hence, the role of type checking is to make sure that every operation is only called with values over which that operation is defined.

The typing discipline typically dictates the way the type checking is carried out in a programming language. In *static type checking*, a proof is constructed that no operation is called with values over which the operation is not defined. If such a proof cannot be produced cheaply by

the type checking rules of the particular language, then the program is statically rejected and labelled as ill-typed. This is known as *static type checking* since the checking is done before executing the program.

Dynamic type checking is a mechanism where most of the type checking occurs at runtime. In a dynamically typed language, the type of any expression or variable is determined through runtime computations. In most popular [3] dynamic language implementations such as Python [39], Ruby [68], JavaScript [2] and PHP, type checking is carried out as the program is executing, while operations are being called. A type error manifests itself as a runtime error or exception rather than being detected by a static type checker.

Dynamically typed languages have been in existence since the 1950s. Lisp [69] was the first dynamically typed high level language. A number of features were pioneered or popularised in this language and new features continued to be introduced in dynamically typed languages. Some examples include hygienic macros and `call/cc` in Scheme [103] and garbage collection in Smalltalk [46] and some versions of Lisp. Just-in-time compilation also originated from Lisp [12]. Dynamically typed languages are largely interpreted and tend to sacrifice performance and static safety guarantees to flexibility and simplicity. Several authors have been working on ways to speed up the execution of these languages [50, 16, 44, 43]. In our research, we tackle the problem of preempting type errors, since just-in-time compilation has significantly reduced the former problem in recent years [43].

A large body of work on reconciling both typing disciplines has emerged over the years. *Soft typing* [22] is a type system where the type checker implicitly converts a dynamically typed program into a statically typed one. It does this by inserting explicit runtime type checks (also called narrowing functions) around the arguments of primitive operations [22]. *Gradual typing* [96, 98] mostly focuses on allowing statically typed portions of code to interact with dynamically typed portions. In a type system which can make this happen, the statically typed portions of the program can be statically type checked, while the dynamically typed portions are checked at runtime. Similar technology has already been adopted in the software industry and C# [76, 70] and Java [88] allow portions of the code to be dynamically typed. Other type systems are discussed in Chapter 2. The existing work simply allows statically typed portions of the code to interact with dynamically typed portions. No current work focuses on detecting runtime type errors as early as possible.

1.2 Problem statement

This thesis tackles the problem of preempting type errors in dynamically typed programs as early as possible. This is a difficult problem especially since we are trying to not restrict the power of the language.

The problem we are dealing with is multi-faceted. Firstly, we would like to find all possible type errors in a dynamically typed language. Secondly, we would also like to find these type errors as early as possible. Lastly, however, we do not want to transform a dynamically typed language into a statically typed one. We would like to retain a language with a dynamically typed semantics. This is a difficult requirement to satisfy, mainly because it is often impossible to statically resolve the actual functions that are called at every point in a program.

Early type error detection. In a statically typed program, type errors can be detected before running it. This is possible because given a program, it is possible to statically compute a type safety guarantee. The language implementation rejects any programs where this cannot be produced. In some languages, types are explicitly declared. In other languages, types can be determined by a decidable type inference process. In dynamically typed languages, the type of an expression or variable is resolved at runtime. A simple form of type checking is usually adopted in dynamically typed languages [39, 68, 2]. This is performed at runtime while the instruction is being executed. Such an approach can give no static type safety guarantees since a type error might be raised at any time and at any point during execution. In soft typing [22], type errors are not raised at an earlier point than traditional runtime type checking.

Easy to understand type error messages. The error messages from the type errors need to explain in relevant detail the reason why a type error can or will manifest itself. This information should reduce the debugging effort, and not increase it [36]:

> "Type errors in Soft Scheme are *pure torture* ... explaining type errors in Soft Scheme remains for PhD-level experts."

Ideally, the type error messages must contain the same dynamic information as type errors in dynamically typed languages, perhaps augmented with extra information.

Detecting all potential type errors. Proponents of unit testing and dynamically typed languages [33] argue that exercising a program written in a dynamically typed language using unit testing is enough to find most type errors. In reality, testing only reveals a minority of potential type errors [34]. Testing programs that run for a long period of time, such as web or phone applications, naturally also takes a long time. Testing is also laborious: one needs to write mock objects and test cases, and also needs to execute these in a realistic environment. On some systems, tests can take hours to execute. Programmers are naturally tempted to circumvent these tests during their development cycle and writing these gets increasingly more complex the larger the system one is testing.

The fact that a modest number of type errors cannot be detected statically should not preclude the use of type inference to detect type errors in dynamically typed programs. Indeed, a human reviewer can find a number of security and functional bugs by simply reviewing a piece of code [65]. In soft typing [22], a number of suspect program points can also be flagged in advance using a process of circular unification. The *minimal text principle* also originates from this

work [22]. This principle states that "the type system should accept *unannotated* dynamically typed programs. Otherwise, the programming interface will be more cumbersome than that provided by a conventional dynamically typed programming language" [22]. We aim to respect this principle.

Dynamically typed semantics. Currently, type systems such as gradual typing [98] and other forms of typing [5, 38, 117] can statically flag some type errors but these rely on type information being explicitly present in the program. On the other hand previous work on static type inference for dynamically typed languages, such as Diamondback Ruby [41, 40], RPython [10], Strongtalk [20] or Soft Scheme [116] introduce restrictions to the language, and force a statically typed semantics on a subset of the original language. The example in Figure 1.1 modified as in Figure 1.2 is therefore not allowed to execute in these languages, despite the fact that the program will not raise type errors for some inputs.

Today's most popular dynamically typed programming languages are compiled to dynamically typed bytecode before being executed. The process of analysing or transforming programs written in a dynamically typed language is more problematic than in languages that are compiled. For instance, code can be dynamically generated, loaded from a network and sometimes imports can only be resolved at runtime. Therefore, retrieving the source code of the function that is to be type checked and the functions that it calls is not always possible. At runtime, we can retrieve the bytecode of any loaded non-primitive functions. For this reason we use a similar approach to RPython [10]. This approach requires us to base the analysis on *bytecode* rather than *source code*. Dynamically typed bytecode is different in nature from lambda calculus or similar languages. Therefore our formal analysis has to reflect this.

1.3 Research objectives

The goal of this work is to develop a type checking mechanism that *eagerly* tries to preempt any type errors in running programs. Under this checking mechanism, a program would compute the same result as a dynamically typed program if the program is well typed. Through a simple system of type checking assertions, if we can determine that the rest of the execution results in a type error, it is more sensible to halt the execution of the program and notify the user than to fail with a runtime exception later on. This can be achieved by strategically inserting explicit type check assertions in the running code.

More specifically, we have the following objectives:

1. We develop a small language called μPython, a dynamically typed language based on a subset of the Python language. This language compiles down to μPython bytecode instructions, which we also define.
2. We formalise a special type analysis for the μPython bytecode language, and prove that the information from our analysis is an overapproximation of the actual runtime types.

3. We formalise a checked μPython semantics, and formally prove that this preempts all type errors at a linearly optimal point.

4. We propose a type check insertion and program transformation process to implement preemptive type checking on μPython programs, running on an unmodified interpreter.

5. We implement this type checking and transformation mechanism for a subset of the original Python 3.3 language and evaluate it on some benchmarks from the computer language benchmarks game [4].

When formalising and implementing preemptive type checking we shall use the following criteria at every step in the process:

Full dynamicity: Can the type checker handle programs where a variable is assigned with a value of different types throughout different program locations?

Minimal text principle [22]: Does the type checker work for unannotated programs?

Rejecting correct programs: Does the type checker give out false positives?

Timing of type errors: Does the type checker catch type errors early?

Type error information: Are the error messages useful for debugging and is the right part of the code being blamed for the type error?

1.4 Overview of approach

Before defining the preemptive type checking mechanism, we need to define a language on which this will be working on. We define μPython, a dynamically typed core language modelled on Python. A key characteristic of μPython (see Chapter 3) is that the types of variables may change during execution. It is a bytecode based language with dynamically typed variables and dynamically bound functions. Although small, the language is still sufficiently expressive to require a rich static type analysis. Our type analysis is actually performed on the bytecode representation of μPython programs. It is however useful to define a source language and therefore we define both a source code syntax and a bytecode syntax. We also define a simple compiler to translate the source to bytecode. Most of our examples are written in the source language but we only define the semantics of the μPython bytecode.

An important part of our solution is the type inference algorithm. Unlike other type inference algorithms, our algorithm infers two kinds of types called present types and future use types (see Section 4.1) for all variables for all program execution points. These are reconstructed using a forward and a backwards analysis respectively. The present type for a variable indicates the type of the value that a variable has last been assigned while the future use type indicates the type that

it is expected to be used as. This information enables us to pinpoint locations where to insert type checking assertions. In order to make sure that the type inference rules are correct, we use formal techniques. For example, we prove that the information for present and future use types is sound. In particular, we show that the present type of any variable is an overapproximation of the actual runtime type. Since we calculate overapproximations of possible runtime types, we use union types to represent sets of types. We use the type information gained from this type inference process to define a runtime semantics for μPython that implements preemptive type checking. We further prove an optimality property that states that any type errors are preempted at least as early as the start of the branch on the current sequence of instructions.

We then describe an algorithm that transforms bytecode programs by inserting type checks and explicit type error exceptions in such a way that the transformed program implements the checked semantics. This transformed bytecode can be executed using the unchecked semantics and the inserted type checks and exceptions implement preemptive type checking for that program.

For the implementation, we make use of Python's reflective capability to analyse the program at runtime. Since we are implementing a just-in-time type analysis, we lose access to the syntax of the program and can only retrieve the bytecode.

There are also other reasons why we are modelling our type system on bytecode rather than concrete syntax:

- We can start our analysis at a later point during the execution, for example after initialisation, and so get a more accurate analysis.
- We can leverage the work done by the bytecode compiler such as lexical analysis and identify which variables are locals or globals.
- We do not need to implement features that are just "syntax sugar".

We also implement our type checking mechanism for a subset of the full Python language. Our type checking process is integrated with the runtime environment. Unlike most analysers, our type checker does not take a program's source code. Instead, our type analysis works directly on a live program and environment, introspecting and analysing the environment for the currently executing program. This program is created and initialised by the standard interpreter. Thus the type checking process is divided into three phases, as shown by Figure 1.4.

The first phase, or the initialisation phase, simply involves reading the source files, compiling to bytecode and executing the program until the type checker is called on a specific function, for example `main`. During this process the environment is initialised, classes and functions are created, and external modules are loaded. During this phase, the full power of the language can be used. In the case of Python, this includes metaclasses, functions such as `eval/exec` and also dynamic code loading. In terms of the example of Figure 1.1, this phase runs up to but not

including Line 21. At that point, the interpreter would have read the source file and compiled both `compute` and `main`. These functions would therefore be present in the environment.

Once the initialisation process has stabilised, the analysis and program transformation process can be much more accurate. The analysis process initially involves a control flow analysis. This can be any kind of control flow analysis as long as the static control flow analysis is an overapproximation of the actual control flow at runtime. There are several algorithms which do this, one of the best known being k-CFA [93, 94]. The analysis process is invoked by loading the preemptive type checking mechanism and invoking it in the code.

We also specialise the inserted type checks, depending on the call site of this function. For this reason, specialised versions of functions are generated with assertions inserted into them. Calls to the original functions are replaced with these specialised functions in the bytecode in the environment. Once this process is complete, the execution of the program is continued. This time however, the program is executing using a preemptive type checking system which guarantees that preemptive type errors are raised earlier. These type errors are also more informative than standard type errors. In some cases the specific function that is being checked, for example `main`, can fail with an error inserted at the first execution point. This happens if `main` is shown to raise a type error under all circumstances. At this point warnings can be issued to aid the programmer with the debugging process.

1.5 Original Contributions

Our main contribution here is the development of the concept of type error preemption for dynamically typed languages. Our type checking mechanism tries to preempt all type errors at the earliest possible point. This is a novel contribution, and a problem that we solve in our work. We also present a number of technological contributions that make this possible.

Analysis on bytecode rather than source code. Tools that work at a bytecode level tend to be more usable than those that work on source code, since they integrate better with the build process. Most type systems are however defined on source languages. In our case, we formalise our type system entirely at the bytecode level. This makes our formalism more useful when building a tool that implements our type checking mechanism.

Innovative implementation in Python. There are several innovations in our approach and implementation. Rather than proposing a type inference mechanism that tries to cope with the difficult programming styles employed in Python, we propose a mechanism that performs type inference at runtime. In general, this simplifies the analysis and increases its accuracy. This approach is also taken by Firefox [51] and arguably the .NET DLR. In our work, however, we do not make any changes to the implementation of Python but implement preemptive type checking as a third party library which can be deployed in existing Python installations. Our

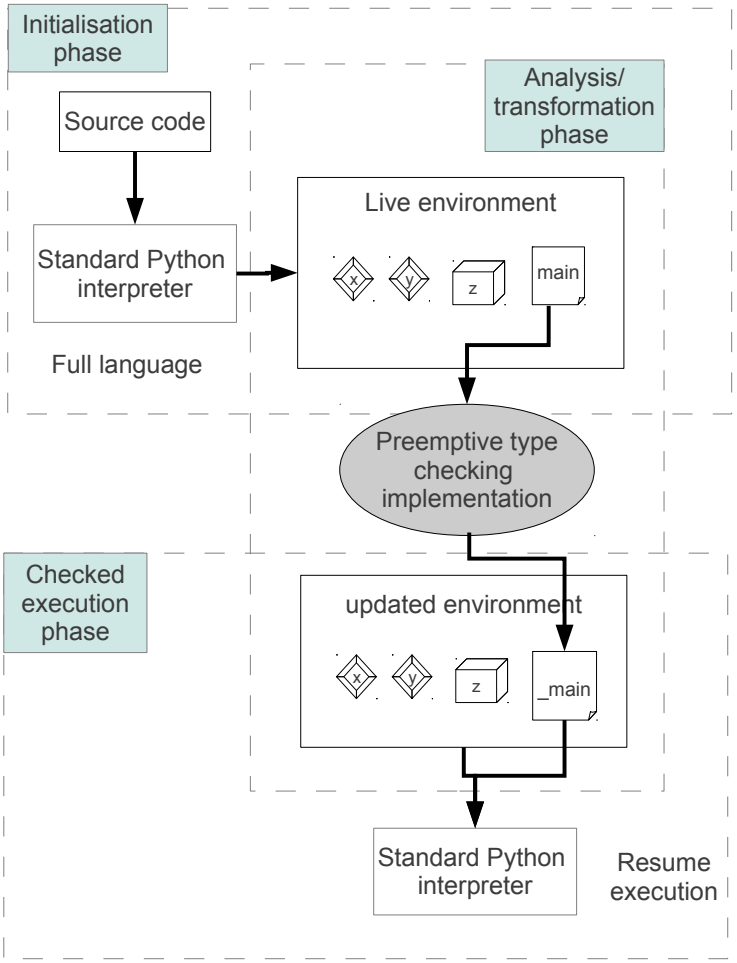

Figure 1.4: Phases of the type checking process.

implementation is able to substitute the bytecode of functions with specialised versions that have type checks inserted into them at initialisation time or runtime.

Helpful type error information. All potential type errors can be determined before execution of the transformed code. This information is available to the user as warnings and type error information. At runtime, the type errors are preempted much earlier than by using existing type checkers. If the statically generated constraints hold, then the program does not raise any type errors.

Present and future use types. Mainstream dynamically typed languages have side effects and a data flow analysis approach yields a more accurate type inference. Values are typically placed in memory before being used at a later point in the program. Our type system, which has present and future use types, distinguishes between variable assignments and operations consuming these variables.

Abstracting nodes in the control flow graph as truncated call stacks. When building our control flow graph of the program, we refer to our nodes as *execution points*. These nodes do

not simply refer to actual locations in the program but are actual call stacks, truncated to a finite depth.

Nonrestrictive type checking. A common way to type check dynamically typed programming languages is to introduce restrictions on the language. In the process, the original language loses its dynamicity. Our type system does not enforce a statically typed semantics on a dynamically typed program. This sets it apart from systems such as DRuby [41] and RPython [10], as the expressiveness of the dynamically typed language is preserved.

1.6 Outline

We present the background on type theory, program analysis and other techniques together with related work in Chapter 2. Chapters 3, 4 and 5 contain the theoretical work of this report. In Chapter 3, we formally define a small dynamically typed language μPython, upon which we base our type inference research. This involves defining source and bytecode grammars, an interpreter and a compiler. In Chapter 4 we formalise our type system and original type inference mechanism while in chapter 5 we propose an alternative, preemptively type checked semantics to the μPython language and also show how this semantics can be mapped back to the original semantics.

In Chapter 6, we implement a type inference for a subset of the full Python language, based on the techniques described in the preceding Chapters. Here, we present a number of optimisations and methods that allow us to adequately implement our tool. We also evaluate the tool on a number of synthetic and also real world benchmarks. We demonstrate that we achieved the objectives outlined in Section 1.3.

Finally, in Chapter 7 we conclude this thesis and we propose further research and work that can be carried out.

Chapter 2

Background and literature review

This chapter gives an overview of the topics that inform this work on preemptive type checking. We start in Section 2.1 by looking at type theory and type systems. This section introduces the different kinds of typing strategies in programming languages and compares and contrasts dynamically typed and statically typed languages. We also look at compilation techniques for dynamically typed languages (see Section 2.2). The remaining sections are dedicated to related or competing concepts and strategies in type checking. Approaches that include combinations of static and dynamic typing are described in Section 2.3. Approaches where the type checking mechanism is soft or optional are described in Section 2.4. Other approaches to type checking utilise type inference and involve defining a statically typed subset of a dynamically typed language. These are described in Section 2.5. Finally, since our approach depends on accurate control flow analysis, we review relevant approaches in Section 2.6.

2.1 Types

In set theoretical terms, a type can be described as a set of values of a particular kind, for example, the set of all natural numbers. Types are also useful abstractions of terms written in a programming language. In this section we introduce type theory concepts required to understand the rest of the thesis. Our presentation is based on [81].

Within the realm of software engineering, formal methods help us to ensure that a system behaves according to some set of rules and specifications. A type system is a formal method which is an integral part of a programming language. Described as a "tractable syntactic method for proving the absence of certain program behaviours" [81], types work by "classifying phrases according to the kinds of values they compute" [81].

Initially, type systems were introduced by Bertrand Russell [89] in the beginning of the 20th century to avoid paradoxes in logic such as Russell's paradox. Even though it was not their

original intended use, they have become indispensable in the design of programming languages. Type systems are ideal to detect bad behaviours of a program.

2.1.1 Simply typed lambda calculus

The lambda calculus is a formal system for function definition, function application and recursion. It is a small functional language, which is a very important part of modern type theory, and therefore relevant to this thesis. The simply typed lambda calculus, introduced by Alonzo Church [24], is a typed version of the lambda calculus. A common syntax of the simply typed lambda calculus is as follows:

$$e ::= c \mid (\lambda x : \tau . e) \mid (e\ e) \mid x$$
$$\tau ::= T \mid \tau \to \tau$$

In the above definitions, we may assume that c is a primitive constant expression such as '34', and that x is a variable name. An expression e can be a constant value. Expressions can also take the form $\lambda x : \tau . e$ where τ is a type such as Int. Expressions of these kind are called lambda abstractions. When such an expression is applied, the expression e is evaluated according to the evaluation strategy for the language. The syntax of function applications is $(e\ e)$. The first sub-expression in the structure is the function that is being called and the second sub-expression is the argument to that function application.

Free vs. bound variables. In the lambda calculus, variables can be either free variables or bound variables. For instance x is a bound variable in the term $M = \lambda x.T$, and a free variable of T. We say x is bound in M and free in T. If T contains a subterm $\lambda x.U$ then x is rebound in this term. This nested, inner binding of x is said to *shadow* the outer binding. Occurrences of x in U are free occurrences of the new x. Bound variables can be substituted by a non-captive fresh variable name to result in an equivalent expression. For example, the lambda expressions $\lambda x : \tau . x$ and $\lambda y : \tau . y$ are equivalent.

Type inference. Figure 2.1 contains the typing rules for the simply typed lambda calculus. Typing rules are written as one or more premises at the top and one conclusion at the bottom. $\Gamma \vdash x : \tau$ is read as "x has type τ under the type environment Γ". The type environment Γ is a mapping from variables to types. The semicolon operator adds a new binding to the environment. For example, in the T-ABS rule for typing abstractions, the premise adds one more assumption $(x : \tau_1)$ to the environment.

Types can be automatically reconstructed or inferred by the Hindley-Milner type inference algorithm, which is designed to work with the Curry-style simply typed lambda calculus. Thanks to this algorithm, one does not need to manually annotate every lambda expression with a type annotation. A comprehensive description of this algorithm is given by Damas and Milner in

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \text{T-VAR} \qquad \frac{\Gamma; x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}\ \text{T-ABS} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2\ : \tau_2}\ \text{T-APP}$$

Figure 2.1: Typing rules for the simply typed lambda calculus

$$\begin{aligned}
[\![\Gamma \vdash x : \tau]\!] &= \Gamma(x) = \tau \\
[\![\Gamma \vdash (\lambda x.e) : \tau]\!] &= \exists \alpha_1 \alpha_2.([\![\Gamma; x : \alpha_1 \vdash e : \alpha_2]\!] \wedge \alpha_1 \to \alpha_2 = \tau) \\
[\![\Gamma \vdash (e_1\ e_2) : \tau]\!] &= \exists \alpha.([\![\Gamma \vdash e_1 : \alpha \to \tau]\!] \wedge [\![\Gamma \vdash e_2 : \alpha]\!])
\end{aligned}$$

Figure 2.2: Constraints introduced for the simply typed lambda calculus.

[28], which is also extended to handle polymorphic types. A more contemporary explanation of these algorithms is available in [82]. Although such an inference algorithm is not easily applicable to dynamically typed programming languages, it is still relevant today in the context of functional languages such as ML and Haskell. Type inference generally consists of a process of constraint generation followed by unification, a process which unifies type variables with actual types. The constraints in Figure 2.2 are introduced for the simply typed lambda calculus. By recursively applying these rules to a term, we introduce a number of fresh variables, together with some constraints. These are then solved through unification. A process that is the reverse of type inference is type erasure. In this process, the explicit type annotations are removed from a program, before it is executed.

2.1.2 Subtyping

In order to support the difficult programming styles employed in dynamically typed programs, we need a structure that enables multiple types to be assigned to the same variable. This leads us (in Chapter 4) to define a type system with union types and a partial ordering. Subtyping, also referred to as subtype polymorphism, is a means of relaxing the requirements of type matching in languages such as the simply typed lambda calculus. Subtyping is found in object-oriented languages as a means to implement inheritance. A complete explanation of subtyping is available in [81].

A type system that supports subtyping has a subsumption relation, denoted for example by $S <: T$. This can be read as "T subsumes S" [81], or "S is a subtype of T". A typical example would be $\mathbb{N} <: \mathbb{Z}$. The subtype relation $<:$ is both reflexive and transitive. In subtyping, there are two special types that can be introduced, $\bot$ and $\top$. The $\bot$ type is the most specific type in a type system. Every type subsumes $\bot$ and there are no values of type $\bot$. On the other hand, $\top$ is the most generic type - it subsumes any type.

Over-approximation. Over-approximation of a type occurs when the type of a value, contained in a variable at a specific location given by the type inference, is less accurate, i.e., is a supertype, than the actual runtime type. If the type of a value in a variable x at a location n is inferred as τ_i and the actual type at runtime is τ_r, if $\tau_i :> \tau_r$ and $\tau_i \neq \tau_r$, then we can say that the type inference is over-approximating the type of variable x at location n.

Under-approximation. Under-approximation of a type occurs when the set of possible types of the value contained within a variable at a specific location during runtime is not a subset or equal to the set of types detected by type inference. Assuming that the type of a value in variable x at a location n is inferred as τ_i and that the actual type at runtime is τ_r, if $\tau_i :> \tau_r$ does not hold, then we can say that the type inference is under-approximating the type of variable x at location n.

2.1.3 Kinds of type systems

In this section we present a taxonomy of typing strategies. There are three main differentiating factors. We tend to describe a programming language according to whether it is dynamically typed or statically typed. A language with any typing strategy can be either strongly or weakly typed. If a language is statically typed, it can be either implicitly or explicitly typed.

Implicitly vs. explicitly typed. Whether a programming language has type annotations or not does not solely depend on whether the language is dynamically typed or statically typed. Statically typed languages may be either *implicitly typed* or *explicitly typed*. In explicitly typed languages, for example Java, any declaration of a variable, field or method has to be explicitly declared and its type has to be explicitly written in the declaration. For example, the types of arguments to a method and its return type are explicitly declared. In implicitly typed languages, these can often be left out and a type inference infers these from the usage of these variables and functions using a technique called unification. For example, in Haskell, a function that returns the factorial of a number can be implicitly typed:

```
fac 0 = 1
fac n = n * fac (n-1)
```

or explicitly typed:

```
fac :: Num a => a -> a
fac 0 = 1
fac n = n * fac (n-1)
```

Liskov and Zilles [64] characterise a language as being either strongly typed or weakly typed. A strongly typed programming language prevents functions being applied on data that does not match the explicitly or implicitly declared type of the argument. Cardelli [21], on the other hand classifies type systems as being either *safe* or *unsafe*. Both typed and untyped languages can be either safe or unsafe languages.

	Typed	Untyped
Safe	Haskell	Lisp
Unsafe	C	Assembler

Figure 2.3: Classifications of some common programming languages [21]

A number of statically typed languages, for example C, allow the user to cast from any type to another without actually checking that the coercion preserves type safety. We therefore consider the type system of C to be unsafe. Most dynamically typed languages are safe. Some examples are Python, Ruby and Lisp/Scheme. Languages that have safe typing do not allow the user to subvert the type system, by placing runtime checks or disallowing typecasts altogether

Statically vs. dynamically typed languages

In practice, mainstream languages tend to have elements from both kinds of type disciplines. Our understanding of what constitutes a statically typed language is one where it is possible to overapproximate most of the types of any variables/expressions cheaply and accurately. This makes it possible to type check a program statically. In dynamically typed languages, most of the type checking is performed at runtime. In this section we explore the advantages and disadvantages of both types of languages and also some of their differences. A comprehensive review of dynamically typed languages is given in [110].

Statically typed languages

Statically typed languages have a number of advantages, namely:

- Type annotations add more information to the program and serve as documentation.
- Most of the type checking does not need to be performed at runtime, and therefore the program runs faster. Optimisations can be performed and runtime dispatching over the types of values can be avoided since assumptions can be made about the types of the program's variables [22].
- Type errors can be detected at compile time, preventing runtime errors [22]. This reduces the testing effort.

Obtaining the types of variables and functions statically is straightforward. This information is either explicitly annotated or can be inferred using an algorithm, typically a variant of the Hindley-Milner algorithm. A type safety proof can be generated for a statically typed program, but typically cannot be generated for a dynamically typed one.

The expressiveness of the type system is sometimes limited. For example, statically typed languages allow programs that divide an integer number by zero to run. In an idealised language, the divisor would be a non-zero integer. Most languages do not offer this degree of flexibility in their type system, and have to resort to runtime checks. Most type systems are therefore under-constrained [110]. On the other hand, a static type checker might reject a program that works perfectly in a dynamically typed language. It is possible to write a program that might work with a less constrained type. For example, a `flatten` function which takes an iterable that can contain other iterables and returns a single flat iterable can be expressed in Python as follows:

```
def flatten(l):
    for el in l:
        if (isinstance(el, collections.Iterable) and
            not isinstance(el, basestring)):
            for sub in flatten(el):
                yield sub
        else:
            yield el
```

This function would however not pass a static type checker.

Rejecting programs that may not raise a type error limits the expressiveness of the language [22]. This is because the type checker conservatively errs on the side of safety. In many cases, this also limits generality [22] and hence reuse. This is one of the reasons why dynamically typed languages are more productive [83, 26]. For example, the Pascal type checker makes it difficult to write a sort procedure that can work with arrays of different lengths [22]. Prechelt [83] also comes to the conclusion that programs written in statically typed languages tend to require more effort to write for the same sets of requirements.

Due to a compilation step, the turnaround time, i.e., the time to build and test a program, is typically higher than in dynamically typed languages. This is especially true in large projects, where code needs to be compiled before the program can start running.

Even though it is easy to write an interpreter for a statically typed language, most languages are typically implemented as a compiler. It would not be sensible to design a statically typed language and relinquish the advantages of static typing. A number of language features are however more difficult to implement. For example, it is hard to support metaprogramming features since the program's structure is lost at runtime. Also, if constraints are not placed on the expressiveness of the metaprogramming features, type checking the code becomes undecidable [104]. It is however possible to support restricted metaprogramming in a statically typed language, and some examples include MetaML [104], Jumbo [58] and Meta-AspectJ [119].

Dynamically typed languages

Dynamically typed languages have a number of advantages over the statically typed languages:

- Lack of type annotations makes the syntax simpler, making the language easier to learn.
- Implementations and programmer tools such as debuggers and profilers are easier to write.
- Dynamically typed languages support higher-level language constructs such as metaprogramming and reflection.

Features such as macros (Lisp [69]), metaprogramming (Lisp [69]), reflection, continuations (Scheme [103]) and garbage collection (Lisp [69], Smalltalk [46]) were originally introduced in dynamically typed languages. Features such as first class functions are still not supported in some mainstream statically typed languages. Meanwhile, some mainstream dynamically typed languages such as Stackless Python [107] support continuations. These features are easier to implement if the programming language is interpreted rather than compiled.

Even though most dynamically typed languages are strongly typed (and also *safe*), most of them do not offer any form of static type checking. There are, however, some language extensions that are able to do more stringent type checking, at an early stage during execution. Some of them, such as traits [75] for Python, require the programmer to manually insert type annotations. Programming languages are increasingly allowing for both dynamically typed and statically typed programming features. Gradual typing [96] could be adopted to allow a fine grained level of detail to the programmer.

Flexibility of the type system is not an issue in dynamically typed languages, since type checking is done at runtime [110]. If a particular operation is supported on the runtime type of the value being used by the operation, then the operation succeeds. This also means that in order to discover all possible type errors, one has to exhaustively test the program, which is typically intractable.

2.1.4 Fixed Points

A number of predicates, sets and algorithms in this thesis require knowledge of fixed points, and therefore we explain these concepts at an early stage. In doing so, we use the same presentation as [77].

Consider a monotone function $f : L \to L$ on a complete lattice $L = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$. A *fixed point* of f is an element $l \in L$ such that $f(l) = l$ and we write

$$Fix(f) = \{l \mid f(l) = l\}$$

for the set of fixed points. The function f is *reductive at* l if and only if $f(l) \sqsubseteq l$ and we write

$$Red(f) = \{l \mid f(l) \sqsubseteq l\}$$

for the set of elements upon which f is reductive; we say that f itself is *reductive* if $Red(f) = L$. Similarly, the function f is *extensive at* l if and only if $f(l) \sqsupseteq l$ and we write

$$Ext(f) = \{l \mid f(l) \sqsupseteq l\}$$

for the set of elements upon which f is extensive; we say that f itself is *extensive* if $Ext(f) = L$.

Since L is a complete lattice, it is always the case that the set $Fix(f)$ has a greatest lower bound in L and we denote it by $lfp(f)$:

$$lfp(f) = \bigsqcap Fix(f)$$

Similarly, the set $Fix(f)$ has a least upper bound in L and we denote it by $gfp(f)$:

$$gfp(f) = \bigsqcup Fix(f)$$

Using the *Knaster-Tarski Fixed Point Theorem* [29], we can show that $lfp(f)$ is the *least fixed point* of f and that $gfp(f)$ is the *greatest fixed point* of f.

2.2 Compiling dynamically typed programs

Dynamically typed language implementations have traditionally been slow [4]. Recent advancements have helped in raising the performance levels of these languages. In this section we explore the latest techniques in Just-in-time (JIT) compilation of dynamically typed languages. This section is especially relevant as our type checking mechanism is implemented using some techniques that are used when implementing JIT compilers. Recently, a lot of work has specialised in JavaScript, and information gathered from type inference is used to optimise the execution of these languages. For instance, an interesting approach adopted in the Firefox JavaScript implementation [51] involves performing a fast but unsound type inference process. The type information is then used to compile specialised machine code versions of code snippets and functions.

2.2.1 Partial evaluation

Traditionally, compiler-compilers relied on the user to describe the syntax and semantics of a programming language in a domain-specific language. Unfortunately, this relies on the user to make a clear distinction between what happens during compile time and during runtime [35]. This is especially difficult in the case of elaborate languages with dynamic features. Partial evaluation is a technique that can be used to generate compilers for dynamically typed languages.

Partial evaluation is a program transformation technique for specialising a program with part of the input, thus giving a faster program for the rest of the inputs. A tutorial on partial evaluation

is given in [25]. A program can be seen as a function of input to output data:

$$prog : I_{static} \times I_{dynamic} \rightarrow O$$

where I_{static} is input to the program that can be determined statically and $I_{dynamic}$ is input that is dynamically determined. The partial evaluator is another program. It takes as input the program and static inputs to specialise the program with. It returns a specialised program (called *residual program*), which takes only the dynamic inputs. The partial evaluation process is represented as follows:

$$(prog, I_{static}) \rightarrow (prog* : I_{dynamic} \rightarrow O)$$

An original approach to build compilers from interpreters by Futamura [42] uses partial evaluation.[1] The technique is known as the Futamura projection, which can be described in three stages:

1. Partially evaluating (specialising) an interpreter with a given source, generating an executable.
2. Partially evaluating (specialising) the interpreter as applied in (1), generating a compiler.
3. Partially evaluating the partial evaluator used in (2), generating a compiler-compiler that given an interpreter returns a compiler.

The transformation process implemented in preemptive type checking involves inserting assertions in a function that are specialised according to a given call site. This resembles partial evaluation since we can statically determine type information. This information is not as accurate as the runtime type information. Hence, the static input to our inserted type checks is the inferred type information and the dynamic input is the actual runtime environment.

2.2.2 Tracing JIT compilation

Traditionally, programs are either interpreted or compiled. Just-in-time (JIT) compilation is a hybrid approach, in which the program is incrementally compiled while being executed. The compiled versions are also cached to improve performance. A considerable number of people have been working on tracing JIT compilers for the last ten years. These kind of JIT compilers record a linear sequence of frequently executed operations, and compile optimised versions of these to machine instructions.

[1] The referenced paper is a re-published paper. The original was published in Japanese during the 60s.

A *trace* is a path through the control flow graph of a program. Jumps are expensive operations that disrupt the efficiency gained by pipelining on superscalar CPUs. Tracing based methods are designed to avoid unnecessary jumps. If a *dominant* trace within the control flow graph (CFG) of a loop can be established then all effort should be invested in optimising that dominant trace. Dynamo [13] is the original tracing JIT compiler. The work was not originally meant to speed up dynamic languages but instead it presents a technique to re-structure compiled programs in such a way that *dominant* traces can be executed without jumps. In a dominant trace, code is inlined within traces and a trace is simply a sequential stream of instructions with some side-exits.

Trace-based JIT compilers yield especially good results when applied to dynamically typed-languages [44, 43]. The stream of consecutive instructions in a trace can be transformed to TSSA (*trace static single assignment*) and optimised aggressively. The Tracemonkey JavaScript engine [43] used in Firefox makes use of this technology. Further techniques have been implemented to enhance the performance of Tracemonkey, namely trace stitching, blacklisting of traces that often revert to the interpreter, nested traces and calling external functions.

Tracing JIT compilers switch between interpretation, compilation and execution of compiled traces during runtime. Because of this, program execution jitters when a tracing JIT compiler pauses to compile a trace. Ha et al. [50] implement a concurrent tracing JIT compiler for JavaScript. While the program is interpreted on one thread, another thread compiles parts of it. When a trace is compiled, the compiled version is executed. In such an implementation, the challenge is seamlessly transferring the control from the interpreter to the compiled code. The advantage of a multithreaded JIT compiler is that more cores are utilised and that the program can execute without pausing between compilation and interpretation.

Our implementation of preemptive type checking as described in Chapter 6 statically analyses the structure of a running program at runtime. Therefore, our implementation performs type inference and program transformation at runtime in a similar way to a traditional JIT compiler.

2.3 Gradual type systems

In this section we discuss combinations of static and dynamic typing disciplines that enable statically typed code to interact with dynamically typed code and vice versa. We summarise the presented techniques and compare these with each other and also with our work. These type systems are often referred to as *gradual type systems*. Several languages in use today can be considered gradually typed languages. These include Boo [31], TypeScript [72] and even Scala [79], C# [15] and Java [88].

2.3.1 Dynamic types

The initial work combining static and dynamic typing focused on increasing the degree of dynamic typing in statically typed languages; for example, Abadi et al. [5] introduced a dynamic type Dyn to model finite disjoint unions or subclassing in object-oriented languages. They argue [5] that finite disjoint unions (C, Algol68) or tagged variant records (Pascal) are a finite version of Dyn. A typical object-oriented language has subclasses and these can be thought of as infinite disjoint unions and are equivalent to Dyn. Abadi et al. shed light on the uses of the dynamic types at that time, e.g., inter-process communication. In the type system proposed by Abadi et al. [5], an explicit injection construct (`dynamic`) is used when a statically typed value is used within a dynamically typed expression. This casts a value to the type Dyn by adding its type-code at runtime. An explicit projection construct (`typecase`) is also available. The use of dynamic types is therefore constrained, i.e., values of type Dyn can only be used in a `typecase`-construct. This is similar to a switch statement that given an argument, dispatches on the types given to the construct. Dynamic types also appear in quasi-static typing [105]. The outcome from a quasi-static type checking process indicates in some cases whether a programme is guaranteed to raise a type error, or guaranteed not to. In other cases however, the outcome would be ambivalent, which means that the program might raise a type error.

The type Dyn also appears in gradual typing [96, 97, 98], which also allows dynamically typed and statically typed code to commingle. The authors suggest that dynamically typed programs are statically typed programs where the static type of any term is of type Dyn. As opposed to the type system by Abadi et al. [5], injection and projection are automated in gradual typing [98]. In gradual typing the notion of type consistency is introduced. Type consistency, denoted by $\backsim$, is a reflective, symmetric but non-transitive relation on types. For example, $\mathsf{Int} \backsim \mathsf{Int}$ and $\mathsf{Int} \not\backsim \mathsf{Str}$. However, the Dyn type is statically consistent with any possible type. For example, $\mathsf{Int} \backsim \mathsf{Dyn}$ and $\mathsf{Int} \to \mathsf{Dyn} \backsim \mathsf{Int} \to \mathsf{Int}$. Therefore, anything can be implicitly cast to Dyn and Dyn can be implicitly cast to any other type [96, 97, 98]. This process does not use subtyping, which contrasts with the approach used in quasi-static typing [105]. In the latter, the type Dyn sits on both the top and the bottom of a subtype lattice. Since the subtyping relation is transitive, the lattice collapses to one point and every type is a subtype of every other type [98]. The type system therefore does not reject any program. Therefore in gradual typing, the notion of a non-transitive type consistency relation is a key improvement compared to previous type systems.

Inspired by gradual and soft typing, `like` typing [117] is yet another way to integrate dynamic and static typing. In `like` typing, values can be of type Dyn and can also be of any static type `C`. Any operation on objects of type Dyn are type checked at runtime. The `like` typing system however is different from gradual typing because it has intermediary types between Dyn and the static types. A variable can be declared as `like C`, where `C` is a static concrete type. Variables declared with this type are checked statically within their scope. Runtime type checking can also occur as these variables may be bound to values of type Dyn. Therefore, uses of variables

of type `like C` are checked statically, but whenever another variable of any type is assigned to a variable of type `like C`, the conformance to `C`'s interface is checked dynamically. Therefore, if a variable `p` is declared to be `like C`, the type checker statically checks that all operations on variable `p`, such as method invocations on `p`, conform to the interface of `C`. For example, if `C` has a method `foo` but not a method `bar`, then `p.foo()` is valid and `p.bar()` is statically rejected. Declaring `p` to be of type `like C` is a static guarantee that it will be used as a `C` so instead of checking at runtime, one can simply do a static check [117]. Assignments to `p` however require a runtime check.

2.3.2 Usage and evaluation

The use of dynamic types in traditionally statically typed languages has seen an increase. For example Boo [31], TypeScript [72], Scala [79], C# 4.0 [15] and Java 7 [88] are modern programming languages that have recently gained momentum and support the inclusion of dynamic types. Bierman et al. [15] describe a type checking process for a variant of C# with type Dyn called $FC^{\#}_{4}$. This language is formalised, and a conversion process from this language to another C# language variant ($C^{\#}_{CLR}$) is presented. In the latter, type information has to be made more explicit than in the former and the Dyn type is translated to `object`. Explicit conversions are used to turn an `object` to any other type. The translation process itself is type-correct, i.e., any resulting $C^{\#}_{CLR}$ code is well typed.

In general, the advantage of gradual typing is the flexibility offered to the programmers in providing them the choice of either static or dynamic typing. This choice is available for each term in a program. Type safety is always preserved – this can be guaranteed statically in annotated code and during runtime in unannotated code. Gradual typing is typically implemented by wrapping a value with a dynamic type whenever it is implicitly cast to a particular interface. This verifies that any subsequent operations on this value respect the target type's contract. A Python implementation of gradual typing is available [111], however the implementation seems to only type check explicitly annotated arguments of functions at runtime.

It has been noted [117] that even the presence of a single wrapper for any value is likely to slow down the execution of a program. Values need to remain wrapped until these are garbage collected. This is because any side-effects can violate the wrapped value's contract [117]. Therefore, any operation on this value can fail at runtime. Wrappers also have to be manipulated at runtime, thus preventing any compiler optimisations as the compiler has to emit code that assumes the presence of wrappers everywhere [117].

Gradual typing, `like` typing and preemptive type checking all aim to find type errors earlier than dynamic typing. The problem of `like` typing is that it is hard for a programmer to learn the intricacies of the type system. As in gradual typing, the programmer has to learn the differences between dynamic and static types. However, yet another level of type complexity is added in `like` typing. As in gradual typing, `like` typing requires that the programmer annotates at least

part of the program to benefit from it. An advantage of `like` typing over gradual typing is that if a variable is declared as `like C` rather than Dyn, and if this variable is assigned with a value that does not respect `C`'s interface then the error is raised during the assignment rather than when the variable is used. In preemptive type checking, this typically happens as well. Furthermore, in preemptive type checking type errors can be found at the earliest state during the program execution, given the limitations of the flow sensitive analysis.

No type inference algorithms have been proposed for `like` typing. However, there are type inferences for gradual typing [98, 84]. Unification based inference [98] can infer static types for parts of the program that are statically typed. When unification fails, a dynamic type is assigned to expressions. Unification is less suitable for use in an object-oriented language, and Rastogi et al. [84] propose an alternative, and implement this for ActionScript.

The concept of blame was introduced in Henglein's [54] work, where the mechanisms involved in coercing dynamic types to static types and vice versa were formalised. Eiffel [71] provides both statically checked types and dynamically checked assertions called *contracts*. The notion of contracts reappears in Findler and Felleisen's work [37]. In this case contracts serve as assertions for higher-order functions. The term *blame* is used in this work, which refers to the origin of a contract violation. In the case of gradual typing, a contract violation is a type error. Tobin-Hochstadt and Felleisen introduce *migratory types* [108]. This work presents a method whereby a program written in an untyped language can be gradually migrated to a typed version of the same language. This happens by gradually annotating certain modules. Constraints are inferred from these annotations, which are transformed into contracts and finally the execution of the program can assign proper blame in case of a type error. Similarly, Flanagan [38] introduces *hybrid types*. The blame calculus has been subsequently refined [32, 99, 7] and it has also been shown that statically annotated code cannot be blamed for runtime type errors [112].

None of the type systems discussed in this section can be described as flow sensitive, hence these do not take into consideration the control flow of a program. Analysing the control flow of a program [41, 94, 49] aids in getting a much more accurate type reconstruction. Also, the effectiveness of these systems to flag type errors typically relies primarily on manually inserted type annotations. There are type systems such as these that come with a type inference [98, 84], however there are no implementations of these at present.

2.4 Soft typing

In this section we mainly describe soft type systems, a generalisation of static and dynamic typing. Soft typing can be applied to dynamically typed languages and has little effect on the runtime semantics of these languages in programs that do not raise type errors.

Soft typing [22] is an early attempt in reconciling both typing disciplines. As in other type systems at the time [105, 5], it is described as a "generalisation of static and dynamic typing ...

that combines the best features of both approaches" [22]. In essence, this is a type system where the type checker implicitly converts a dynamically typed program into one that can be statically typed. It does this by inserting explicit runtime type checks (also called narrowing functions) around the arguments of primitive operations [22]. Since the inserted type checks are explicit, a programmer can review these to ascertain that they hold during the execution of the program. Since the dynamically typed program has been effectively transformed into a statically typed program, compilers can then generate more efficient code of softly typed programs.

Soft typing does not take into consideration assignments [22], however there is other work [52] that can be leveraged to add support for assignments. There are two guiding principles in soft typing [22]:

Minimal Text Principle: The program accepted by the system should be unannotated and dynamically typed. Otherwise, the program would be more verbose and cumbersome.

Minimal Failure Principle: The type system should be rich enough so that "typical" programs can be statically type checked. Otherwise, if unnecessary runtime checks are introduced, programmers would be more inclined to ignore them as most of them would be "false positives".

The first problem encountered when assigning types is when *heterogeneous expressions* are encountered [22]. An example of such an expression is a ternary expression that can evaluate to values of different types based on its predicate. In order to unify these expressions, *union types* are introduced. For example, the union type of Int and Str is denoted as $\mathsf{Int} \sqcup \mathsf{Str}$. In soft typing, this is encoded in Remi's notation [85], which is an ingenious encoding of union types that allows soft typing to reuse the standard Hindley-Milner type inference algorithm. There are however shortcomings with this notation [54]. In soft typing there is no typing rule for induced containments of union types (e.g., $\upsilon <: \upsilon', \nu <: \nu' \vdash \upsilon \sqcup \nu <: \upsilon' \sqcup \nu'$), and the subtype rule for recursive types is unsound [54]. The original work [22] focused on the functional subset of Scheme and did not handle assignments, so that there is no notion of preemption. The extended version of the practical soft type system for Scheme [116] handles assignments, but restricts all occurrences of the assigned variables to have the same type, which makes it impossible to successfully type check even the simple example from Figure 1.1.

In soft typing, the information gathered from the type inference is then used to insert *narrowing functions*, functions that explicitly cast values from one type to another and raise an error if this is not possible. Soft typing was extended with conditional types [8] in order to solve some of the shortcomings of the original soft typing. Conditional types are types that depend on certain predicates to be applicable and are introduced at control flow splits. The result is that the constraints introduced by the type inference can be analysed and solved and therefore fewer type checks need to be inserted in the generated code. Soft typing has also been applied to Python [90] and Erlang [78, 66]. Nystrom [78] also bases his type system on a data flow analysis but

does not distinguish between the notions of *present* and *future use*, introduced in preemptive type checking (see Section 4.1). The success of these approaches has varied. Soft typing enables faster program execution due to a reduction of runtime checks and the opportunity to use the type information gathered as part of the analysis for more efficient compilation. Warnings are issued in advance, indicating where and how type errors might occur. Wright and Cartwright [116] developed a soft type system for R4RS, a modern scheme dialect. This type system is a soft typing system that extends the Hindley-Milner static type system with union types and recursive types. They claim that this typically eliminates 90% or runtime checks and consequently programs run 10 to 15% faster. Despite this, soft typing cannot give a guarantee that type errors will not occur. Also, any type error messages are of little use to the programmer [117, 36].

Soft typing has a reputation for being brittle [117] as a simple mistype in a method name will insert an explicit runtime check that will always fail. An error is then raised at the moment when this method is called. The types generated by a soft typing analysis are generally complex and of little use to the programmer or to an IDE. Such types are subsequently hidden from the programmer, making the model opaque [117]. Therefore, small changes to the code can have a large impact on the runtime performance [117].

A generalised and related concept is that of pluggable type systems. These type systems are "neither syntactically nor semantically required, and have no effect on the dynamic semantics of the language." [19]. The notion of pluggable type systems is summarised by Bracha [19]. Since preemptive type checking does not affect the semantics of μPython in runtime executions that terminate without raising type errors (Section 5.2) and no type annotations are required, we consider our type system to be a pluggable one. Soft typing can also be considered a pluggable type system [19].

There are similarities between soft typing and preemptive type checking on many levels. In both our work and in soft typing, type checks are inserted into the user's code. In soft typing these type checks are inserted in a function's argument, if needed. Just like dynamic typing, a type error is only raised at the last possible moment. In contrast, preemptive type checking is guaranteed to raise a type error at an earliest point. Instead of unification, we use the information from different flow sensitive type analysis to reconstruct an approximation of the runtime type of any variable at any point throughout a program's execution and what its value will be eventually used as at a future point.

2.5 Static type inference for dynamically typed languages

The techniques described in this section are commonly described by their authors to be applicable to dynamically typed languages. However, this is not really the case. Most of these techniques rely on static type inference and thus impose a statically typed semantics to the languages they are being applied to. Type inference is used to find type errors in dynamically typed languages and to apply optimisations. We summarise the presented techniques and compare

these with each other and also our work. An important issue we will be discussing is the set of restrictions that are placed on these languages to enable type inference.

MetaML An interesting observation we can make from the type systems presented in this section is that it is claimed that a reason why type inference is deemed to be tricky for dynamically typed languages is the fact that such languages often have unrestricted metaprogramming features. Type inference can be implemented in metaprogramming languages, as can be seen in languages such as MetaML [92, 104]. Metaprogramming is restricted in this language. For example, the object bracket and escape notation is used to generate code rather than strings. In addition, an important restriction is placed on code fragments, namely these should always be lambda abstractions. Thus the generated code snippets can be safely typed.

Python and Ruby A completely different approach to statically type check languages with metaprogramming features is presented in RPython [10], a statically typed subset of the Python language. All metaprogramming features (including `eval` and metaclasses) may be used during the initialisation of the Python classes. In languages such as Python or RPython, even determining which file is imported when an import statement is executed can be undecidable.

In RPython, metaprogramming features cannot be used during the running of the program. RPython also rejects programs where types cannot be statically resolved. It can therefore be compared with a statically typed version of Python. A similar attempt to give a statically typed semantics to a dynamically typed language is Diamondback Ruby (DRuby) [41]. DRuby also accepts type annotations, which help the type inference. Given its dynamic nature, DRuby can only give warnings about potential type errors. It does not catch all type errors and sometimes raises type errors for programs that work well. DRuby's static type system is elaborate and supports features such as union and intersection types, subtyping, object types, parametric polymorphism and mixins. The type inference algorithm is also flow aware.

Unlike RPython, DRuby does not support metaprogramming features such as eval. Furr et al. also developed PRuby [40], an extension to DRuby. PRuby tries to address some of the shortcomings of DRuby, related to metaprogramming. Determining the type of the result from functions such as `eval` is undecidable if `eval` accepts arbitrary strings. However, by profiling a running Ruby program, a sample of strings which are passed to `eval` and similar functions can be gathered. PRuby then transforms the program into one that does not make use of these features. The resulting transformed program is statically checked using DRuby. DRuby is also used in a statically typed implementation of Ruby on Rails (RoR) [9]. This works by transforming RoR applications into plain Ruby. The transformation avoids the use of metaprogramming features and the resulting application is then type checked using DRuby.

JavaScript. Features of JavaScript that make type inference difficult include the use of prototypes instead of classes, first class functions and weak, dynamic typing. Different type systems have been proposed for JavaScript [11, 106]. Anderson [11] proposes a structural type system [80] for a subset of the JavaScript language JS_0. This subset excludes prototypes and first-class functions. This type inference algorithm allows the dynamic addition of attributes to JavaScript

objects. However sophisticated, this type system cannot be applied to Python, as the class and object creation mechanism is much more dynamic than in JavaScript. Also, no consideration is made of control flow and state and therefore simple sorting functions from the Python standard library cannot be adequately type checked [49]. Thiemann [106] proposes a type system where a type is described by its base type and its features (such as members). Although a type inference mechanism is not proposed, an implementation is available. More recently, a semantics for the JavaScript language has been formalised [48], although no type inference mechanism has been proposed. Recency types [53] deal with ad hoc object initialisation patterns, i.e., objects can be created at one point and members assigned dynamically. In order to deal with this, only "contexts" of linear instruction sequences that are placed between special labels are considered. In these contexts, an object is instantiated and its fields are assigned. These labels that delineate the contexts are referred to as $\texttt{MASK}^k$ expressions. These labels are automatically placed by the system, although not enough detail is given on how the placing of these $\texttt{MASK}^k$ expressions is determined. The concept of a recency type is similar to *present types* in preemptive type checking. Present types are more sophisticated as these can change throughout linear interprocedural flows of execution, although preemptive type checking does not support objects yet. Similarly, Guha et al. introduce a type system where control flow and state is taken into consideration [49]. This enables typing of programs that make use of idioms [49] such as *heap-sensitive reasoning*, *dynamic dispatch* and *type tests*. The type system is modelled for a simple semantics for JavaScript [48]. Similar to our approach, the type system supports joins and ordering. The type environments are labelled, however these are simply program points and not abstractions of call stacks as in our approach. There is also no distinction between present and future use types. Another interesting approach for type checking JavaScript involves introducing dependent types [23]. In this approach an SMT solver is employed to check the type derivations, which are derived for all the values present in the program.

Scheme. Felleisen and Tobin-Hochstadt [109] propose the notion of *occurrence typing* for implementing a statically typed version of Scheme. A translation of the simple example in Figure 1.1 is statically rejected by this system. Bigloo [91] is another statically typed subset of the Scheme language that supports optional type annotations. These type annotations are written as assertion-style contracts which are constraints on procedures. These constraints are used to generate more efficient code when the compiler can prove they are correct. Similar to gradual typing, these are turned into runtime checks when the compiler cannot prove them correct.

Smalltalk. Strongtalk [20] is a statically typed subset of Smalltalk with features such as polymorphic signatures, protocol based inheritance, generics and parametric polymorphism. The language also supports the typecase construct, where runtime type checks are presumably carried out. This work does not however define a formal type system or describe how omitted type annotations are treated.

Erlang. Marlow and Wadler [66] propose a type system which supports recursive types and subtyping. Programs are not accepted if matching or case expressions are not exhaustive [78]. Therefore, only a subset of the language is supported.

SELF. Agesen [6] proposes a type inference mechanism for SELF. This inference mechanism works by generating constraints and unifying these constraints to obtain the desired type information.

2.6 Control flow analysis

Control flow analysis is generally the first step of any form of non-trivial program analysis. These include variable elimination and, more importantly, flow sensitive type inference, as in preemptive type checking. Control flow analysis is trivial in simple imperative languages but much harder in a higher-order languages, such as Scheme and Python. In these languages, functions are first class citizens. We see that the top five languages currently in use [3] allow functions to be passed around as arguments to functions. Indeed, most of the languages currently in use on production systems (C, C++, Java, C#, PHP, Ruby, JavaScript, etc...) are higher-order languages. It is also argued [73] that all object-oriented languages are implicitly higher-order, because method invocation is resolved dynamically – the invoked method depends on the type of the object that is present at the invocation point.

If function calls are dynamically bound at run time, statically determining which function is actually called is undecidable. Classic data-flow algorithms therefore cannot be used, because it is presumed that an interprocedural control flow graph is statically computable. An over-approximated CFG may still be computed through program analysis. This computation, however, requires type information in order to be precise. We therefore have a chicken and egg problem: inferring type information requires building a control-flow graph, and vice versa. In a language where functions are passed as arguments, such as in Scheme, the target of a function call may not be explicit, for example: `(lambda (f) (f x))`. Therefore, a control flow analysis must take into consideration the argument applied to f and also where it is invoked. Scheme also has control flow instructions such as `call/cc`, which make the control flow analysis harder.

Shivers' work [93, 94] mainly deals with intra-procedural analysis. His framework works on code that is translated to continuation passing style (CPS). This makes the structure of the code uniform and simplifies the implementation of `call/cc`. Various functional language implementations convert code into CPS at some intermediate stage.

There are various orders of control flow analysis discussed [93, 94], but generally the simpler the analysis, the faster but also the less accurate it is. The crucial step in these forms of analysis is that of determining an overapproximation of functions bound to any variable. The simplest analysis is the 0th-order control flow analysis or 0CFA. Suppose that throughout the execution of a program, a variable x is bound to n different values $v_{i=1}, v_{i=2}, ..., v_{i=n}$ in n different contexts. Then, in 0CFA, evaluation of x in an environmental context i results in the entire set $v_1, v_2, ..., v_n$ rather than one v_i. 0CFA analyses the pure control-flow structure; it can determine which functions are called from which call sites.

A more precise analysis distinguishes the dynamic frames allocated when a function is called from two distinct call sites. This is called 1st-order control flow analysis (1CFA). So, if a function is called from different call sites, its variables are bound in different frames. All values passed to the function from a given call site are merged, but values passed from different call sites remain distinct.

Shivers designed the control flow analysis for CPS Scheme, a language that he also defines. In order to find an exact control flow semantics, one can simply instrument the interpreter and record the control flow while a program is being interpreted. This, however, is unfeasible as programs may not terminate and several executions might produce different control flow graphs. Considering all possible executions and joining all control flow graphs from every interpretation is therefore in-feasible. This is not only because the external environments are generally uncountably infinite but also because a finite program can give rise to an unbounded number of distinct environments. Shivers therefore defines an abstract control flow semantics, which approximates the exact flow semantics and proves that this can be approximated.

Although some variants of k-CFA are intractable [95], this algorithm was adapted to other languages and representations [95] and also to other problems like CFA in OO-style programs [73]. Since our type inference depends on a control flow analysis, choosing the appropriate algorithm can make a difference to the effectiveness of preemptive type checking.

2.7 Summary

In this chapter we have described the relevant background material, which includes type systems and type theory. We explained the different kinds of typing strategies in programming languages. We compared and contrasted dynamically typed and statically typed languages. We also looked at compilation techniques for dynamically typed languages, especially in the context of JavaScript.

Finally, we explored related or competing concepts and strategies in type checking. These include combinations of static and dynamic typing, soft or optional typing, type inference and control flow analysis in higher order languages.

Chapter 3

The μPython language

The Python language has been in development for more than 20 years and over that period a great number of features and instructions were introduced. It is therefore useful to formalise a core calculus of this language before formalising the type checking mechanism itself. It is common practice [55, 102, 15] to simplify a real programming language or to define a small calculus based on a real programming language. This makes it possible to experiment with the language and perform rigorous proofs.

In this chapter we define μPython as a dynamically typed core language modelled on Python. It is a bytecode-based language with dynamically typed variables and dynamically bound functions. Although small, the language is still sufficiently expressive to require a rich static type analysis.

We present the high-level syntax of μPython in Section 3.1. We omit its formal operational semantics, as it is standard; also, our type analysis is exclusively performed at the bytecode level and the high-level syntax is used only for illustrative purposes. This language is compiled down to bytecode, which we define in Section 3.2. We also describe the compilation from the source language into bytecode and how the bytecode is interpreted (Section 3.3). We try to keep faithful to the spirit of the original Python language in source code, bytecode, compilation and interpretation. In Section 3.4 we show an example μPython program, together with its compilation and execution.

3.1 μPython source code

We define a syntax for our language in Figure 3.1. A line break or a semicolon is used to separate statements, while indentation delineates blocks. μPython supports function definitions, conditional statements, assignments and while loops. In μPython, expressions are either function calls, constants or identifiers. Valid expressions are also valid statements.

Statements:

$$
\begin{array}{rl}
s ::= & \mathsf{def}\ f(x) : s \qquad \text{(function definition)} \\
\mid & \mathsf{return}\ e \qquad \text{(function return)} \\
\mid & e \qquad \text{(expression)} \\
\mid & \mathsf{pass} \qquad \text{(empty statement)} \\
\mid & \mathsf{raise} \qquad \text{(exception)} \\
\mid & x\ =\ e \qquad \text{(assignment)} \\
\mid & \mathsf{if}\ e :\ s\ \mathsf{else} : s \qquad \text{(conditional)} \\
\mid & \mathsf{while}\ e : s \qquad \text{(loop)} \\
\mid & s; s \qquad \text{(sequence)}
\end{array}
$$

Expressions:

$$
\begin{array}{rl}
e ::= & x \qquad \text{(variable)} \\
\mid & c \qquad \text{(constant)} \\
\mid & e(e) \qquad \text{(function application)} \\
\mid & \mathsf{intOp}(e) \qquad \text{(prime integer function)} \\
\mid & \mathsf{strOp}(e) \qquad \text{(prime string function)} \\
\mid & \mathsf{isInst}(e, \tau) \qquad \text{(instance check)}
\end{array}
$$

Types:

$$\tau ::= \mathsf{Int} \mid \mathsf{Str} \mid \mathsf{Bool} \mid \mathsf{Un} \mid \mathsf{Fn}$$

Constants:

$$c ::= n \mid str \mid \mathsf{true} \mid \mathsf{false} \mid * \mid \mathsf{U}$$

Figure 3.1: Syntax of the μPython language

In μPython (as in Python), function definitions are simply assignments of anonymous functions to variable names. Functions can be reassigned at any point and within any control flow structure or scope. μPython supports higher order functions, where functions are first class citizens. For simplicity, functions can only take zero or one arguments; functions with more arguments must be curried. There are three built-in functions. isInst is a reflection operator to check the dynamic type of an expression, and always returns a boolean. intOp and strOp represent prime integer and string operations, which implicitly raise a type error if their argument is of the wrong type. Note that conditional statements and function calls will also implicitly raise a type error when their guard or function expressions do not evaluate to boolean or function types respectively. This contrasts with the raise operation that will immediately raise an *explicit* exception error to terminate execution.

We have a single namespace $\mathbb{V}$ that comprises both variable and function names and use the metavariables x, y (respectively f, g) to denote names that are intended to represent variables (respectively functions). In μPython, all variables have global scope. Figure 3.5 on Page 40 shows a sample μPython program.

instr	::=	LC c	(load constant)	|	intOp
	|	LG x	(load global)	|	strOp
	|	SG x	(store global)	|	isInst τ
	|	JP n	(unconditional jump)	|	raise
	|	JIF n	(jump if false)	|	
	|	CF f	(call function)	|	
	|	RET	(return from call)	|	

Figure 3.2: The μPython bytecodes

3.2 μPython bytecode

Our type analysis is actually performed on the bytecode representation of μPython programs. Since we are doing a just-in-time type analysis, which is invoked at runtime, we lose the syntax of the program and can only retrieve the bytecode. An advantage of analysing bytecode is that we leverage the work done by the bytecode compiler such as lexical analysis the identification of variables as either locals or globals. We also do not need to implement features that are just "syntax sugar".

The bytecode (Figure 3.2) is based on a simplified machine model consisting of a store (for mapping variables to constants), an integer-valued program counter and a single accumulator register for calculations. The Python runtime contains an evaluation stack that is used as working memory. μPython does not have an evaluation stack. We make use of a reserved variable called tos, in the environment, instead of a stack. This works like an accumulator but has stack-like properties. For example, this gets invalidated when read (pop) and takes the role of the *top of stack* in the full Python language. In the actual implementation of our type inference, we fully support an evaluation stack (Section 6.6). We use the metavariables u, v to range over names including tos. Apart from the constants defined in Figure 3.1, lists of instructions can also act as constants.

Similar to the high-level syntax, we choose a subset of actual Python bytecodes, albeit with minor modifications, sufficient to represent the challenges involved with static type analysis in a dynamically typed language. We reuse the namespace $\mathbb{V}$ for variable and function names but, in order to model functions, we extend the set of constants to include constants of type Fn made of finite sequences of bytecode instructions. For technical convenience we also add a constant U of type Un and a non-deterministic boolean value $*$. We shall be referring to this reduced language for any formal definitions.

Following our bytecode definitions, we informally describe the bytecode instructions as follows:

LC c – Loads the constant supplied as operand in the top of stack: $tos := c$.

LG x – Loads the value stored in a global variable into tos: $tos := x$. The name of the global variable $x \in \mathbb{V}$ is supplied as the operand to the instruction.

SG x – Stores the value held inside tos to a global variable supplied as operand. This consumes tos and therefore tos becomes U. The name of the global variable $x \in \mathbb{V}$ is supplied as the operand to the instruction.

JP n – Jumps unconditionally to the location supplied as operand: $pc := n$.

JIF n – Jumps to the location supplied as operand if the top of stack contains false value. Consumes tos. Raises a TypeError if tos is not a boolean.

CF f – Calls function f, where f is a global variable. To execute this the machine finds the sequence of instructions P mapped from f in the store and pushes this program on to the call stack with program counter 0. Raises a TypeError if f is not a function.

RET– Returns from the function call by popping an element from the call stack. Then execution is resumed from the previous location found on the call stack. If the call stack is empty, the execution is halted.

intOp – Consumes tos, i.e., tos becomes U, if the value in tos is an Int. Raises a TypeError if tos is not an Int.

strOp – Consumes tos if the value in tos is a Str. Raises a TypeError if tos is not a Str.

isInst τ – Given a type τ, replaces the current tos with true or false depending on whether the value in tos is of the type τ.

raise – Raises an assertion error.

We assume well-formed bytecode where jumps only refer to actual program locations other than location 0 and every program has a RET-instruction at its final location.

3.3 Compiling and running μPython

We use the μPython source language primarily for examples, since our analysis is based on the μPython bytecode. We therefore define a compiler that translates μPython source to labelled bytecode. This is defined as a compilation operation C, which is defined inductively over the structure of terms by the rules in Figure 3.3. These rules are applied in top-down order since there are two overlapping rules for $f(e)$ and $e_1(e_2)$. In this case, we want to give precedence to the former rule since it yields more optimal code.

The process of mapping labels l_1 and l_2 in the generated bytecode to bytecode offsets is not specified but is straightforward. The translator also needs to generate a fresh name f for function applications of the form $e_1(e_2)$. An interesting rule is the one for creating function definitions ($C(\mathsf{def}\ f(x) : s)$). To compile the function definition, the function body s is compiled and its bytecode is stored in f.

$$
\begin{aligned}
C(x) &= \mathsf{LG}\ x \\
C(c) &= \mathsf{LC}\ c \\
C(f()) &= \mathsf{CF}\ f \\
C(f(e)) &= C(e); \mathsf{CF}\ f \\
C(\mathsf{intOp}(e)) &= C(e); \mathsf{intOp} \\
C(\mathsf{strOp}(e)) &= C(e); \mathsf{strOp} \\
C(\mathsf{isInst}(e, \tau)) &= C(e); \mathsf{isInst}\ \tau \\
C(e_1(e_2)) &= C(e_1); \mathsf{SG}\ f; C(e_2); \mathsf{CF}\ f \\
C(\mathsf{def}\ f() : s) &= \mathsf{LC}\ C(s); \mathsf{SG}\ f \\
C(\mathsf{def}\ f(x) : s) &= \mathsf{LC}\ [\mathsf{SG}\ x; C(s);]; \mathsf{SG}\ f \\
C(\mathsf{return}\ e) &= C(e); \mathsf{RET} \\
C(\mathsf{pass}) &= \varepsilon \\
C(\mathsf{raise}) &= \mathsf{raise} \\
C(x = e) &= C(e); \mathsf{SG}\ x \\
C(\mathsf{if}\ e : s_1\ \mathsf{else} : s_2) &= C(e); \mathsf{JIF}\ l_1; C(s_1); \mathsf{JP}\ l_2;\ l_1 : C(s_2);\ l_2 : \\
C(\mathsf{while}\ e : s) &= l_1 : C(e); \mathsf{JIF}\ l_2; C(s); \mathsf{JP}\ l_1;\ l_2 : \ldots \\
C(s_1; s_2) &= C(s_1); C(s_2)
\end{aligned}
$$

Figure 3.3: Compiler for μPython. Compilation operation C is defined inductively over the structure of terms in the rules. Rules are applied in top-down order.

We formalise the semantics through rules for single execution steps of an abstract machine, as shown in Figure 3.4. These are a direct formalisation of the informal description given in the previous section. The states of the machine, $State^{\rightarrow}$, are one of the termination states TypeError, Exception, or End, or of the form $\langle \Sigma, S \rangle$. The *environment* Σ is a mapping from names, including tos, to constants and S is a *call stack* of ⟨program, program counter⟩ pairs.

For technical convenience, our machine initially performs an initialisation step. If we assume that the machine begins in state $\langle \Sigma_0, \varepsilon \rangle$ where ε is an empty call stack, the state is initialised to Σ_I, a store that contains mappings for built-ins and that maps all other names to U. We write M for the initial, or main, program and P_n to refer to the bytecode instruction at location n in program P. We write $\Sigma(u)$ to denote lookup in Σ and $\Sigma \oplus (u \mapsto c)$ to denote the environment Σ updated with the mapping $u \mapsto c$. We also write $\Sigma(u) : \tau$ whenever Σ maps u to a constant of principal type τ.

3.4 Example

Now that we have defined μPython, we present a simple program example in Figure 3.5. In this program, a function f is defined, which performs an integer operation on variable x. The program then branches non-deterministically. If the consequent branch is taken, x is assigned to a string. If the alternative branch is taken, x is assigned to an integer. When f is called, an integer operation is performed on x. This program raises a TypeError depending on the branch taken at runtime.

$$
\begin{array}{rcll}
\langle \Sigma_0, \varepsilon \rangle & \rightarrow & \langle \Sigma_I, \langle M, 0 \rangle :: \varepsilon \rangle & \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \rightarrow & \mathsf{End} & \text{if } P_{pc} = \mathsf{RET}, S = \varepsilon \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \rightarrow & \langle \Sigma, S \rangle & \text{if } P_{pc} = \mathsf{RET}, S \neq \varepsilon \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \rightarrow & \langle \Sigma \oplus (tos \mapsto c), \langle P, pc+1 \rangle :: S \rangle & \text{if } P_{pc} = \mathsf{LC}\ c \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \rightarrow & \langle \Sigma \oplus (tos \mapsto \Sigma(x)), \langle P, pc+1 \rangle :: S \rangle & \text{if } P_{pc} = \mathsf{LG}\ x \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \rightarrow & & \text{if } P_{pc} = \mathsf{SG}\ x \\
\multicolumn{4}{c}{\langle \Sigma \oplus (x \mapsto \Sigma(tos)) \oplus (tos \mapsto \mathsf{U}), \langle P, pc+1 \rangle :: S \rangle} \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \rightarrow & \langle \Sigma, \langle P, n \rangle :: S \rangle & \text{if } P_{pc} = \mathsf{JP}\ n \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \rightarrow & \langle \Sigma \oplus (tos \mapsto \mathsf{U}), \langle P, n \rangle :: S \rangle & \text{if } P_{pc} = \mathsf{JIF}\ n, \Sigma(tos) = \mathsf{false} \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \rightarrow & \langle \Sigma \oplus (tos \mapsto \mathsf{U}), \langle P, pc+1 \rangle :: S \rangle & \text{if } P_{pc} = \mathsf{JIF}\ n, \Sigma(tos) = \mathsf{true} \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \rightarrow & \mathsf{TypeError} & \text{if } P_{pc} = \mathsf{JIF}\ n, \neg\Sigma(tos) : \mathsf{Bool} \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \rightarrow & \langle \Sigma, \langle P', 0 \rangle :: \langle P, pc+1 \rangle :: S \rangle & \text{if } P_{pc} = \mathsf{CF}\ f, \Sigma(f) = P' \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \rightarrow & \mathsf{TypeError} & \text{if } P_{pc} = \mathsf{CF}\ f, \neg\Sigma(f) : \mathsf{Fn} \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \rightarrow & \langle \Sigma \oplus (tos \mapsto \mathsf{U}), \langle P, pc+1 \rangle :: S \rangle & \text{if } P_{pc} = \mathsf{intOp}, \Sigma(tos) : \mathsf{Int} \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \rightarrow & \langle \Sigma \oplus (tos \mapsto \mathsf{U}), \langle P, pc+1 \rangle :: S \rangle & \text{if } P_{pc} = \mathsf{strOp}, \Sigma(tos) : \mathsf{Str} \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \rightarrow & \mathsf{TypeError} & \text{if } P_{pc} = \mathsf{intOp}, \neg\Sigma(tos) : \mathsf{Int} \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \rightarrow & \mathsf{TypeError} & \text{if } P_{pc} = \mathsf{strOp}, \neg\Sigma(tos) : \mathsf{Str} \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \rightarrow & \langle \Sigma \oplus (tos \mapsto \mathsf{true}), \langle P, pc+1 \rangle :: S \rangle & \text{if } P_{pc} = \mathsf{isInst}\ \tau, \Sigma(tos) : \tau \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \rightarrow & \langle \Sigma \oplus (tos \mapsto \mathsf{false}), \langle P, pc+1 \rangle :: S \rangle & \text{if } P_{pc} = \mathsf{isInst}\ \tau, \neg\Sigma(tos) : \tau \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \rightarrow & \mathsf{Exception} & \text{if } P_{pc} = \mathsf{raise}
\end{array}
$$

Figure 3.4: Semantics of the μPython Bytecode

```
def f():
    return intOp(x)
if *:
    x = '42'
else:
    x = 42
f()
```

Figure 3.5: A simple μPython example

We first go through the compilation process of the source program defined in Figure 3.5. This is done by applying the compilation operation C, defined in Figure 3.3, inductively over the program. We show this at each line, where we underline the parts that are going to be compiled

on the following line.

$$
\begin{aligned}
& C(\underline{\mathsf{def}\ f() : \mathsf{return}\ \mathsf{intOp}(x);\quad \mathsf{if} * : x = \text{'42'}\ \mathsf{else} : x = 42;\quad f()}) \\
= \;& C(\underline{\mathsf{def}\ f() : \mathsf{return}\ \mathsf{intOp}(x)}); C(\mathsf{if} * : x = \text{'42'}\ \mathsf{else} : x = 42;\quad f()) \\
= \;& \mathsf{LC}\ [C(\underline{\mathsf{return}\ \mathsf{intOp}(x)})]; \mathsf{SG}\ f; C(\mathsf{if} * : x = \text{'42'}\ \mathsf{else} : x = 42;\quad f()) \\
= \;& \mathsf{LC}\ [C(\underline{\mathsf{intOp}(x)}); \mathsf{RET}]; \mathsf{SG}\ f; C(\mathsf{if} * : x = \text{'42'}\ \mathsf{else} : x = 42;\quad f()) \\
= \;& \mathsf{LC}\ [\mathsf{LG}\ x; \mathsf{intOp}; \mathsf{RET}]; \mathsf{SG}\ f; C(\underline{\mathsf{if} * : x = \text{'42'}\ \mathsf{else} : x = 42;\quad f()}) \\
= \;& \mathsf{LC}\ [\mathsf{LG}\ x; \mathsf{intOp}; \mathsf{RET}]; \mathsf{SG}\ f; C(\underline{\mathsf{if} * : x = \text{'42'}\ \mathsf{else} : x = 42}); C(f()) \\
= \;& \mathsf{LC}\ [\mathsf{LG}\ x; \mathsf{intOp}; \mathsf{RET}]; \mathsf{SG}\ f; C(\underline{*}); \mathsf{JIF}\ l_1; C(x = \text{'42'}); \mathsf{JP}\ l_2; l_1 : C(x = 42); l_2 : C(f()) \\
= \;& \mathsf{LC}\ [\mathsf{LG}\ x; \mathsf{intOp}; \mathsf{RET}]; \mathsf{SG}\ f; \mathsf{LC}\ *; \mathsf{JIF}\ l_1; C(\underline{x = \text{'42'}}); \mathsf{JP}\ l_2; l_1 : C(x = 42); l_2 : C(f()) \\
= \;& \mathsf{LC}\ [\mathsf{LG}\ x; \mathsf{intOp}; \mathsf{RET}]; \mathsf{SG}\ f; \mathsf{LC}\ *; \mathsf{JIF}\ l_1; C(\underline{\text{'42'}}); \mathsf{SG}\ x; \mathsf{JP}\ l_2; l_1 : C(x = 42); l_2 : C(f()) \\
= \;& \mathsf{LC}\ [\mathsf{LG}\ x; \mathsf{intOp}; \mathsf{RET}]; \mathsf{SG}\ f; \mathsf{LC}\ *; \mathsf{JIF}\ l_1; \mathsf{LC}\ \text{'42'}; \mathsf{SG}\ x; \mathsf{JP}\ l_2; l_1 : C(\underline{x = 42}); l_2 : C(f()) \\
= \;& \mathsf{LC}\ [\mathsf{LG}\ x; \mathsf{intOp}; \mathsf{RET}]; \mathsf{SG}\ f; \mathsf{LC}\ *; \mathsf{JIF}\ l_1; \mathsf{LC}\ \text{'42'}; \mathsf{SG}\ x; \mathsf{JP}\ l_2; l_1 : C(\underline{42}); \mathsf{SG}\ x; l_2 : C(f()) \\
= \;& \mathsf{LC}\ [\mathsf{LG}\ x; \mathsf{intOp}; \mathsf{RET}]; \mathsf{SG}\ f; \mathsf{LC}\ *; \mathsf{JIF}\ l_1; \mathsf{LC}\ \text{'42'}; \mathsf{SG}\ x; \mathsf{JP}\ l_2; l_1 : \mathsf{LC}\ 42; \mathsf{SG}\ x; l_2 : C(\underline{f()}) \\
= \;& \mathsf{LC}\ [\mathsf{LG}\ x; \mathsf{intOp}; \mathsf{RET}]; \mathsf{SG}\ f; \mathsf{LC}\ *; \mathsf{JIF}\ l_1; \mathsf{LC}\ \text{'42'}; \mathsf{SG}\ x; \mathsf{JP}\ l_2; l_1 : \mathsf{LC}\ 42; \mathsf{SG}\ x; l_2 : \mathsf{CF}\ f
\end{aligned}
$$

The labels in the bytecode program are now replaced with explicit offsets in the bytecode instruction lists. The instruction RET is also appended to the end to mark the end of the program.

$$
\mathsf{LC}\ [\underset{0}{\mathsf{LG}}\ x; \overset{0}{\underset{1}{\mathsf{intOp}}}; \underset{2}{\mathsf{RET}}]; \overset{1}{\mathsf{SG}}\ f; \overset{2}{\mathsf{LC}}\ *; \overset{3}{\mathsf{JIF}}\ 7; \overset{4}{\mathsf{LC}}\ \text{'42'}; \overset{5}{\mathsf{SG}}\ x; \overset{6}{\mathsf{JP}}\ 9; \overset{7}{\mathsf{LC}}\ 42; \overset{8}{\mathsf{SG}}\ x; \overset{9}{\mathsf{CF}}\ f; \overset{10}{\mathsf{RET}}
$$

For presentation purposes, we define this program as M and P^f such that:

$$
\begin{aligned}
M &= [\overset{0}{\mathsf{LC}}\ P^f;\ \overset{1}{\mathsf{SG}}\ f;\ \overset{2}{\mathsf{LC}}\ *;\ \overset{3}{\mathsf{JIF}}\ 7;\ \overset{4}{\mathsf{LC}}\ \text{'42'};\ \overset{5}{\mathsf{SG}}\ x;\ \overset{6}{\mathsf{JP}}\ 9;\ \overset{7}{\mathsf{LC}}\ 42;\ \overset{8}{\mathsf{SG}}\ x;\ \overset{9}{\mathsf{CF}}\ f;\ \overset{10}{\mathsf{RET}}] \\
P^f &= [\underset{0}{\mathsf{LG}}\ x;\ \underset{1}{\mathsf{intOp}};\ \underset{2}{\mathsf{RET}}]
\end{aligned}
$$

The semantics of μPython (see Figure 3.4) is defined over the bytecode. Each rule is a single execution step of the abstract machine. We now show how every rule is applied, at every step. On the right hand side, we include the instruction that is present at the top of the current call stack in P_{pc}. The machine starts at state $\langle \Sigma_0, \varepsilon \rangle$ and the first step is a bootstrapping step which takes the machine to $\langle \Sigma_I, \langle M, 0 \rangle :: \varepsilon \rangle$. This loads the main or initial program M together with the offset of the first bytecode instruction (0) onto the call stack.

$$
\begin{aligned}
\langle \Sigma_0, \varepsilon \rangle &\to \langle \Sigma_I, \langle M, 0 \rangle :: \varepsilon \rangle && (\mathsf{LC}\ P^f) \\
&\to \langle \Sigma_I \oplus (tos \mapsto P^f), \langle M, 1 \rangle \rangle && (\mathsf{SG}\ f) \\
&\to \langle \Sigma_I \oplus (f \mapsto P^f) \oplus (tos \mapsto \mathsf{U}), \langle M, 2 \rangle \rangle && (\mathsf{LC}\ *) \\
&\to \langle \Sigma_I \oplus (f \mapsto P^f) \oplus (tos \mapsto *), \langle M, 3 \rangle \rangle && (\mathsf{JIF}\ 7)
\end{aligned}
$$

At this point, the jump depends on the actual value of $*$. This is a non deterministic boolean value. We resume the execution of this example with the assumption that $*$ is false.

$$\begin{aligned}
&\rightarrow \langle \Sigma_I \oplus (f \mapsto P^f) \oplus (tos \mapsto \mathsf{U}), \langle M, 7\rangle\rangle && (\mathsf{LC}\ 42)\\
&\rightarrow \langle \Sigma_I \oplus (f \mapsto P^f) \oplus (tos \mapsto 42), \langle M, 8\rangle\rangle && (\mathsf{SG}\ x)\\
&\rightarrow \langle \Sigma_I \oplus (f \mapsto P^f) \oplus (tos \mapsto \mathsf{U}) \oplus (x \mapsto 42), \langle M, 9\rangle\rangle && (\mathsf{CF}\ f)\\
&\rightarrow \langle \Sigma_I \oplus (f \mapsto P^f) \oplus (tos \mapsto \mathsf{U}) \oplus (x \mapsto 42), \langle P^f, 0\rangle :: \langle M, 9\rangle\rangle && (\mathsf{LG}\ x)\\
&\rightarrow \langle \Sigma_I \oplus (f \mapsto P^f) \oplus (tos \mapsto 42) \oplus (x \mapsto 42), \langle P^f, 1\rangle :: \langle M, 9\rangle\rangle && (\mathsf{intOp})\\
&\rightarrow \langle \Sigma_I \oplus (f \mapsto P^f) \oplus (tos \mapsto \mathsf{U}) \oplus (x \mapsto 42), \langle P^f, 2\rangle :: \langle M, 9\rangle\rangle && (\mathsf{RET})\\
&\rightarrow \langle \Sigma_I \oplus (f \mapsto P^f) \oplus (tos \mapsto \mathsf{U}) \oplus (x \mapsto 42), \langle M, 10\rangle\rangle && (\mathsf{RET})\\
&\rightarrow \mathsf{End}
\end{aligned}$$

Hence we have shown one possible execution and outcome of the program. This results in a termination state End. The other outcome would have raised a TypeError.

3.5 Relationship to Python 3.3

In order to simplify the formalisation as much as possible, we cut down on the size of the language through assumptions and simplifications of the original Python bytecodes as follows:

All variables are global. In standard Python, there are both local and global variables. There are also variables that appear in certain scopes such as closures and object attributes. One of the biggest challenge in Python and similar languages is that the types of global variables are mutable at any execution point in the program.

No evaluation stack. As we mentioned already, we make use of a reserved variable called tos, in the environment, instead of a stack. This works like an accumulator. With this restriction in place, only functions with zero or one arguments are supported. If a function that takes more arguments is required, then we can simply make use of currying.

The argument to call function CF *is the actual function name.* In Python, the function called by this instruction is pushed on the stack first. Then, the arguments are also pushed on the stack. Since we no longer have a stack, we modify this bytecode instruction to include the function name as its argument and the function's argument is passed through tos.

No need for the make function instruction. In Python, a bytecode instruction `MAKE_FUNCTION` is available. This takes a code object on the top of the execution stack and transforms it into a function. The code object encapsulates the bytecode. This instruction also takes an integer argument which indicates how many arguments the function takes, and leaves a function in the

top of the stack. In μPython, however, functions only take at most a single argument and are just a list of bytecode instructions. We therefore do not need the `MAKE_FUNCTION` instruction.

Simpler constants. In μPython, constants are either strings, integers, booleans, or programs (lists of bytecode instructions). Full Python has many more kinds of constants.

We have also shortened the names of our bytecode instructions.

3.6 Conclusion

In this chapter we have formalised a simple dynamically typed language called μPython. This is modelled on a core subset of the Python language. We defined the syntax of the high-level language and also the compiled bytecode. We then formalised a non-optimising compiler, which compiles the source language to bytecode. The semantics of the language was formalised for the bytecode. An example showing the source code, compilation and execution of a program was also given in this chapter.

We shall use this language in the next two chapters. Our type inference algorithm and type checking mechanism is defined on this core

Chapter 4

Type inference for μPython

This chapter contains a formal description of the type inference algorithms for μPython. We mainly give a description of the type system and the rules of the type inference mechanism. We also present proofs that show that the type inference algorithms are correct in terms of correctness properties that we also define in this chapter.

We formalise our type inference algorithm for the μPython bytecode rather than source code. Although it is common for programming language tools to work on bytecode, formalising of type inference algorithms on a bytecode language is less common. Therefore the main reason why we model bytecode is to remain faithful to the implementation, which performs a bytecode analysis. Since our analysis is performed at runtime, we do not have the structure of the source code and we can only retrieve the bytecode.

4.1 Types

A key characteristic of μPython as a dynamically typed language is that the types of variables may change during execution. Therefore, to determine whether a type error may occur we need to establish, for any given point in the program execution, two pieces of information: the type a variable actually has and the type a variable may be used as in future. We call these the *present* and *future use* types.

In the case of the present type, $x : \tau$ holds at a particular point if *after* executing the instruction at that point, x contains a value whose type is τ. To establish this, we perform a traditional forwards analysis over the execution points of the program; the present type of a variable depends on the instructions that have previously been executed. Obviously the precise present runtime type of a variable cannot be statically determined so our analysis uses an over-approximation of this to

determine the present types. We extend the grammar of types to be

$$
\begin{array}{rcll}
\tau & ::= & \mathsf{Int} & \text{integer} \\
& | & \mathsf{Str} & \text{string} \\
& | & \mathsf{Bool} & \text{boolean} \\
& | & \mathsf{Un} & \text{uninitialised} \\
& | & \mathsf{Fn} & \text{function type} \\
& | & \bot & \text{bottom type} \\
& | & \top & \text{top type} \\
& | & \tau \sqcup \tau & \text{union type}
\end{array}
$$

In order to represent the different type possibilities for a given variable we make use of the familiar concept of union types. These come equipped with a natural subtyping order. We define the subtyping order inductively using the following rules:

$$
\tau <: \tau \qquad \frac{\tau <: \tau' \quad \tau' <: \tau''}{\tau <: \tau''} \qquad \bot <: \tau \qquad \tau <: \top
$$

$$
\frac{\tau <: \tau'}{\tau <: \tau' \sqcup \tau''} \qquad \frac{\tau <: \tau''}{\tau <: \tau' \sqcup \tau''} \qquad \frac{\tau <: \tau'' \quad \tau' <: \tau''}{\tau \sqcup \tau' <: \tau''}
$$

We note that types form a lattice. The join operation on two types τ, τ', for example, is simply defined as the equivalence class of $\tau \sqcup \tau'$. The partial order of the lattice is the least partial order induced by the subtyping preorder. In order to define the meet operation, we do not need to introduce a syntactic construct to the type grammar. Instead, we will define the meet operation in Section 4.4 in terms of $\sqcup$ for a finite number of type terms. A part of the type lattice is illustrated in Figure 4.1, where $\bot$ and $\top$ sit at the bottom and the top of the lattice respectively. Two types of interest are Fn and Un. Fn is the type of any callable function while Un indicates that the variable is currently unassigned, and is defined as the type of the constant U.

Dual to the analysis of present types we establish the future use type using a backwards analysis so that the future use type depends on the next instructions that will be executed. In the case of the future use type, for $x : \tau$ to hold at point s, x must have a type that is "compatible with" τ before executing the operation at s. If it does not, the program will result in a type error, either at s or at any point accessible from s. The future use types are primarily introduced by function calls, but also by other instructions such as conditional jumps, which require the operand to be a Bool. A type τ is "compatible with" another type τ' if it is possible to use a value with type τ whenever we require a value of type τ'. In μPython, this relation is captured by the subtype relation. Looking at the present and future use types at every program execution point, it is possible to determine whether a program will reduce to a TypeError. We can now note that there is a relationship between $\top$ and Un. Un is the type of any variable that is not assigned in the present type environment while $\top$ is the type of any dead variable in the future use type environment.

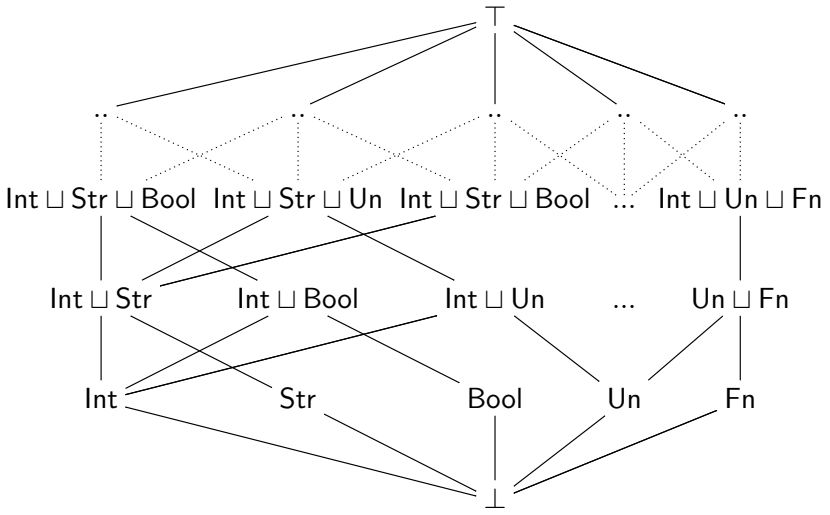

Figure 4.1: Fragment of the type lattice, formed under the subtype operation.

4.2 Program execution points

Our type analysis establishes the type of any variable at any point. Since variables can change type during execution, a naive idea of a program execution point might be a simple code location. We must realise, however, that the variables in the outer scope of a function can have different types according to where a function is called. Therefore, the entire call stack is important in determining the current types of any variable. In principle, program execution points must therefore be full call stacks and the control flow graph (CFG) of a μPython program is therefore a relation $S \to S'$ between call stacks. This is unfortunate because, even for finite programs, the CFG of all possible program execution points could then be infinite. This has drastic consequences for a static analysis.

We address this issue by over-approximating the CFG via the simple means of *truncating* call stacks. Specifically, given a call stack S, and an integer $N \geq 1$, we write $\lfloor S \rfloor_N$ to mean the equivalence class of all call stacks whose prefix of length N is the same as that of the stack S. We typically omit N as this is fixed throughout. We refer to these equivalence classes as *truncated execution points* and it is clear that, for each program, they form a finite, truncated CFG as follows:

$$\lfloor S \rfloor \to \lfloor S' \rfloor \text{ if and only if } S_0 \to S'_0 \text{ for some } S_0 \in \lfloor S \rfloor, S'_0 \in \lfloor S' \rfloor$$

We will use a shorthand notation in the remainder by writing s to mean $\lfloor S \rfloor$, s' to mean $\lfloor S' \rfloor$, etc. We will also make extensive use of the following two functions: given a truncated execution point s we write $\mathrm{prev}(s)$ for the set of nodes from which s can be reached in the truncated CFG of the program. Similarly, $\mathrm{next}(s)$ denotes the set of nodes which can be reached from s.

Although we do not formalise the process of constructing control flow graphs of execution points, clearly this process is closely linked with abstract interpretation [27]. In this perspective, our concrete state space is the set of program states $\langle \Sigma, S \rangle$, which is uncountably infinite. Our abstract state space is the set of truncated execution points s. Therefore, the abstraction function α can be naturally expressed as:

$$\alpha(X) = \{\lfloor S \rfloor \mid \langle \Sigma, S \rangle \in X\}$$

where X is a set of states $State^{\rightarrow}$. The concretisation function γ can be naturally expressed as:

$$\gamma(Y) = \{\langle \Sigma, S \rangle \mid S = s :: ... \wedge s \in Y\}$$

where Y is a set of truncated execution points.

When the μPython interpreter is started, it is started with an empty call stack. Thus the first execution point is denoted as ε. At the heart of our analysis is the forwards/backwards traversal of the truncated CFG using the $\mathrm{prev}(s)$ and $\mathrm{next}(s)$ functions in order to find the present and future use types of variables. Functions prev and next return finite sets.

We assume that the set of execution points returned by prev contains all possible previous execution points of a given program execution point that could appear at runtime. This means that for any state of a running program, if $\langle \Sigma, S \rangle \rightarrow \langle \Sigma', S' \rangle$ then

$$s \in \mathrm{prev}(s') \tag{4.1}$$

We similarly assume the same for $\mathrm{next}(s)$:

$$s' \in \mathrm{next}(s) \tag{4.2}$$

In Figure 4.2, we have an illustration depicting the correspondence between program points and stacks. In this figure, program states $\langle \Sigma_0, S_0 \rangle$ and $\langle \Sigma_1, S_1 \rangle$ are executed by a single step to yield states $\langle \Sigma'_0, S'_0 \rangle$ and $\langle \Sigma'_1, S'_1 \rangle$ respectively, and S_0 and S_1 both truncate to s. However, the truncations of S'_0 and S'_1 are s'_0 and s'_1 respectively. Therefore, $\mathrm{next}(s)$ has to at least contain $\{s'_0, s'_1\}$.

The simplest way to truncate the call stack is to retain only the last element. In this case, this is the currently executing function and program counter. The longer the truncated stack is, the smaller the overapproximation of the previous and next program execution points will be and the more precise the inferred types will be.

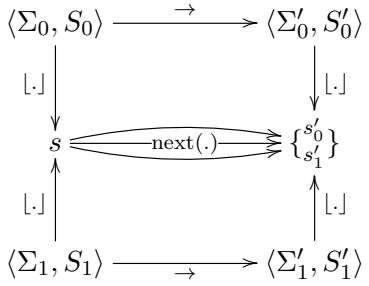

Figure 4.2: An illustration of the correspondence between stacks and execution points, where $\{s'_0, s'_1\} \subseteq \text{next}(s)$.

4.3 Type inference

Since the types associated with variables depend on the points in the control flow, the inference mechanism traverses the control flow graph. We propose a type inference mechanism that is similar to symbolic execution of the program using an abstract semantics of μPython encoded inside inference rules. These inference rules capture the present and future use types of a particular variable at a particular execution point. We start by defining type judgements that are inductively defined relations between execution points, variable names and types. For example, in the case of present types, the judgement has the form $s \vdash u : \tau$. This denotes that u has type τ after executing the instruction at s.

For inferring the present type of any variable or tos right after the execution point ε, i.e., just before executing the code M, we introduce the following rule:

$$\frac{\Sigma_I(u) : \tau}{\varepsilon \vdash u : \tau}\,\textsf{INIT}$$

This rule effectively says that type of u is the type of the value appearing in the initial runtime environment Σ_I. One can note that we make reference to a runtime environment in the type inference rules, which are typically designed for static analysis. This is because our type inference algorithms are designed to be invoked at runtime and therefore we can use the type information available in the runtime environment at the point that the inference algorithm is used. We now introduce another rule for a subset of the μPython instructions. Note that the derivation in the conclusion makes reference to type derivations in its premise:

$$\frac{\begin{array}{c} s = \langle P, pc \rangle :: \ldots \quad P_{pc} \in \{\textsf{LC}\ c, \textsf{JIF}\ pc', \textsf{RET}\} \\ s_i \vdash x : \bigsqcup \tau_i \text{ for each } s_i \in \text{prev}(s) \end{array}}{s \vdash x : \bigsqcup \tau_i}\,\textsf{PREV}$$

This rule says that if the instruction at s is any one of $\{\textsf{LC}\ c, \textsf{JIF}\ pc', \textsf{RET}\}$, then the present type of a variable x at s is obtained by joining the present type of x at every execution point s_i preceding s. The proof tree for this rule therefore spans through the control flow graph of

the program, and branches whenever there is a control flow join in the graph. Let us now apply these rules on a small program M. In M, we load a non-deterministic boolean a number of times until the value false is loaded, and exit:

$$M = [\underset{0}{\mathsf{LC}}\ *; \underset{1}{\mathsf{JIF}}\ 0; \underset{2}{\mathsf{RET}}]$$

The control flow graph of M is therefore:

$$\varepsilon \longrightarrow \langle M, 0\rangle \longrightarrow \langle M, 1\rangle \longrightarrow \langle M, 2\rangle$$

We can now attempt to infer the present type of x, which is not used in M, after executing the instruction at execution point $\langle M, 2\rangle$. We utilise the two rules we have just defined for inferring the present types for the μPython subset used in M. We also assume that the initial environment Σ_I maps everything to Un. We build our tree by starting with the judgement $\langle M, 2\rangle \vdash x : \tau$, and proceed to build the proof tree as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Sigma_I(x) : \mathsf{Un}}{\varepsilon \vdash x : \mathsf{Un}}\,\textsc{init} \quad \dfrac{\cdots\cdots}{\langle M, 1\rangle \vdash x : \mathsf{Un} \sqcup \ldots}\,\textsc{prev}}{\langle M, 0\rangle \vdash x : \mathsf{Un} \sqcup \ldots}\,\textsc{prev}}{\langle M, 1\rangle \vdash x : \mathsf{Un} \sqcup \ldots}\,\textsc{prev}}{\langle M, 2\rangle \vdash x : \mathsf{Un} \sqcup \ldots}\,\textsc{prev}$$

Unfortunately, we quickly discover that the structure of the tree will repeat itself due to the cycle between $\langle M, 1\rangle$ and $\langle M, 0\rangle$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Sigma_I(x) : \mathsf{Un}}{\varepsilon \vdash x : \mathsf{Un}}\,\textsc{init} \quad \dfrac{\cdots\cdots}{\langle M, 1\rangle \vdash x : \mathsf{Un} \sqcup \ldots}\,\textsc{prev}}{\langle M, 0\rangle \vdash x : \mathsf{Un} \sqcup \ldots}\,\textsc{prev}}{\Sigma_I(x) : \mathsf{Un}\ \ \langle M, 1\rangle \vdash x : \mathsf{Un} \sqcup \ldots}\,\textsc{init}}{\langle M, 0\rangle \vdash x : \mathsf{Un} \sqcup \ldots}\,\textsc{prev}}{\langle M, 1\rangle \vdash x : \mathsf{Un} \sqcup \ldots}\,\textsc{prev}}{\langle M, 0\rangle \vdash x : \mathsf{Un} \sqcup \ldots}\,\textsc{prev}}{\langle M, 1\rangle \vdash x : \mathsf{Un} \sqcup \ldots}\,\textsc{prev}}{\langle M, 2\rangle \vdash x : \mathsf{Un} \sqcup \ldots}\,\textsc{prev}$$

We need a way to break this cycle, which can occur in any program that has loops or recursion. In order to address this, we introduce a mechanism called *trails*, which we explain in the next section.

$$s = \langle P, pc \rangle :: \ldots$$

$$\frac{\Sigma_I(u) : \tau}{\langle \varepsilon, \mathcal{T} \rangle \vdash_p u : \tau}\textsf{pINIT} \qquad \frac{\langle s, u \rangle \in \mathcal{T}}{\langle s, \mathcal{T} \rangle \vdash_p u : \bot}\textsf{pTRAIL} \qquad \frac{\langle s, u \rangle \notin \mathcal{T} \quad P_{pc} = \textsf{raise}}{\langle s, \mathcal{T} \rangle \vdash_p u : \bot}\textsf{pRAISE}$$

$$\frac{\langle s, tos \rangle \notin \mathcal{T} \quad P_{pc} = \textsf{LC}\ c \quad c : \tau}{\langle s, \mathcal{T} \rangle \vdash_p tos : \tau}\textsf{pLC} \qquad \frac{\langle s, tos \rangle \notin \mathcal{T} \quad P_{pc} = \textsf{isInst}\ \tau}{\langle s, \mathcal{T} \rangle \vdash_p tos : \textsf{Bool}}\textsf{pINST}$$

$$\frac{\langle s, tos \rangle \notin \mathcal{T} \quad P_{pc} \in \{\textsf{SG}\ x, \textsf{JIF}\ n, \textsf{strOp}, \textsf{intOp}\}}{\langle s, \mathcal{T} \rangle \vdash_p tos : \textsf{Un}}\textsf{pUSE}$$

Figure 4.3: Inference rules for the $\vdash_p$ judgement (axioms).

4.4 Type inference rules and trails

We introduce trails in the type judgements for both present and future use types. The type inference is therefore expressed using two inductively defined relations written as

$$\langle s, \mathcal{T} \rangle \vdash_p u : \tau \quad \text{and} \quad \langle s, \mathcal{T} \rangle \vdash_f u : \tau$$

where s is a truncated execution point and $\mathcal{T}$ is a *trail*. A trail is a set of pairs $\langle s, u \rangle$ of truncated execution points and variables. They represent the previously visited execution points (together with the variables that triggered the visit) and are used to ensure termination of the inference. This is explained in more detail later on.

Similar to the judgement in the preceding section, $\langle s, \mathcal{T}_\emptyset \rangle \vdash_p u : \tau$ (where $\mathcal{T}_\emptyset$ is the empty trail) denotes that u will have type τ *after* the current instruction has been executed. The judgement $\langle s, \mathcal{T}_\emptyset \rangle \vdash_f u : \tau$ denotes that the variable u is required to have type τ in order to execute the instructions from the current instruction onward without raising a TypeError.

We now define the type inference rules with trails for both present and future use types. The rules for this are given in Figures 4.3–4.6.

The *axioms* for inferring $\vdash_p$ (cf. Figure 4.3) account for situations in which the present type is fully determined by the current instruction. For example, after loading a constant (Rule pLC) the accumulator is known to have the type of the constant that has just been loaded. The inference rules in Figure 4.4 all follow a shared pattern: the types of the relevant variables in each previous state are calculated and the present type is the union of these. The relevant variables are instruction dependent. For example, in Rule pSG1 for the instruction SG x the type of x depends on the type of tos in the previous states.

Again, for the rules for $\vdash_f$ we have axioms in Figure 4.5 and inference rules in Figure 4.6. Many of the axioms assign a future use type of $\top$ to a variable. This type indicates that there are no constraints on this variable and follows in cases where that variable is just about to be

$$\frac{s = \langle P, pc\rangle :: ... \quad \langle s, x\rangle \notin \mathcal{T} \quad P_{pc} = \mathsf{SG}\ x \quad \langle s_i, \mathcal{T} \cup \{\langle s, x\rangle\}\rangle \vdash_p tos : \tau_i \text{ for each } s_i \in \text{prev}(s)}{\langle s, \mathcal{T}\rangle \vdash_p x : \bigsqcup \tau_i}\ \mathsf{pSG1}$$

$$\frac{s = \langle P, pc\rangle :: ... \quad \langle s, tos\rangle \notin \mathcal{T} \quad P_{pc} = \mathsf{LG}\ x \quad \langle s_i, \mathcal{T} \cup \{\langle s, tos\rangle\}\rangle \vdash_p x : \tau_i \text{ for each } s_i \in \text{prev}(s)}{\langle s, \mathcal{T}\rangle \vdash_p tos : \bigsqcup \tau_i}\ \mathsf{pLG1}$$

$$\frac{s = \langle P, pc\rangle :: ... \quad \langle s, y\rangle \notin \mathcal{T} \quad P_{pc} = \mathsf{SG}\ x \quad x \neq y \quad \langle s_i, \mathcal{T} \cup \{\langle s, y\rangle\}\rangle \vdash_p y : \tau_i \text{ for each } s_i \in \text{prev}(s)}{\langle s, \mathcal{T}\rangle \vdash_p y : \bigsqcup \tau_i}\ \mathsf{pSG2}$$

$$\frac{s = \langle P, pc\rangle :: ... \quad \langle s, y\rangle \notin \mathcal{T} \quad P_{pc} = \mathsf{LG}\ x \quad \langle s_i, \mathcal{T} \cup \{\langle s, y\rangle\}\rangle \vdash_p y : \tau_i \text{ for each } s_i \in \text{prev}(s)}{\langle s, \mathcal{T}\rangle \vdash_p y : \bigsqcup \tau_i}\ \mathsf{pLG2}$$

$$\frac{s = \langle P, pc\rangle :: ... \quad \langle s, u\rangle \notin \mathcal{T} \quad P_{pc} \in \{\mathsf{RET}, \mathsf{JP}\ pc', \mathsf{CF}\ f\} \quad \langle s_i, \mathcal{T} \cup \{\langle s, u\rangle\}\rangle \vdash_p u : \tau_i \text{ for each } s_i \in \text{prev}(s)}{\langle s, \mathcal{T}\rangle \vdash_p u : \bigsqcup \tau_i}\ \mathsf{pRET/JP/CF}$$

$$\frac{s = \langle P, pc\rangle :: ... \quad \langle s, x\rangle \notin \mathcal{T} \quad P_{pc} \in \{\mathsf{LC}\ c, \mathsf{JIF}\ pc', \mathsf{strOp}, \mathsf{intOp}, \mathsf{isInst}\ \tau\} \quad \langle s_i, \mathcal{T} \cup \{\langle s, x\rangle\}\rangle \vdash_p x : \tau_i \text{ for each } s_i \in \text{prev}(s)}{\langle s, \mathcal{T}\rangle \vdash_p x : \bigsqcup \tau_i}\ \mathsf{p^*}$$

Figure 4.4: Inference rules for the $\vdash_p$ judgement.

overwritten (Rules fSET and fSG1). Hence, whatever is currently present in these variables is irrelevant. Otherwise, the immediate uses are recorded in the type (Rules fJIF, fSTR and fINT). Two interesting rules are fLG1 and fCF1. In these a variable is used but its contents remain intact so there may be future uses also. In the case of fCF1, f has to be a function type and also whatever it needs to be in the succeeding points. For this purpose we introduce a *meet operation* on types, written $\sqcap$. This is defined in the following rules, which are applied in top-down order:

$$\begin{array}{rcl} \tau \sqcap (\tau_1 \sqcup \tau_2) & = & (\tau \sqcap \tau_1) \sqcup (\tau \sqcap \tau_2) \\ (\tau_1 \sqcup \tau_2) \sqcap \tau & = & (\tau_1 \sqcap \tau) \sqcup (\tau_2 \sqcap \tau) \\ \tau \sqcap \top = \tau & & \top \sqcap \tau = \tau \\ \tau \sqcap \tau = \tau & & \tau_1 \sqcap \tau_2 = \bot \end{array}$$

We use trails of computations to prevent cycles in the proof tree when trying to infer the type of a particular variable. Without trails, cycles would occur (see previous section) since some of the rules for $\vdash_p$ and $\vdash_f$ are defined in terms of other derivations for $\vdash_p$ or $\vdash_f$ in the previous or next execution points in a program. Since a program usually contains cycles in the control flow graph

$$\frac{}{\langle \varepsilon, \mathcal{T} \rangle \vdash_f u : \top}\textsf{fINIT} \qquad \frac{s = \langle P, pc \rangle :: ... \quad P_{pc} = \textsf{RET}}{\langle \langle P, pc \rangle :: \varepsilon, \mathcal{T} \rangle \vdash_f u : \top}\textsf{fEND} \qquad \frac{P_{pc} = \textsf{SG}\ x \quad \langle s, x \rangle \notin \mathcal{T}}{\langle s, \mathcal{T} \rangle \vdash_f x : \top}\textsf{fSG1}$$

$$\frac{\langle s, u \rangle \in \mathcal{T}}{\langle s, \mathcal{T} \rangle \vdash_f u : \bot}\textsf{fTRAIL} \qquad \frac{P_{pc} = \textsf{raise} \quad \langle s, u \rangle \notin \mathcal{T}}{\langle s, \mathcal{T} \rangle \vdash_f u : \top}\textsf{fRAISE} \qquad \frac{P_{pc} \in \{\textsf{LC}\ c, \textsf{LG}\ x, \textsf{isInst}\ \tau\} \quad \langle s, tos \rangle \notin \mathcal{T}}{\langle s, \mathcal{T} \rangle \vdash_f tos : \top}\textsf{fSET}$$

$$\frac{P_{pc} = \textsf{JIF}\ pc' \quad \langle s, tos \rangle \notin \mathcal{T}}{\langle s, \mathcal{T} \rangle \vdash_f tos : \textsf{Bool}}\textsf{fJIF} \qquad \frac{P_{pc} = \textsf{strOp} \quad \langle s, tos \rangle \notin \mathcal{T}}{\langle s, \mathcal{T} \rangle \vdash_f tos : \textsf{Str}}\textsf{fSTR} \qquad \frac{P_{pc} = \textsf{intOp} \quad \langle s, tos \rangle \notin \mathcal{T}}{\langle s, \mathcal{T} \rangle \vdash_f tos : \textsf{Int}}\textsf{fINT}$$

Figure 4.5: Inference rules for the $\vdash_f$ judgement (axioms).

such as loops, these would also manifest themselves in the proof tree of a $\vdash_p$ or $\vdash_f$ judgement, as these branches along splits/joins in the control flow graph.

However, whenever the current execution point and variable are already present in the given trail, pTRAIL or fTRAIL are applied and the inferred type is $\bot$. This is because no more type information can be gained by passing through the same execution point looking for the type of the same variable twice. If this trail entry is not present, the entry is added to the current trail if another rule is applied. Now that we explained the role of trails, we recap the notation $\langle s, \mathcal{T} \rangle \vdash_p u : \tau$. This denotes that u will have type τ *after* the current instruction has been executed, omitting all type information that can be obtained by considering the items in the trail.

The type inference algorithms encoded in our inference rules are clearly related to symbolic execution, with some key variations. Our inference rules only relate one variable at a time with each application. In the case of present types and whenever there is a control flow join, the analysis is forked and the result from each is joined using $\sqcup$. In the case of future use types and whenever there is a control flow split, the same happens as well. This helps collapse the state space of the analysis. Another difference to symbolic execution is that we ignore the predicates of conditional jump instructions. This is because we only look at one variable at a time and we do not keep track of the actual possible values inside the variables, but only their type. Due to this design, we also explore some unfeasible paths. Since it is not possible in general to statically determine the iteration bounds in loops and recursion, we need a mechanism to minimise the number of times the analysis iterates through the loops. In our case, we solve this problem using trails.

It is worth noting that the trail sets $\mathcal{T}$ are finitely bounded. This is due to the fact that call stacks are truncated to a fixed depth and that, for a given program, there are finitely many code locations and finitely many variables. For a given program, we write $\mathcal{T}_U$ to denote the maximum trail containing all truncated execution point/variable pairs. Trails greatly facilitate the termination proof in the next section. Furthermore, if the type system is extended such that

$$\frac{s = \langle P, pc\rangle :: \ldots \quad \langle s, x\rangle \notin \mathcal{T} \quad P_{pc} = \mathsf{LG}\ x \quad \langle s_i, \mathcal{T} \cup \{\langle s, x\rangle\}\rangle \vdash_f tos : \upsilon_i \text{ for each } s_i \in \text{next}(s) \quad \langle s_i, \mathcal{T} \cup \{\langle s, x\rangle\}\rangle \vdash_f x : \nu_i \text{ for each } s_i \in \text{next}(s)}{\langle s, \mathcal{T}\rangle \vdash_f x : \bigsqcup(\upsilon_i \sqcap \nu_i)}\text{fLG1}$$

$$\frac{s = \langle P, pc\rangle :: \langle P', n\rangle :: \ldots \quad \langle s, u\rangle \notin \mathcal{T} \quad P_{pc} = \mathsf{RET} \quad \langle s_i, \mathcal{T} \cup \{\langle s, u\rangle\}\rangle \vdash_f u : \tau_i \text{ for each } s_i \in \text{next}(s)}{\langle s, \mathcal{T}\rangle \vdash_f u : \bigsqcup \tau_i}\text{fRET}$$

$$\frac{s = \langle P, pc\rangle :: \ldots \quad \langle s, tos\rangle \notin \mathcal{T} \quad P_{pc} = \mathsf{SG}\ x \quad \langle s_i, \mathcal{T} \cup \{\langle s, tos\rangle\}\rangle \vdash_f x : \tau_i \text{ for each } s_i \in \text{next}(s)}{\langle s, \mathcal{T}\rangle \vdash_f tos : \bigsqcup \tau_i}\text{fSG2}$$

$$\frac{s \in \langle P, pc\rangle :: \ldots \quad \langle s, y\rangle \notin \mathcal{T} \quad P_{pc} \in \{\mathsf{LG}\ x, \mathsf{SG}\ x\} \quad x \neq y \quad \langle s_i, \mathcal{T} \cup \{\langle s, y\rangle\}\rangle \vdash_f y : \tau_i \text{ for each } s_i \in \text{next}(s)}{\langle s, \mathcal{T}\rangle \vdash_f y : \bigsqcup \tau_i}\text{fLG2/SG3}$$

$$\frac{s = \langle P, pc\rangle :: \ldots \quad \langle s, f\rangle \notin \mathcal{T} \quad P_{pc} = \mathsf{CF}\ f \quad \langle s_i, \mathcal{T} \cup \{\langle s, f\rangle\}\rangle \vdash_f f : \tau_i \text{ for each } s_i \in \text{next}(s)}{\langle s, \mathcal{T}\rangle \vdash_f f : \bigsqcup(\tau_i \sqcap \mathsf{Fn})}\text{fCF1}$$

$$\frac{s = \langle P, pc\rangle :: \ldots \quad \langle s, u\rangle \notin \mathcal{T} \quad P_{pc} = \mathsf{CF}\ f \quad u \neq f \quad \langle s_i, \mathcal{T} \cup \{\langle s, u\rangle\}\rangle \vdash_f u : \tau_i \text{ for each } s_i \in \text{next}(s)}{\langle s, \mathcal{T}\rangle \vdash_f u : \bigsqcup \tau_i}\text{fCF2}$$

$$\frac{s = \langle P, pc\rangle :: \ldots \quad \langle s, u\rangle \notin \mathcal{T} \quad P_{pc} = \mathsf{JP}\ n \quad \langle s_i, \mathcal{T} \cup \{\langle s, u\rangle\}\rangle \vdash_f u : \tau_i \text{ for each } s_i \in \text{next}(s)}{\langle s, \mathcal{T}\rangle \vdash_f u : \bigsqcup \tau_i}\text{fJP}$$

$$\frac{s = \langle P, pc\rangle :: \ldots \quad \langle s, x\rangle \notin \mathcal{T} \quad P_{pc} \in \{\mathsf{LC}\ c, \mathsf{JIF}\ n, \mathsf{intOp}, \mathsf{strOp}, \mathsf{isInst}\ \tau\} \quad \langle s_i, \mathcal{T} \cup \{\langle s, x\rangle\}\rangle \vdash_f x : \tau_i \text{ for each } s_i \in \text{next}(s)}{\langle s, \mathcal{T}\rangle \vdash_f x : \bigsqcup \tau_i}\text{f*}$$

Figure 4.6: Inference rules for the $\vdash_f$ judgement.

the number of equivalence classes of types is no longer finite, the termination proof in the next section would still hold.

In the previous section we showed an example, where we tried to infer present types but ran into a cycle. We shall now use the rules defined in Figure 4.5 and Figure 4.6 to derive the future use type of a variable f at $\langle M, 0\rangle$ for program M:

$$M = [\underset{0}{\mathsf{CF}\ f}; \underset{1}{\mathsf{LG}\ f}; \underset{2}{\mathsf{intOp}}; \underset{3}{\mathsf{RET}}]$$

Since there is no value that can be used both as a function and as an integer, we expect that the future use type of f be $\bot$. We proceed to build the proof tree, which starts with the judgement $\langle\langle M, 0\rangle, \mathcal{T}_\emptyset\rangle \vdash_f f : \bot$. We omit the control flow graph in this example, as there is no branching. This is simply a sequence of ascending execution points.

$$\dfrac{\dfrac{\dfrac{}{\langle\langle M,3\rangle,\{..,\langle\langle M,2\rangle,f\rangle\}\rangle \vdash_f f:\top}\,\mathsf{fEND}}{\langle\langle M,2\rangle,\{..,\langle\langle M,1\rangle,f\rangle\}\rangle \vdash_f f:\top}\,\mathsf{f^*} \quad \dfrac{}{\langle\langle M,2\rangle,\{..,\langle\langle M,1\rangle,f\rangle\}\rangle \vdash_f tos:\mathsf{Int}}\,\mathsf{fINT}}{\langle\langle M,1\rangle,\{\langle\langle M,0\rangle,f\rangle\}\rangle \vdash_f f:\mathsf{Int}}\,\mathsf{fLG1}}{\langle\langle M,0\rangle,\mathcal{T}_\emptyset\rangle \vdash_f f:\bot}\,\mathsf{fCF1}$$

We note that as we go up the proof tree, the trail is updated with the last element derived in the tree. For presentation purposes, we omit the previous elements from the trail. Note that the meet operation is used in the conclusion of fCF1 on Fn and Int, and in the conclusion of fLG1 on Int and $\top$.

In the following sections we show that the type inference rules are correct and terminating.

4.5 Termination of type inference algorithm

Our first theorem states that the application of the type inference for a finite program produces a finite proof tree. Since $\vdash_p$ and $\vdash_f$ are defined recursively, this property is not trivial.

Theorem 4.1 (Termination)**.** *The rules for judgements $\vdash_p$ and $\vdash_f$ produce finite proof trees.*

Proof. We assume that there exists some infinite proof of $\langle s, \mathcal{T}\rangle \vdash_p u : \tau$ or $\langle s, \mathcal{T}\rangle \vdash_f u : \tau$ and show that this leads to a contradiction. None of the rules in Figures 4.3 – 4.6 have infinite branching since $\mathrm{prev}(s)$ and $\mathrm{next}\ (s)$ return a finite number of truncated call stacks. By König's lemma [62], an infinite tree with finite branching has to have an infinite path, which we call Π. We proceed by analysing the structure of Π.

Clearly, Π cannot contain any of the axioms in Figure 4.3 and Figure 4.5. All other inference rules in Figure 4.4 and Figure 4.6 inductively use either $\vdash_p$ or $\vdash_f$ judgements. We note however that for any inductive use of any of the aforementioned rules, an element is added to the trail. For example, in rule p*, $\langle s, u\rangle$ is added to trail $\mathcal{T}$. This element $\langle s, u\rangle$ is not present in the trail, as one of the preconditions of this rule is $\langle s, u\rangle \notin \mathcal{T}$. Elements are never removed from the trail but always inserted. This means that the size of the trail along Π is strictly increasing.

Since the trail is a finite set, with the maximal trail being $\mathcal{T}_U$, the size of the trail along all paths must be bounded by $|\mathcal{T}_U|$. This contradicts the property that Π has strictly increasing trails. Therefore there can be no infinite path and hence no infinite proof tree produced by the rules for judgements $\vdash_p$ and $\vdash_f$. □

4.6 Soundness for present types

The notion of soundness for present types is relatively straightforward. Given a derivation $\langle s, \mathcal{T}_\emptyset \rangle \vdash_p u : \tau$, we expect that the actual runtime type of the constant u after executing the current instruction in s to be a subtype of τ. This is formally expressed in the next theorem.

Theorem 4.2. *Consider the uninitialised state $\langle \Sigma_0, \varepsilon \rangle$, which is executed n steps to yield an environment Σ and a call stack S, i.e.,*

$$\langle \Sigma_0, \varepsilon \rangle \xrightarrow{n} \langle \Sigma, S \rangle \tag{4.3}$$

Suppose that this state is executed in a single step, $\langle \Sigma, S \rangle \rightarrow \langle \Sigma', S' \rangle$, to yield another state $\langle \Sigma', S' \rangle$. Let the inferred type τ_p of variable u be obtained by the judgement $\langle s, \mathcal{T}_\emptyset \rangle \vdash_p u : \tau_p$, where s is a finite truncation of a stack S and $\mathcal{T}_\emptyset$ is the empty trail, and let τ_r be the runtime type, i.e., $\Sigma'(u) : \tau_r$.

Then, the inferred type is an over-approximation of the runtime type, i.e.,

$$\tau_r <: \tau_p \tag{4.4}$$

Proof. We proceed by induction on n.

Base case. For $n = 0$, we have $\langle \Sigma, S \rangle = \langle \Sigma_0, \varepsilon \rangle$, and hence $s = \varepsilon$. Since $\langle \Sigma_0, \varepsilon \rangle$ can only reduce to $\langle \Sigma_I, \langle M, 0 \rangle :: \varepsilon \rangle$ in a single step, then $\langle \Sigma', S' \rangle = \langle \Sigma_I, \langle M, 0 \rangle :: \varepsilon \rangle$. Therefore, we need to show that (4.4) holds if we obtain our inferred type τ_p from $\langle \varepsilon, \mathcal{T}_\emptyset \rangle \vdash_p u : \tau_p$ and our runtime type τ_r from $\Sigma_I(u) : \tau_r$. For $\langle \varepsilon, \mathcal{T}_\emptyset \rangle \vdash_p u : \tau_p$ the only possible rule used here is pINIT. By this rule, we conclude that $\tau_p = \tau_r$. This means that (4.4) holds since the subtype operator is reflexive.

Inductive case. We assume that the claim holds for some $n > 0$, i.e., when the inferred type τ_p of any variable u is obtained by the judgement $\langle s, \mathcal{T}_\emptyset \rangle \vdash_p u : \tau_p$ and the runtime type τ_r is obtained by the judgement $\Sigma'(u) : \tau_r$.

We now show that the claim also holds for $n+1$. In particular, we consider the situation when the program $\langle \Sigma', S' \rangle$ is executed a further step, i.e., $\langle \Sigma', S' \rangle \rightarrow \langle \Sigma'', S'' \rangle$. In this case we obtain the inferred type using the judgement $\langle s', \mathcal{T}_\emptyset \rangle \vdash_p u : \tau'_p$ and the runtime type using the judgement $\Sigma''(u) : \tau'_r$. We show that

$$\tau'_r <: \tau'_p \tag{4.5}$$

by analysing all applicable preconditions and patterns for the $\vdash_p$ judgement.

We start by noting that the pINIT rule is not applicable because this relies on the call stack being empty. As $n > 0$, we know that s' is non-empty because there is no step that yields an empty

stack, i.e., $\langle \Sigma, S \rangle \not\rightarrow \langle \Sigma', \varepsilon \rangle$. We therefore do not need to consider these cases. pRAISE is also not applicable because we cannot execute another step after reducing to a TypeError.

The pTRAIL rule is also not applicable because its precondition requires the variable u for execution point s' to be in the trail. Since $\mathcal{T}_\emptyset$ is empty, we have that if $\langle s', u \rangle \notin \mathcal{T}_\emptyset$. For the remaining rules, we can assume that the corresponding preconditions on the trail hold.

We start by analysing the cases that apply to the remaining axioms shown in Figure 4.3. We only show the case for pLC, however the remaining cases all follow the same pattern.

Case pLC, i.e., u is tos, s' has the form $\langle P', pc' \rangle :: ...$ and $P'_{pc'}$ is LC c..

In this case, the inferred type τ'_p is obtained using the judgement $\langle s', \mathcal{T}_\emptyset \rangle \vdash_p tos : \tau'_p$. By pLC, τ'_p is such that $c : \tau'_p$. In particular, τ'_p is a primitive type, i.e., not a union type. To get the runtime type we consider the semantics of μPython when u is tos, s' has the form $\langle P', pc' \rangle :: ...$ and $P'_{pc'}$ is LC c.. From this we see that $\Sigma'(tos)$ is c and so τ'_r is such that $c : \tau'_r$. From this, and the fact that τ'_p is a primitive type, we can immediately conclude that $\tau'_p = \tau'_r$ and thus (4.5) holds as required.

We now focus on the recursive rules shown in Figure 4.4. The proofs for these cases follow the same pattern and we only elaborate on one case.

Case pLG1. u is tos, s' has the form $\langle P', pc' \rangle :: ...$ and $P'_{pc'}$ is LG x..

In this case, the inferred type τ'_p is obtained using the judgement $\langle s', \mathcal{T}_\emptyset \rangle \vdash_p tos : \tau'_p$.

By pLG1, we have $\tau'_p = \bigsqcup \tau_i$ such that:

$$\langle s_i, \mathcal{T} \cup \{\langle s', tos \rangle\} \rangle \vdash_p x : \tau_i \text{ for each } s_i \in \text{prev}(s')$$

for all $s_i \in \text{prev}(s')$. From (4.1), we know that at least one of the truncated call stacks returned by prev is a truncation of the runtime call stack at the previous runtime step. Hence $s \in \text{prev}(s')$. Let τ_p, τ''_p be such that

$$\langle s, \mathcal{T}_\emptyset \rangle \vdash_p x : \tau_p \quad \text{and} \quad \langle s, \mathcal{T}_\emptyset \cup \{\langle s', tos \rangle\} \rangle \vdash_p x : \tau''_p$$

Since τ''_p must be one of the τ_i joined together to compute τ'_p, we know that $\tau''_p <: \tau'_p$ and since $\mathcal{T}_\emptyset \subseteq \mathcal{T}_\emptyset$, we use Lemma 4.3 to conclude that

$$\tau_p <: \tau'_p$$

Recall that our assumption in the inductive hypothesis states that

$$\tau_r <: \tau_p$$

where τ_r is such that $\Sigma(x) : \tau_r$. By using this assumption, our previous conclusion $\tau_p <: \tau'_p$ and by exploiting the transitivity property of the subtype operator, we know that

$$\tau_r <: \tau'_p$$

We now refer to the semantics of μPython in Figure 3.4 for this case. From this we know $\Sigma(x)$ is $\Sigma'(tos)$. Hence, $\tau_r = \tau'_r$ and therefore we conclude that $\tau'_r <: \tau'_p$ as required. □

The previous proof depends on the next two lemmas, which relate judgements with different elements in their respective trails. For instance, consider τ derived using $\langle s, \mathcal{T}\rangle \vdash_p u : \tau$ and τ'', derived using $\langle s, \mathcal{T} \cup \{\langle s', v\rangle\}\rangle \vdash_p u : \tau''$. We intuit that $\tau'' <: \tau$. This is because in the case of the proof tree deriving τ'', the rule pTRAIL is more likely to be applied. Indeed, we prove a stronger property in Lemma 4.4. We now consider τ', derived using $\langle s', \mathcal{T}'\rangle \vdash_p v : \tau'$. In the proof for the previous theorem, and in cases where a type is derived using any of the rules in Figure 4.4, a pattern emerges. In these cases, throughout the proof we see that $\tau' = \tau'' \sqcup \ldots$ (and hence $\tau'' <: \tau'$) but we need to prove that $\tau <: \tau'$. We do so in the next lemma.

Lemma 4.3 (Bounding)**.** *For any variables u, v, execution points s, s', and trails $\mathcal{T}, \mathcal{T}'$ such that $\mathcal{T}' \subseteq \mathcal{T}$. We have that*

$$\begin{aligned} \langle s, \mathcal{T}\rangle &\vdash_p u : \tau \\ \langle s', \mathcal{T}'\rangle &\vdash_p v : \tau' \\ \langle s, \mathcal{T} \cup \{\langle s', v\rangle\}\rangle &\vdash_p u : \tau'' \end{aligned} \tag{4.6}$$

and $\tau'' <: \tau'$, then

$$\tau <: \tau' \tag{4.7}$$

Proof. We proceed by induction on the size n of the set difference between the universal trail $\mathcal{T}_U$ and the actual trail, i.e., $\text{size}(\mathcal{T}_U - \mathcal{T})$. The universal trail is defined as the trail containing all combinations of $\langle s, u\rangle$ for all u, s. Therefore we prove that the above lemma holds for all n.

Base case. We start with $n = 0$, which means that $\text{size}(\mathcal{T}_U - \mathcal{T}) = 0$. Since there is no trail bigger than $\mathcal{T}_U$, $\mathcal{T}$ is the trail $\mathcal{T}_U$. We substitute $\mathcal{T} = \mathcal{T}_U$ into (4.6), and we rewrite our judgements as:

$$\begin{aligned} \langle s, \mathcal{T}_U\rangle &\vdash_p u : \tau \\ \langle s', \mathcal{T}'\rangle &\vdash_p v : \tau' \\ \langle s, \mathcal{T}_U \cup \{\langle s', v\rangle\}\rangle &\vdash_p u : \tau'' \end{aligned}$$

The universal trail contains all possible trail elements. Therefore $\langle s, u\rangle \in \mathcal{T}_U$ and by pTRAIL we conclude that $\tau = \bot$. This means that our claim $\tau <: \tau'$ holds since $\bot$ is a subtype of any type.

Inductive case. We assume that the Lemma holds for some $\text{size}(\mathcal{T}_U - \mathcal{T}) = n$, i.e., for any variables u, v, execution points s, s', and trails $\mathcal{T}, \mathcal{T}'$ such that $\mathcal{T}' \subseteq \mathcal{T}$ and $\text{size}(\mathcal{T}_U - \mathcal{T}) = n$.

If

$$\begin{aligned} \langle s, \mathcal{T} \rangle \vdash_p u : \tau \\ \langle s', \mathcal{T}' \rangle \vdash_p v : \tau' \\ \langle s, \mathcal{T} \cup \{\langle s', v \rangle\} \rangle \vdash_p u : \tau'' \end{aligned} \tag{4.8}$$

and $\tau'' <: \tau'$, then

$$\tau <: \tau' \tag{4.9}$$

We then show that it also holds for $n + 1$. For this, we choose two trail variables $\mathcal{T}''$ and $\mathcal{T}'''$, where $\text{size}(\mathcal{T}_U - \mathcal{T}'') = n + 1$, and $\mathcal{T}''' \subseteq \mathcal{T}''$. This means that the number of elements in $\mathcal{T}''$ is one smaller than the number of elements in $\mathcal{T}$ as defined in (4.8). In particular, we have to show that for any variables u, v, execution points s, s', and trails $\mathcal{T}'', \mathcal{T}'''$ such that $\mathcal{T}''' \subseteq \mathcal{T}''$ and $\text{size}(\mathcal{T}_U - \mathcal{T}'') = n + 1$, and

$$\begin{aligned} \langle s, \mathcal{T}'' \rangle \vdash_p u : \tau \\ \langle s', \mathcal{T}''' \rangle \vdash_p v : \tau' \\ \langle s, \mathcal{T}'' \cup \{\langle s', v \rangle\} \rangle \vdash_p u : \tau'' \end{aligned} \tag{4.10}$$

and $\tau'' <: \tau'$, then

$$\tau <: \tau' \tag{4.11}$$

We proceed by analysing the proof of the judgement $\langle s, \mathcal{T}'' \cup \{\langle s', v \rangle\} \rangle \vdash_p u : \tau''$ by a case analysis on the last rule (see Figure 4.3 and Figure 4.4) used in the proof.

Case pINIT, i.e., looking up a variable in the entry point of the program.

We have $s = \varepsilon$ and we rewrite some of our judgements from (4.10) correspondingly as:

$$\begin{aligned} \langle \varepsilon, \mathcal{T}'' \rangle \vdash_p u : \tau \\ \langle \varepsilon, \mathcal{T}'' \cup \{\langle s', v \rangle\} \rangle \vdash_p u : \tau'' \end{aligned}$$

From the hypothesis of pINIT, we conclude that

$$\begin{aligned} \Sigma_I(u) : \tau \\ \Sigma_I(u) : \tau'' \end{aligned}$$

From this we can see that $\tau = \tau''$, which implies that $\tau <: \tau''$. Since our inductive hypothesis states that $\tau'' <: \tau'$, by transitivity we also have $\tau <: \tau'$ as required.

Case pTRAIL.

In this case, $\langle s, u \rangle \in \mathcal{T}''$, i.e., the variable u for execution point s is already in the trail. If we assume $\langle s, u \rangle$ is not $\langle s', v \rangle$. $\langle s, \mathcal{T}'' \rangle \vdash_p u : \tau$ becomes $\langle s, \mathcal{T}'' \rangle \vdash_p u : \bot$. Since τ is $\bot$, then $\tau <: \tau'$ because $\bot$ is a subtype of any type.

Now, if we assume $\langle s, u\rangle$ is $\langle s', v\rangle$, we rewrite our judgements from (4.10) to:

$$\langle s, \mathcal{T}''\rangle \vdash_p u : \tau$$
$$\langle s, \mathcal{T}'''\rangle \vdash_p u : \tau'$$

Since $\mathcal{T}''' \subseteq \mathcal{T}''$, the original claim $\tau <: \tau'$ holds according to Lemma 4.4.

Case pLC, u is tos, s has the form $\langle P, pc\rangle :: ...$ and P_{pc} is LC c.

We rewrite our judgements from (4.10) to:

$$\langle s, \mathcal{T}''\rangle \vdash_p tos : \tau$$
$$\langle s', \mathcal{T}'''\rangle \vdash_p v : \tau'$$
$$\langle s, \mathcal{T}'' \cup \{\langle s', v\rangle\}\rangle \vdash_p tos : \tau''$$

From the hypothesis of pLC, we conclude that $c : \tau$ and $c : \tau''$. Since τ and τ'' are primitive types, $\tau = \tau''$ and since our hypothesis states that $\tau'' <: \tau'$, then $\tau <: \tau'$ as required.

All other axioms in Figure 4.3 follow the same pattern as this case.

We now look at the recursive rules in Figure 4.4. The proofs for these cases follow the same pattern. We will only look at the case for pLG1 and omit the other cases.

Case pLG1.

In this case u is tos, s has the form $\langle P, pc\rangle :: ...$ and P_{pc} is LG x. and we rewrite our judgements from (4.10) accordingly to:

$$\langle s, \mathcal{T}''\rangle \vdash_p tos : \tau$$
$$\langle s', \mathcal{T}'''\rangle \vdash_p v : \tau'$$
$$\langle s, \mathcal{T}'' \cup \{\langle s', v\rangle\}\rangle \vdash_p tos : \tau''$$

By pLG1 $\tau = \bigsqcup \tau_i$ and $\tau'' = \bigsqcup \tau_i''$ where

$$\langle s_i, \mathcal{T}'' \cup \{\langle s, tos\rangle\}\rangle \vdash_p x : \tau_i$$
$$\langle s_i, \mathcal{T}'' \cup \{\langle s, tos\rangle\} \cup \{\langle s', v\rangle\}\rangle \vdash_p x : \tau_i''$$

for all $s_i \in \text{prev}(s)$.

Let $\mathcal{T}$ be $\mathcal{T}'' \cup \{\langle s, tos\rangle\}$, then we can rewrite the above as

$$\langle s_i, \mathcal{T}\rangle \vdash_p x : \tau_i$$
$$\langle s_i, \mathcal{T} \cup \{\langle s', v\rangle\}\rangle \vdash_p x : \tau_i''$$

for all $s_i \in \text{prev}(s)$.

Note that since $\tau'' = \bigsqcup \tau_i''$, then $\tau_i'' <: \tau''$, and since $\tau'' <: \tau'$ by assumption, then $\tau_i'' <: \tau'$ for each τ_i''. Note also that $\langle s', \mathcal{T}''' \rangle \vdash_p v : \tau'$ as part of our hypothesis. Furthermore note that $\mathcal{T}''' \subseteq \mathcal{T}'' \subseteq \mathcal{T}'' \cup \{\langle s, tos \rangle\} = \mathcal{T}$. The hypothesis in (4.8) all hold and $\text{size}(\mathcal{T}_U - \mathcal{T}) = n$ so by the inductive hypothesis, $\tau_i <: \tau'$ for each τ_i. Therefore $\tau = \bigsqcup \tau_i <: \tau'$ as required.

All other recursive cases follow the same pattern. □

The next lemma states that with fewer elements in a trail we get a more general type.

Lemma 4.4. *For any variable u, execution point s and trails $\mathcal{T}$, $\mathcal{T}'$ such that $\mathcal{T}' \subseteq \mathcal{T}$, $\tau <: \tau'$ where*

$$\begin{aligned} \langle s, \mathcal{T} \rangle &\vdash_p u : \tau \\ \langle s, \mathcal{T}' \rangle &\vdash_p u : \tau' \end{aligned} \tag{4.12}$$

Proof. We proceed by induction on n, which we define as the size of the set difference between the universal trail $\mathcal{T}_U$ and the actual trail, i.e., $\text{size}(\mathcal{T}_U - \mathcal{T})$. Therefore we prove that the above lemma holds for all n.

Base case. We start with $n = 0$ so that $\mathcal{T}$ is the universal trail $\mathcal{T}_U$.

We substitute $\mathcal{T}$ with $\mathcal{T}_U$ in the judgements (4.12):

$$\begin{aligned} \langle s, \mathcal{T}_U \rangle &\vdash_p u : \tau \\ \langle s, \mathcal{T}' \rangle &\vdash_p u : \tau' \end{aligned}$$

Since $\langle s, u \rangle \in \mathcal{T}_U$, by pTRAIL we can conclude that $\tau = \bot$. Hence our claim $\tau <: \tau'$ holds as required.

Inductive case. We assume that the Lemma holds for some $\text{size}(\mathcal{T}_U - \mathcal{T}) = n$, i.e., that for all variables u, execution points s and trails $\mathcal{T}$, $\mathcal{T}'$ such that $\mathcal{T}' \subseteq \mathcal{T}$, $\tau <: \tau'$ where

$$\begin{aligned} \langle s, \mathcal{T} \rangle &\vdash_p u : \tau \\ \langle s, \mathcal{T}' \rangle &\vdash_p u : \tau' \end{aligned} \tag{4.13}$$

We now show that it also holds for $n + 1$. For this we choose trail variable $\mathcal{T}''$, where $\text{size}(\mathcal{T}_U - \mathcal{T}'') = n + 1$ and some $\mathcal{T}'''$ such that $\mathcal{T}''' \subseteq \mathcal{T}''$. In particular, we show that for any variables u, v, execution points s, s', and trails $\mathcal{T}'', \mathcal{T}'''$ such that $\mathcal{T}''' \subseteq \mathcal{T}''$ and $\text{size}(\mathcal{T}_U - \mathcal{T}'') = n + 1$, $\tau <: \tau'$ where

$$\begin{aligned} \langle s, \mathcal{T}'' \rangle &\vdash_p u : \tau \\ \langle s, \mathcal{T}''' \rangle &\vdash_p u : \tau' \end{aligned} \tag{4.14}$$

We proceed by analysing the last rule used in the proof of $\langle s, \mathcal{T}''' \rangle \vdash_p u : \tau'$

Case pINIT, i.e., looking up a variable in the entry point of the program.

In this case $s = \varepsilon$, and we can therefore rewrite (4.14) as:

$$\langle \varepsilon, \mathcal{T}'' \rangle \vdash_p u : \tau$$
$$\langle \varepsilon, \mathcal{T}''' \rangle \vdash_p u : \tau'$$

From pINIT, we see that $\Sigma_I(u) : \tau$ and $\Sigma_I(u) : \tau'$ hold. Both τ and τ' are primitive types and hence $\tau <: \tau'$ as required.

Case pTRAIL, $\langle s, u \rangle \in \mathcal{T}''$, i.e., the variable u for execution point s is already in the trail, and hence

τ is $\bot$. Therefore $\tau <: \tau'$ as required.

For the remaining cases we assume that $\langle s, u \rangle \notin \mathcal{T}''$ and since $\mathcal{T}''' \subseteq \mathcal{T}''$, we also have $\langle s, u \rangle \notin \mathcal{T}'''$.

The proofs for the cases that match the remaining rules in Figure 4.3 follow the same pattern. We only elaborate the case that matches rule pLC.

Case pLC, i.e., u is tos, s has the form $\langle P, pc \rangle :: ...$ and P_{pc} is LC c..

We rewrite our judgements from (4.14) as

$$\langle s, \mathcal{T}'' \rangle \vdash_p tos : \tau$$
$$\langle s, \mathcal{T}''' \rangle \vdash_p tos : \tau'$$

In this case we see that rule pLC tells us that $c : \tau$ and $c : \tau'$. Since τ and τ' are primitive, $\tau <: \tau'$ as required.

The proofs for the cases that match the recursive rules in Figure 4.4, all follow the same pattern. We only elaborate the case that matches rule pLG1.

Case pLG1, i.e., u is tos, s has the form $\langle P, pc \rangle :: ...$ and P_{pc} is LG x..

We rewrite our judgements from (4.14) as

$$\langle s, \mathcal{T}'' \rangle \vdash_p tos : \tau$$
$$\langle s, \mathcal{T}''' \rangle \vdash_p tos : \tau'$$

By pLG1 $\tau = \bigsqcup \tau_i$ and $\tau'' = \bigsqcup \tau_i''$ where

$$\begin{aligned} \langle s_i, \mathcal{T}'' \cup \{\langle s, tos \rangle\} \rangle \vdash_p x : \tau_i \\ \langle s_i, \mathcal{T}''' \cup \{\langle s, tos \rangle\} \rangle \vdash_p x : \tau_i' \end{aligned} \tag{4.15}$$

for $s_i \in \text{prev}(s)$.

Since $\tau = \bigsqcup \tau_i$ and $\tau' = \bigsqcup \tau_i'$, we show $\tau <: \tau'$ by showing that $\tau_i <: \tau_i'$ for all τ_i.

Let $\mathcal{T}$ be $\mathcal{T}'' \cup \{\langle s, tos\rangle\}$ and $\mathcal{T}'$ be $\mathcal{T}''' \cup \{\langle s, tos\rangle\}$. Then by rewriting (4.15) we have the hypothesis (4.13), where $\text{size}(\mathcal{T}_U - \mathcal{T}) = n$.

By the inductive hypothesis we have $\tau_i <: \tau_i'$ for all τ_i, τ_i' as required. □

4.7 Soundness for future use types

The correctness criteria for future use types are more subtle. The future use types describe constraints on the future uses of a variable and we will use these constraints to report type errors preemptively by raising type error exceptions. So, correctness in this case means that, supposing we execute the program under a preemptively type checked semantics, if we raise a type error exception then the same program running in the unchecked semantics would continue executing to reach an actual type error. In addition, we must also allow for the possibility that the program in the non-preemptive semantics could diverge before reaching the detected future error.

In order to formalise the above, we need to define the preemptively type checked semantics and a predicate on states that holds whenever a future divergence or type error is guaranteed. We begin by defining the diverge-error predicate coinductively:

Definition 4.5. A predicate $R^{\Uparrow}$ on $\langle \Sigma, S\rangle$ is called a diverge-error predicate if whenever $\langle \Sigma, S\rangle \in R^{\Uparrow}$ then $\langle \Sigma, S\rangle \to \langle \Sigma', S'\rangle \wedge \langle \Sigma', S'\rangle \in R^{\Uparrow}$ or $\langle \Sigma, S\rangle \to \mathsf{TypeError}$.

Let $\Uparrow$ be the largest diverge-error predicate. It follows that a state that is in a diverge-error predicate cannot reach the state End or Exception.

We now define another predicate over $\langle \Sigma, S\rangle$, which holds whenever the future use types for all variables subsumes the actual runtime type.

Definition 4.6. The state compatibility predicate $StateComp$ on $\langle \Sigma, S\rangle$ holds if for all variables u, the current runtime type of u is a subtype of the inferred future type for the execution point s corresponding to stack S, i.e.,

$$\Sigma(u) : \tau_r$$
$$\langle s, \mathcal{T}_\emptyset\rangle \vdash_f u : \tau_f$$

and $\tau_r <: \tau_f$.

The next theorem demonstrates that this simple predicate is sufficient for preemptive type checking. In principle, if we had to implement a runtime environment that implements a weak version of preemptive type checking, this could check whether $StateComp$ holds at every execution step. Obviously, this is computationally expensive. We will see in the next chapter that $StateComp$ can be refined to make better use of static type information.

Theorem 4.7. *If* $\langle \Sigma, S\rangle \notin StateComp$, *then* $\langle \Sigma, S\rangle \in \Uparrow$.

Proof. We use coinduction here by proving that the complement of $StateComp$ is itself a diverge-error predicate, i.e., if $\langle \Sigma, S \rangle \notin StateComp$ then:

- Either $\langle \Sigma, S \rangle \rightarrow \mathsf{TypeError}$.
- Or $\langle \Sigma, S \rangle \rightarrow \langle \Sigma', S' \rangle$ and $\langle \Sigma', S' \rangle \notin StateComp$.

If $\langle \Sigma, S \rangle \notin StateComp$, then there is a variable u for which its runtime type is not a subtype of its future use type, i.e., $\tau_r \not<: \tau_f$ such that

$$\frac{\Sigma(u) : \tau_r \qquad \langle s, \mathcal{T}_\emptyset \rangle \vdash_f u : \tau_f}{} \tag{4.16}$$

We choose this u and consider the last rule used to infer the above $\vdash_f$ judgement. We start by looking at the axioms (see Figure 4.5).

In cases matching rules fSET, fEND, fRAISE and fSG1, we conclude that $\tau_f = \top$, which means that

$$\tau_r \not<: \tau_f$$

cannot hold as there is no type τ_r such that $\tau_r \not<: \top$, and hence these cases cannot arise.

Another case that we do not need to consider is fTRAIL as this rule only applies to a non-empty trail.

We now examine the remaining axioms, namely fJIF, fSTR and fINT. These follow the same pattern, so we will use the case matching rule fJIF as an example.

Case fJIF, i.e., u is tos, s has the form $\langle P, pc \rangle :: ...$ and P_{pc} is JIF n..

In this case, $\tau_f = \mathsf{Bool}$. Therefore $\tau_r \not<: \tau_f$ means that the type τ_r of the value held in tos before the currently executing instruction is not a subtype of Bool. This means that $\neg \tau_r : \mathsf{Bool}$. From the semantics of μPython (see Figure 3.4), if we execute the current instruction (a conditional jump) and tos is not of type Bool, we get a type error, i.e.,

$$\langle \Sigma, S \rangle \rightarrow \mathsf{TypeError}$$

as required.

We now proceed to analyse the recursive rules in Figure 4.6. All cases, except fLG1 and fCF1 which we shall tackle later, follow the same pattern. We therefore elaborate the case for fJP and omit the other cases.

Case fJP, i.e., s has the form $\langle P, pc \rangle :: ...$ and P_{pc} is JP n

We know from the reduction semantics that a unique $\langle \Sigma', S' \rangle$ state exists such that $\langle \Sigma, S \rangle \rightarrow \langle \Sigma', S' \rangle$, so it suffices to show that $\langle \Sigma', S' \rangle \notin StateComp$ for this $\langle \Sigma', S' \rangle$.

The inferred type τ_f is obtained using the judgement $\langle s, \mathcal{T}_\emptyset \rangle \vdash_f u : \tau_f$.

By fJP, $\tau_f = \bigsqcup \tau_i$ such that:

$$\langle s_i, \mathcal{T} \cup \{\langle s, u \rangle\} \rangle \vdash_f u : \tau_i \text{ for each } s_i \in \text{next}(s)$$

From our initial proof condition (4.2), we know that at least one of the execution points returned by $\text{next}(s)$ is a truncation of the runtime call stack S'. Hence $s' \in \text{next}(s)$.

Let τ''_f be such that $\langle s', \mathcal{T}_\emptyset \cup \{\langle s, u \rangle\} \rangle \vdash_f u : \tau''_f$. Since $\mathcal{T}_\emptyset \subseteq \mathcal{T}_\emptyset$, we use our result from Lemma 4.8 below and conclude that

$$\tau'_f <: \tau_f \sqcup \tau''_f$$

where τ'_f is such that $\langle s', \mathcal{T}_\emptyset \rangle \vdash_f u : \tau'_f$.

Since τ''_f is one of the types joined together to compute τ_f, we know that $\tau''_f <: \tau_f$, and we can therefore rewrite the previous relation as:

$$\tau'_f <: \tau_f$$

We combine this with the hypothesis $\tau_r \not<: \tau_f$, to see that $\tau_r \not<: \tau'_f$.

It only remains to consider the runtime type of u in Σ'. According to the semantics of μPython (see Figure 3.4) for this case $\Sigma'(u)$ is simply τ_r and so we can conclude that $\langle \Sigma', S' \rangle \notin \mathit{StateComp}$ as required.

The proof for cases fLG1 and fCF1 are more intricate. These also follow similar patterns, so we will look at the case for fLG1.

Case fLG1, i.e., u is x, s has the form $\langle P, pc \rangle :: \ldots$ and P_{pc} is LG x.

Again, $\langle \Sigma', S' \rangle$ exists and is unique so we choose this and prove $\langle \Sigma', S' \rangle \notin \mathit{StateComp}$. The inferred type τ_f is obtained using the judgement $\langle s, \mathcal{T}_\emptyset \rangle \vdash_f x : \tau_f$. By fLG1, $\tau_f = \bigsqcup \upsilon_i \sqcap \nu_i$ such that:

$$\langle s_i, \mathcal{T} \cup \{\langle s, x \rangle\} \rangle \vdash_f \mathit{tos} : \upsilon_i \text{ for each } s_i \in \text{next}(s)$$
$$\langle s_i, \mathcal{T} \cup \{\langle s, x \rangle\} \rangle \vdash_f x : \nu_i \text{ for each } s_i \in \text{next}(s)$$

From our initial proof condition (4.2), we know that at least one of the execution points returned by $\text{next}(s)$ is a truncation of the runtime call stack S'.

Hence $s' \in \text{next}(s)$.

Let υ'' and ν'' be such that

$$\langle s', \mathcal{T}_\emptyset \cup \{\langle s, x\rangle\}\rangle \vdash_f tos : \upsilon''$$
$$\langle s', \mathcal{T}_\emptyset \cup \{\langle s, x\rangle\}\rangle \vdash_f x : \nu''$$

Given that $\mathcal{T}_\emptyset \subseteq \mathcal{T}_\emptyset$, we use our result from Lemma 4.8 and conclude that

$$\upsilon' <: \tau_f \sqcup \upsilon'' \tag{4.17}$$
$$\nu' <: \tau_f \sqcup \nu'' \tag{4.18}$$

where υ' and ν' are such that

$$\langle s', \mathcal{T}_\emptyset\rangle \vdash_f tos : \upsilon'$$
$$\langle s', \mathcal{T}_\emptyset\rangle \vdash_f x : \nu'$$

We combine (4.17) and (4.18) into

$$\upsilon' \sqcap \nu' <: (\tau_f \sqcup \upsilon'') \sqcap (\tau_f \sqcup \nu'')$$

which we can rearrange as

$$\upsilon' \sqcap \nu' <: (\upsilon'' \sqcap \nu'') \sqcup \tau_f$$

Since $(\upsilon'' \sqcap \nu'')$ is one of the types joined together to compute τ_f, we know that $(\upsilon'' \sqcap \nu'') <: \tau_f$, and we can therefore rewrite the previous relation as:

$$(\upsilon' \sqcap \nu') <: \tau_f$$

We combine this result with $\tau_r \not<: \tau_f$, as stated in the hypothesis and conclude that $\tau_r \not<: (\upsilon' \sqcap \nu')$. Essentially this means that at least one of the following holds:

$$\tau_r \not<: \upsilon'$$
$$\tau_r \not<: \nu'$$

From the μPython semantics for this case, we can conclude that $\Sigma'(x)$ and $\Sigma'(tos)$ are the same as $\Sigma(x)$ by executing a single step. Therefore τ_r is also such that

$$\Sigma'(tos) : \tau_r$$
$$\Sigma'(x) : \tau_r$$

Since $\tau_r \not<: \upsilon'$ or $\tau_r \not<: \nu'$, the runtime type of tos or the runtime type of x is not a subtype of its future use type.

We therefore conclude that $\langle \Sigma', S' \rangle \notin StateComp$ as required. □

The next lemma is the future use types equivalent of Lemma 4.3. This lemma is used to relate the type derived using a $\vdash_f$ judgement with a trail $\mathcal{T}$ to the type derived using a $\vdash_f$ judgement with a trail that has an additional element to $\mathcal{T}$. For instance, consider τ derived using $\langle s, \mathcal{T} \rangle \vdash_f u : \tau$, τ' derived using the judgement $\langle s', \mathcal{T} \rangle \vdash_f u : \tau'$ and τ'', derived using $\langle s, \mathcal{T} \cup \{\langle s', v \rangle\} \rangle \vdash_f u : \tau''$. As in the case for present types, we know that $\tau'' <: \tau'$. However, in the proof for the previous theorem, and in cases where a type is derived using any of the rules in Figure 4.6, a pattern emerges. In these cases, we get to a stage where $\tau = \tau'' \sqcup \ldots$ and hence $\tau = \tau \sqcup \tau'' \sqcup \ldots$, but we need to prove that $\tau' <: \tau \sqcup \tau''$. We do so in the next lemma, and this shows that when one type derivation τ depends on a premise consisting of other type derivations τ'', then not only is τ more general than τ'', but it is also more general than τ', i.e., τ'' derived the topmost element of the trail removed.

Lemma 4.8 (Bounding)**.** *For any variables u, v, execution points s, s', and trails $\mathcal{T}, \mathcal{T}'$ such that $\mathcal{T}' \subseteq \mathcal{T}$, then $\tau' <: \tau \sqcup \tau''$ where*

$$\begin{aligned} \langle s, \mathcal{T}' \rangle &\vdash_f v : \tau \\ \langle s', \mathcal{T} \rangle &\vdash_f u : \tau' \\ \langle s', \mathcal{T} \cup \{\langle s, v \rangle\} \rangle &\vdash_f u : \tau'' \end{aligned} \tag{4.19}$$

Proof. We proceed by induction on n, where as in Lemma 4.3, this is defined as the size of the set difference between the universal trail $\mathcal{T}_U$ and the actual trail, i.e., $\text{size}(\mathcal{T}_U - \mathcal{T})$. Therefore we prove that the above lemma holds for all n.

Base case. We start by proving the lemma holds for $n = 0$, which means that $\text{size}(\mathcal{T}_U - \mathcal{T}) = 0$. This means that $\mathcal{T} = \mathcal{T}_U$ and that $\langle s, u \rangle \in \mathcal{T}_U$ since the universal trail contains all possible trail elements. By fTRAIL we conclude that $\tau' = \bot$ and therefore $\tau' <: \tau \sqcup \tau''$ as required.

Inductive case. We assume that the Lemma holds for $\text{size}(\mathcal{T}_U - \mathcal{T}) = n$, i.e., that for all variables u, v, execution point s, s', and trails $\mathcal{T}, \mathcal{T}'$ such that $\mathcal{T}' \subseteq \mathcal{T}$, $\tau' <: \tau \sqcup \tau''$ where

$$\begin{aligned} \langle s, \mathcal{T}' \rangle &\vdash_f v : \tau \\ \langle s', \mathcal{T} \rangle &\vdash_f u : \tau' \\ \langle s', \mathcal{T} \cup \{\langle s, v \rangle\} \rangle &\vdash_f u : \tau'' \end{aligned} \tag{4.20}$$

We then show that it also holds for $n + 1$. For this, we choose two trail variables $\mathcal{T}''$ and $\mathcal{T}'''$, where $\text{size}(\mathcal{T}_U - \mathcal{T}'') = n + 1$, and $\mathcal{T}''' \subseteq \mathcal{T}''$. In particular, we have to show that for any variables u, v, execution points s, s', and trails $\mathcal{T}'', \mathcal{T}'''$ such that $\mathcal{T}''' \subseteq \mathcal{T}''$ and $\text{size}(\mathcal{T}_U - \mathcal{T}'') =$

$n+1$, $\tau' <: \tau \sqcup \tau''$ where

$$\begin{aligned} \langle s, \mathcal{T}''' \rangle &\vdash_f v : \tau \\ \langle s', \mathcal{T}'' \rangle &\vdash_f u : \tau' \\ \langle s', \mathcal{T}'' \cup \{\langle s, v \rangle\} \rangle &\vdash_f u : \tau'' \end{aligned} \tag{4.21}$$

We proceed by analysing the last rule used to establish the judgement $\langle s', \mathcal{T}'' \cup \{\langle s, v \rangle\} \rangle \vdash_f u : \tau''$.

It happens that most of these cases have a similar pattern to the cases in the proof of Lemma 4.4. We shall therefore only cover the most difficult cases in this proof.

The cases for fTRAIL follow a similar pattern to the cases for pTRAIL in Lemma 4.4. This means that we assume that in the following cases, $\langle s, v \rangle \neq \langle s', u \rangle$ and $\langle s', u \rangle \notin \mathcal{T}''$.

Cases that match rules fSET, fEND, fINIT and fRAISE follow the same pattern so we look at only one example.

Case fEND, i.e., $s' = \langle P, pc \rangle :: \varepsilon$ and $P_{pc} = \mathsf{RET}$

In this case τ'' is defined such that

$$\langle \langle P, pc \rangle :: \varepsilon, \mathcal{T}'' \cup \{\langle s, v \rangle\} \rangle \vdash_f u : \tau''$$

By fEND we conclude that τ'' is $\top$ and therefore $\tau' <: \tau \sqcup \tau''$ as required.

Cases that match rules fJIF fSTR and fINT follow the same pattern, so we only look at one case.

Case fJIF, i.e., u is tos, s' has the form $\langle P', pc' \rangle :: \ldots$ and $P'_{pc'}$ is JIF n.

In this case τ' is defined such that

$$\langle s', \mathcal{T}'' \rangle \vdash_f tos : \tau'$$

and τ'' is defined such that

$$\langle s', \mathcal{T}'' \cup \{\langle s, v \rangle\} \rangle \vdash_f tos : \tau''$$

By fJIF we conclude that $\tau' = \mathsf{Bool}$ and $\tau'' = \mathsf{Bool}$. From this, we easily conclude that $\tau' <: \tau \sqcup \tau''$ holds as required.

The rest of the cases match the recursive rules in Figure 4.6. The proofs for these cases are all similar to each other, with the most intricate being the case that matches fLG1.

Case fLG1, i.e., u is x, s' has the form $\langle P', pc' \rangle :: \ldots$ and $P'_{pc'}$ is LG x.

We rewrite our judgements from (4.21) into

$$\begin{aligned}\langle s, \mathcal{T}''' \rangle \vdash_f v : \tau \\ \langle s', \mathcal{T}'' \rangle \vdash_f x : \tau' \\ \langle s', \mathcal{T}'' \cup \{\langle s, v \rangle\} \rangle \vdash_f x : \tau''\end{aligned}$$

By fLG1, $\tau' = \bigsqcup(v_i' \sqcap \nu_i')$ and $\tau'' = \bigsqcup(v_i'' \sqcap \nu_i'')$, where v_i', ν_i', v_i'' and ν_i'' are defined as follows

$$\begin{aligned}\langle s_i', \mathcal{T}'' \cup \{\langle s', x \rangle\} \rangle \vdash_f tos : v_i' \\ \langle s_i', \mathcal{T}'' \cup \{\langle s', x \rangle\} \rangle \vdash_f x : \nu_i' \\ \langle s_i', \mathcal{T}'' \cup \{\langle s', x \rangle\} \cup \{\langle s, v \rangle\} \rangle \vdash_f tos : v_i'' \\ \langle s_i', \mathcal{T}'' \cup \{\langle s', x \rangle\} \cup \{\langle s, v \rangle\} \rangle \vdash_f x : \nu_i''\end{aligned}$$

Let $\mathcal{T}$ be $\mathcal{T}'' \cup \{\langle s', x \rangle\}$. We rewrite the above to be

$$\begin{aligned}\langle s_i', \mathcal{T} \rangle \vdash_f tos : v_i' \\ \langle s_i', \mathcal{T} \rangle \vdash_f x : \nu_i' \\ \langle s_i', \mathcal{T} \cup \{\langle s, v \rangle\} \rangle \vdash_f tos : v_i'' \\ \langle s_i', \mathcal{T} \cup \{\langle s, v \rangle\} \rangle \vdash_f x : \nu_i''\end{aligned}$$

and apply the inductive hypothesis twice to obtain $v_i' <: \tau \sqcup v_i''$ and $\nu_i' <: \tau \sqcup \nu_i''$ for each i. We see that

$$\begin{aligned}\tau' = \bigsqcup(v_i' \sqcap \nu_i') <: \bigsqcup(\tau \sqcup v_i'') \sqcap (\tau \sqcup \nu_i'') &= \bigsqcup \tau \sqcup (v_i'' \sqcap \nu_i'') \\ &= \tau \sqcup \bigsqcup(v_i'' \sqcap \nu_i'') \\ &= \tau \sqcup \tau''\end{aligned}$$

as required. □

The next lemma is the future use types equivalent of Lemma 4.4. The next lemma follows the same pattern to Lemma 4.4 and therefore we omit its details.

Lemma 4.9. *For any variable u, execution point s and trails $\mathcal{T}$, $\mathcal{T}'$ such that $\mathcal{T}' \subseteq \mathcal{T}$, then $\tau <: \tau'$ such that*

$$\begin{aligned}\langle s, \mathcal{T} \rangle \vdash_f u : \tau \\ \langle s, \mathcal{T}' \rangle \vdash_f u : \tau'\end{aligned}$$

Proof. We proceed by induction, as in Lemma 4.4. □

```
1 def f():
2     def f():
3         y=2
4     y='a'
5 f()
6 f()
```

Figure 4.7: Functions can redefine themselves

```
1 def f():
2     y='a'
3     if *:
4         f()
5 f()
```

Figure 4.8: Simple recursion

```
1 def f():
2     y='a'
3     if *:
4         g()
5 def g():
6     y=3
7     if *:
8         f()
9 f()
```

Figure 4.9: Mutual recursion

4.8 Type inference examples

In these examples, we demonstrate the capabilities of the type inference to handle different usage scenarios in dynamically typed languages. Figure 4.7 shows a short and strange example where we have a function `f` defined in line 1 that redefines `f` at line 2 when this is called. It also sets `y` with a string value (line 4). The first time `f` is called (line 5), `y` becomes a string value and `f` is redefined. The second time `f` is called (line 6), `y` becomes an integer value. Since execution points indicate the actual functions that have been called running up to the point, i.e., a truncated stack, our type inference can therefore handle this test case with relative ease.

Figure 4.8 is an example that exercises simple recursion. Function `f` can call itself. Our type inference manages to terminate and correctly infer the type of `y` as Str despite the fact that there could possibly be infinite paths through the control flow graph. Although more information can be revealed by going through a loop more than once, there is no more type information that can be gained by going through the same location looking for the same information twice.

Figure 4.9 is another recursion example. In this example, `f` calls `g` while `g` calls `f`. The type of `y` after calling `f` (line 9) is Str ⊔ Int.

In general there is no fixed number of times that a loop must be unrolled in order to compute a fixpoint for a data flow analysis algorithm. Indeed, one can easily construct pathological examples that require substantial unrolling of loops. These would look similar to Figure 4.10

```
1  x2=''
2  x3=''
3  x4=''
4  x5=3
5  while *:
6      x1=x2
7      x2=x3
8      x3=x4
9      x4=x5
10 # type of x1 ?
```

Figure 4.10: Analysing this example requires going around the loop several times.

taken to the extreme (such as repeating the pattern indefinitely). Our trails however implicitly determine a minimal the number of times that a loop needs to be explored for any variable. For example, in Figure 4.10, if the type inference tries to determine the type of `x1` at line 7, it would then try to determine the type of `x2` due to the instructions arising at line 3. Since the type inference has not yet tried to determine the type of `x2`, the trail does not contain `x2` in any entry. The type inference will therefore go round the loop another time and another time for `x3`, `x4` and `x5`. In total, the loop is covered five times. The present type of `x1` at line 7 in this case is $\mathsf{Un} \sqcup \mathsf{Str} \sqcup \mathsf{Int}$.

4.9 Conclusion

In this chapter we have defined the most critical part of the type checking mechanism, the type inference. The novelty of our approach comes from the separation between present and future use types. The type information gathered from this analysis will be crucial to determine the optimal points where to insert the type checking assertions in the original code.

Apart from formalising the type inference, we have also proved its termination and soundness. These properties are extremely important for our type checking mechanism to be useful.

Chapter 5

Type checking and assertion insertion

In this chapter we leverage the type inference mechanism from the previous chapter and explain how the inferred type information is used in preemptive type checking. We start by extending the runtime semantics for the μPython bytecode, defined in Figure 3.4, such that programs are executed using preemptive type checking. We prove correctness and optimality properties for this semantics, which we call the *checked semantics*. We then present an algorithm for inserting explicit type checks into a μPython program such that the program behaves like a program running under a checked semantics when interpreted under an unchecked semantics. We also illustrate the algorithms presented in this chapter on an example μPython program.

5.1 Checked μPython semantics

We now define an alternative semantics for μPython. In this semantics, type errors are raised at the earliest point (given our optimality condition in Definition 5.7) at which it can be determined that the execution will lead to a TypeError.

In principle, an interpreter can raise type errors earlier by using the $StateComp$ predicate (Definition 4.6). This would involve checking at every execution step whether the runtime type of every variable is subsumed by its future use type. Naturally this is very computationally expensive and the resulting interpreter would be too slow. Fortunately, we also have the capability to statically compute present types, which are overapproximations of runtime types. Therefore, we can use this information to reduce the number of runtime type checks that need to be performed. In practice, present and future use types remain fairly consistent between adjacent execution points. Therefore, if a runtime type check is necessary at one point, we want to avoid repeating this at subsequent points if possible. As a first step towards defining a preemptively type checked semantics for μPython, we refine $StateComp$ into $EdgeComp$, a weaker version that makes better use of statically determined type information. $EdgeComp$ can be partially evaluated statically and in many cases a result can be determined without accessing the runtime environment Σ. In these cases, dynamic checks do not need to be performed.

Definition 5.1. A relation $EdgeComp$ on $\langle s, s', \Sigma' \rangle$ holds if for all variables u, we can conclude from

$$\begin{aligned} \langle s, \mathcal{T}_\emptyset \rangle \vdash_f u : \tau_f \\ \langle s', \mathcal{T}_\emptyset \rangle \vdash_f u : \tau'_f \\ \langle s, \mathcal{T}_\emptyset \rangle \vdash_p u : \tau_p \\ \Sigma'(u) : \tau'_r \end{aligned} \tag{5.1}$$

that

$$\tau_f = \tau'_f \quad \text{or} \quad \tau_p <: \tau'_f \quad \text{or} \quad \tau'_r <: \tau_p \sqcap \tau'_f$$

Essentially this says that as the program moves from a state s to a state s', then there is no error to report if either (1) there is no change in the future use types, (2) the statically approximated runtime type is a subtype of future uses, or (3) the actual new runtime type of a variable is within the future use set (modulated by the present type). Clearly only (3) requires the inspection of the runtime types and even then, where the meet $\tau_p \sqcap \tau'_f$ is $\bot$, we know statically that the predicate must fail as there are no constants of type $\bot$. The predicate $EdgeComp$ is used extensively in our checked μPython semantics, as is the following predicate that allows type incompatibilities to be propagated backwards through the CFG.

We have seen that in some cases $EdgeComp$ can be partially evaluated with static information inferred for a particular edge and we can statically conclude that this does not hold. In such cases, we would like to preempt the type error whenever the current execution point goes through such an edge. Furthermore, at points where all paths would lead to a point where a type error can be preempted, we can raise a controlled type exception even earlier. We therefore introduce a new predicate that makes this possible.

Definition 5.2. The fail edge predicate $FailEdge$ is a least predicate that holds at $\langle s, s' \rangle$ if $s \in \text{prev}(s')$ and either $\forall \Sigma' \cdot \langle s, s', \Sigma' \rangle \notin EdgeComp$ or $\{\langle s', s'' \rangle \mid s'' \in \text{next}(s')\} \subseteq FailEdge$.

As an example, suppose that the inferred present type τ_p for a variable u is $\mathsf{Int} \sqcup \mathsf{Str}$ and the inferred future use type τ'_f is Bool and that $\tau_f \neq \tau'_f$. Thus τ_p is not a subtype of τ'_f and $\tau_p \sqcap \tau'_f = \bot$. Since there is no primitive type that is a subtype of $\bot$, then we can conclude that $EdgeComp$ does not hold for the given state. We can arrive to this conclusion without checking the actual type τ'_r of u in the environment Σ'.

We denote the state space of the unchecked semantics by $State^{\rightarrow}$. Our checked semantics for μPython makes use of a different state space, $State^{\dashrightarrow}$ (see Figure 5.1). The main difference to $State^{\rightarrow}$ is that $\mathsf{TypeError}$ is not a valid element of $State^{\dashrightarrow}$. This is because our checked semantics does not raise a type error. Instead it detects these type errors earlier and raises controlled exceptions ($\mathsf{Exception}$). The state $\langle \Sigma, S \rangle$ can denote a state in either $State^{\rightarrow}$ or $State^{\dashrightarrow}$. However, when this state appears in context, it should be clear to which set it belongs.

The checked semantics is defined in terms of the original μPython semantics in Figure 3.4. A crucial difference between the two is that the outcome of a single step can be different. In

$State^{\dashrightarrow}$	::=	Exception	preemptive type error exception
	|	End	end state
	|	$\langle \Sigma, S \rangle$	environment and stack

Figure 5.1: Syntax of checked states

the original semantics, executing $\langle \Sigma, S \rangle$ by a single step can reduce to $\langle \Sigma', S' \rangle$. However, in the checked semantics, executing $\langle \Sigma, S \rangle$ by a single step can reduce to a preemptive type error exception instead, i.e., $\langle \Sigma, S \rangle \dashrightarrow \mathsf{Exception}$.

Definition 5.3. The checked semantics is defined as a binary relation $\dashrightarrow$ on the set of states, $State^{\dashrightarrow}$ comprised of $\langle \Sigma, S \rangle$ states, End, and Exception

$$
\begin{array}{llll}
\langle \Sigma, S \rangle & \dashrightarrow & \mathsf{End} & \text{if } \langle \Sigma, S \rangle \rightarrow \mathsf{End} \\
\langle \Sigma, S \rangle & \dashrightarrow & \mathsf{Exception} & \text{if } \langle \Sigma, S \rangle \rightarrow \mathsf{Exception} \\
\langle \Sigma, S \rangle & \dashrightarrow & \mathsf{Exception} & \text{if } \langle \Sigma, S \rangle \rightarrow \langle \Sigma', S' \rangle \wedge \langle s, s', \Sigma' \rangle \notin \mathit{EdgeComp} \\
\langle \Sigma, S \rangle & \dashrightarrow & \mathsf{Exception} & \text{if } \langle \Sigma, S \rangle \rightarrow \langle \Sigma', S' \rangle \wedge \langle s, s' \rangle \in \mathit{FailEdge} \\
\langle \Sigma, S \rangle & \dashrightarrow & \langle \Sigma', S' \rangle & \text{if } \langle \Sigma, S \rangle \rightarrow \langle \Sigma', S' \rangle \text{ otherwise}
\end{array}
$$

Definition 5.4. A relation $R^{\leq}$ on $State^{\rightarrow} \times State^{\dashrightarrow}$, which relates only identical non-terminating states (i.e., if $\langle \Sigma, S \rangle R^{\leq} \langle \Sigma_1, S_1 \rangle$ then $\Sigma = \Sigma_1$ and $S = S_1$) is called an error-preserving simulation if the following holds:

- $\langle \Sigma, S \rangle \not\rightarrow \mathsf{TypeError}$
- If $\langle \Sigma, S \rangle \rightarrow \mathsf{End}$ then $\langle \Sigma, S \rangle \dashrightarrow \mathsf{End}$.
- If $\langle \Sigma, S \rangle \rightarrow \mathsf{Exception}$ then $\langle \Sigma, S \rangle \dashrightarrow \mathsf{Exception}$.
- If $\langle \Sigma, S \rangle \rightarrow \langle \Sigma', S' \rangle$ then either
  - $\langle \Sigma, S \rangle \dashrightarrow \langle \Sigma', S' \rangle \wedge \langle \Sigma', S' \rangle R^{\leq} \langle \Sigma', S' \rangle$ or
  - $\langle \Sigma, S \rangle \dashrightarrow \mathsf{Exception} \wedge \langle \Sigma', S' \rangle \in \Uparrow$

Let $\lesssim$ be the largest error-preserving simulation.

5.2 Maintaining error preserving simulations

In this section we prove that programs running under preemptive type checking can never raise a TypeError. We also show that under preemptive type checking, if a program raises a controlled exception Exception, then if the same program is run using the original semantics, the program will never reduce to End.

Theorem 5.5. *Let R^{SC} be defined as*

$$\{\langle \Sigma, S\rangle, \langle \Sigma, S\rangle \mid \langle \Sigma_0, \varepsilon\rangle \xrightarrow{*} \langle \Sigma, S\rangle \wedge \langle \Sigma, S\rangle \in \mathit{StateComp}\} \tag{5.2}$$

R^{SC} is an error-preserving simulation.

Proof. Wherever $\langle \Sigma, S\rangle R^{SC} \langle \Sigma, S\rangle$ holds, $\langle \Sigma, S\rangle \in \mathit{StateComp}$, i.e. for all variables u then $\tau_r <: \tau_f$ such that

$$\begin{aligned} \Sigma(u) &: \tau_r \\ \langle s, \mathcal{T}_\emptyset\rangle &\vdash_f u : \tau_f \end{aligned} \tag{5.3}$$

where $s = \lfloor S \rfloor$.

From the definition of error-preserving simulation in Definition 5.4, we need to prove that *all* of the following hold:

$$\langle \Sigma, S\rangle \not\to \mathsf{TypeError} \tag{5.4}$$

$$\text{if } \langle \Sigma, S\rangle \to \mathsf{End} \quad \text{then } \langle \Sigma, S\rangle \dashrightarrow \mathsf{End} \tag{5.5}$$

$$\text{if } \langle \Sigma, S\rangle \to \mathsf{Exception} \quad \text{then } \langle \Sigma, S\rangle \dashrightarrow \mathsf{Exception} \tag{5.6}$$

We also need to prove that the following hold:

$$\text{if } \langle \Sigma, S\rangle \to \langle \Sigma', S'\rangle \text{ then } \langle \Sigma, S\rangle \dashrightarrow \langle \Sigma', S'\rangle \wedge \langle \Sigma', S'\rangle R^{SC} \langle \Sigma', S'\rangle \tag{5.7}$$

$$\text{or} \quad \langle \Sigma, S\rangle \dashrightarrow \mathsf{Exception} \wedge \langle \Sigma, S\rangle \in \Uparrow \tag{5.8}$$

By definition of the checked semantics, if $\langle \Sigma, S\rangle \to \mathsf{End}$ then $\langle \Sigma, S\rangle \dashrightarrow \mathsf{End}$. Therefore we have shown that (5.5) holds as required. The same is true for Exception, i.e., (5.6).

We now proceed to prove that a type error cannot be raised, i.e., (5.4). We prove this, i.e., we assume $\langle \Sigma, S\rangle \to \mathsf{TypeError}$ holds and find a contradiction. We analyse all cases of the μPython semantics where $\langle \Sigma, S\rangle \to \mathsf{TypeError}$.

Case fJIF, i.e., u is tos, s has the form $\langle P, pc\rangle :: \ldots$ and P_{pc} is JIF n. and $\neg(\Sigma(tos) : \mathsf{Bool})$

From (5.3), we infer for this case that τ_f is Bool. Since $\langle \Sigma, S\rangle \in \mathit{StateComp}$, we know that $\tau_r <: \tau_f$. As there is no valid runtime type that is a subtype of Bool other than Bool, this implies that:

$$\tau'_r = \mathsf{Bool}$$

and hence, from (5.3):

$$\Sigma(tos) : \mathsf{Bool}$$

This contradicts the assumption of the current case.

All other cases where $\langle \Sigma, S \rangle \rightarrow \mathsf{TypeError}$, i.e., fJIF, fSTR and fINT, follow this pattern and lead to a contradiction. In fCF1, $\tau_f <: \mathsf{Fn}$ so a contradiction may arise earlier. We therefore conclude that $\langle \Sigma, S \rangle \not\rightarrow \mathsf{TypeError}$ as required.

We now consider cases where $\langle \Sigma, S \rangle \rightarrow \langle \Sigma', S' \rangle$.

Since we know that $\langle \Sigma_0, \varepsilon \rangle \xrightarrow{*} \langle \Sigma, S \rangle$, we can conclude that $\langle \Sigma_0, \varepsilon \rangle \xrightarrow{*} \langle \Sigma', S' \rangle$. We also need to show that either (5.7) *or* (5.8) holds. We proceed by case analysis on $\dashrightarrow$ for the cases where $\langle \Sigma, S \rangle \rightarrow \langle \Sigma', S' \rangle$.

Case $\langle s, s', \Sigma' \rangle \notin \mathit{EdgeComp}$

From the definition of our checked μPython semantics in Definition 5.3 for this case, we can conclude that

$$\langle \Sigma, S \rangle \dashrightarrow \mathsf{Exception} \tag{5.9}$$

By analysing the definition of $\mathit{EdgeComp}$, i.e. Definition 5.1, the current case implies that there is a u such that $\tau'_r \not<: \tau_p \sqcap \tau'_f$, where

$$\begin{aligned} \langle s, \mathcal{T}_\emptyset \rangle \vdash_p u : \tau_p \\ \langle s', \mathcal{T}_\emptyset \rangle \vdash_f u : \tau'_f \\ \Sigma'(u) : \tau'_r \end{aligned}$$

Now since we know from Theorem 4.2 that $\tau'_r <: \tau_p$, we can say that there is a u such that $\tau'_r \not<: \tau'_f$. This means that $\langle \Sigma', S' \rangle \notin \mathit{StateComp}$ (see Definition 4.6).

Hence we know from Theorem 4.7 that $\langle \Sigma', S' \rangle \in \Uparrow$. From the definition of diverge-error relation, this means that $\langle \Sigma, S \rangle \in \Uparrow$ also holds. Therefore combining this result with (5.9), we have shown that (5.8) holds as required.

Case $\langle s, s' \rangle \in \mathit{FailEdge}$

From the checked μPython semantics $\langle \Sigma, S \rangle \dashrightarrow \mathsf{Exception}$. Using coinduction, this means that we need to show that $\langle \Sigma, S \rangle \in \Uparrow$. To do this we must show that $\mathit{FailEdge}$ projects to a diverge-error relation.

That is, let R be $\{\langle \Sigma, S \rangle \mid \langle \Sigma, S \rangle \rightarrow \langle \Sigma', S' \rangle \wedge \langle s, s' \rangle \in \mathit{FailEdge}\}$ and we show that R is a diverge-error relation.

Suppose $\langle \Sigma, S \rangle \in R$, then $\langle s, s' \rangle \in \mathit{FailEdge}$, so either $\langle s, s', \Sigma' \rangle \notin \mathit{EdgeComp}$ and hence $\langle \Sigma, S \rangle \in \Uparrow$, or $\langle s', s'' \rangle \in \mathit{FailEdge}$ for all $s'' \in \mathrm{next}(s')$, as required.

Case $\langle s, s', \Sigma' \rangle \in \mathit{EdgeComp}$

From Definition 5.3 of our checked μPython semantics, we can conclude that

$$\langle \Sigma, S \rangle \dashrightarrow \langle \Sigma', S' \rangle \tag{5.10}$$

In order to prove that (5.7) holds, we need to show that:

$$\langle \Sigma', S' \rangle R^{SC} \langle \Sigma', S' \rangle$$

holds, that is,

$$\langle \Sigma_0, \varepsilon \rangle \xrightarrow{*} \langle \Sigma', S' \rangle \wedge \langle \Sigma', S' \rangle \in StateComp$$

$\langle \Sigma_0, \varepsilon \rangle \xrightarrow{*} \langle \Sigma', S' \rangle$ is clear. We therefore need to show that $\langle \Sigma', S' \rangle \in StateComp$, i.e., that for any u, where

$$\begin{aligned} \Sigma'(u) &: \tau'_r \\ \langle s', \mathcal{T}_\emptyset \rangle &\vdash_f u : \tau'_f \end{aligned}$$

we have

$$\tau'_r <: \tau'_f \tag{5.11}$$

In order to prove (5.11), we start by looking at the definition of $EdgeComp$. This states that

$$\tau_f = \tau'_f \quad \text{or} \quad \tau_p <: \tau'_f \quad \text{or} \quad \tau'_r <: \tau_p \sqcap \tau'_f$$

where

$$\begin{aligned} \langle s, \mathcal{T}_\emptyset \rangle &\vdash_f u : \tau_f \\ \langle s', \mathcal{T}_\emptyset \rangle &\vdash_f u : \tau'_f \\ \langle s, \mathcal{T}_\emptyset \rangle &\vdash_p u : \tau_p \\ \Sigma'(u) &: \tau'_r \end{aligned} \tag{5.12}$$

With Theorem 4.2 guaranteeing $\tau'_r <: \tau_p$, this implies $\tau_f = \tau'_f$ or $\tau'_r <: \tau'_f$.

If $\tau'_r <: \tau'_f$, then we are done. On the other hand, if $\tau_f = \tau'_f$ it is sufficient to show that

$$\tau'_r <: \tau_f \tag{5.13}$$

If $\Sigma(u) = \Sigma'(u)$ then $\tau_r = \tau'_r$ where τ_r is given by $\Sigma(u) : \tau_r$. Because $\langle \Sigma, S \rangle \in StateComp$, we know that $\tau_r <: \tau_f$ and therefore $\tau'_r <: \tau'_f$ as required.

We now assume that $\Sigma(u) \neq \Sigma'(u)$, and keep in mind that $\langle \Sigma, S \rangle \not\to \mathsf{TypeError}$. The instructions for which this is the case are typed using fSET/SG1.

The proof for all these cases follows a similar pattern. We give an example for fSET, where we need to show that

$$\tau'_r <: \tau_f$$

holds. In this case τ_f is such that $\langle s, \mathcal{T}_\emptyset \rangle \vdash_f tos : \tau_f$ and by fSET, $\tau_f = \top$ and therefore

$$\tau'_r <: \top$$

This is trivially true as the top type is at the top of the lattice.

We have therefore demonstrated that (5.11) holds as required. □

Since R^{SC} is an error-preserving simulation and $\lesssim$ is the largest error-preserving simulation, then $R^{SC} \subseteq \lesssim$.

The next corollary is an important result we get from our proofs. This signifies that any program that is run using the checked semantics can never get stuck, but reduces to End or Exception.

Corollary 5.6. *Consider a maximal trace* $\langle \Sigma_0, \varepsilon \rangle \dashrightarrow^* N \not\dashrightarrow$. *Then* N *is either* End *or* Exception.

Proof. We note immediately that $\langle \Sigma_0, \varepsilon \rangle \in \mathit{StateComp}$ holds by virtue of rule fINIT of Figure 4.5. Therefore we have $\langle \Sigma_0, \varepsilon \rangle \, R^{SC} \, \langle \Sigma_0, \varepsilon \rangle$ and hence by the above corollary we have $\langle \Sigma_0, \varepsilon \rangle \lesssim \langle \Sigma_0, \varepsilon \rangle$. Now, suppose for contradiction that N is neither End or Exception. Then we must have N being some $\langle \Sigma, S \rangle$ such that $\langle \Sigma, S \rangle \lesssim \langle \Sigma, S \rangle$. This tells us that $\langle \Sigma, S \rangle \not\to$ TypeError and, by the definition of $\to$ we must have $\langle \Sigma, S \rangle \to \langle \Sigma', S' \rangle$ for some $\langle \Sigma', S' \rangle$. This means that $N \dashrightarrow N'$ for some N' also, contradicting maximality. □

5.3 Optimality

Now that we have shown the correctness of our type inference, we also investigate to what extent our type checking mechanism detects type errors in advance. We define optimality as the ability to detect type errors at the earliest point. Unfortunately our system does not quite enjoy this property. For example consider the following program

```
if *:
  x='a'
  y='a'
else:
  x=5
  y=5
if *:
  intOp(x)
  strOp(y)
else:
  strOp(x)
  intOp(y)
```

In this case we know that this program is destined to yield a type error from the very outset. However, if we consider each variable individually, this is not apparent; for each variable there is a possible error-free control flow through the program, and only by considering the dependencies between these, does the error become apparent. Since our analysis works on a per-variable basis, it only identifies the error after the resolution of the second conditional. Limitations of these kind are not unique to preemptive type checking, and all forms of data flow analysis suffer from this same problem.

We prove that our inference system satisfies a milder form of optimality in general. We have optimality along execution sequences in which there are no branches of control flow. What this means is that if there is a branch that leads to a TypeError in the unchecked semantics, then this branch is never executed because the execution would be preempted by an Exception before traversing into the branch.

Definition 5.7. A reduction step $\langle \Sigma, S \rangle \rightarrow \langle \Sigma', S' \rangle$ is said to happen along a linear path if $\mathrm{next}(s) = \{s'\}$. A reduction sequence $\langle \Sigma, S \rangle \xrightarrow{*} \langle \Sigma', S' \rangle$ is said to happen along a linear path if each step in the sequence happens along a linear path.

Theorem 5.8 (Linear optimality)**.** *Consider a state $\langle \Sigma_0, \varepsilon \rangle$ that is executed a number of times using our checked semantics until it reaches a state $\langle \Sigma, S \rangle$.*

$$\langle \Sigma_0, \varepsilon \rangle \dashrightarrow^{*} \langle \Sigma, S \rangle$$

Suppose that if this state is executed by the unchecked semantics along a linear execution path, this execution path ends in a TypeError*, i.e.*

$$\langle \Sigma, S \rangle \xrightarrow{*} \mathsf{TypeError}$$

Then, $\langle \Sigma, S \rangle$ ⇢ Exception

Proof. We prove this by contradiction, assuming that our checked semantics does not find type errors in a linearly optimal manner. We assume that $\langle \Sigma, S \rangle$ ⇢ $\langle \Sigma', S' \rangle$, but $\langle \Sigma', S' \rangle$ ⇢ Exception.

We first consider two cases: $\langle s', s'' \rangle \in \mathit{FailEdge}$ or $\langle s', s'' \rangle \notin \mathit{FailEdge}$. We consider the first case, i.e., $\langle s', s'' \rangle \in \mathit{FailEdge}$. Since s, s' and s'' form part of a linear trail, from the definition of $\mathit{FailEdge}$, we can conclude that $\langle s, s' \rangle \in \mathit{FailEdge}$. By the checked semantics, in this case $\langle \Sigma, S \rangle$ ⇢ Exception so we have found a contradiction in our hypothesis. From this point onward we therefore assume that $\langle s', s'' \rangle \notin \mathit{FailEdge}$.

Since $\langle \Sigma, S \rangle$ ⇢ $\langle \Sigma', S' \rangle$ and $\langle \Sigma', S' \rangle$ ⇢ Exception, from the definition of the checked μPython semantics in Definition 5.3, together with our previous assumption, we know:

$$\langle s', s'' \rangle \notin \mathit{FailEdge} \tag{5.14}$$

$$\langle s', s'', \Sigma'' \rangle \notin \mathit{EdgeComp} \tag{5.15}$$

where $\langle \Sigma', S' \rangle \rightarrow \langle \Sigma'', S'' \rangle$.

From the definition $\mathit{EdgeComp}$, (5.15) implies that we can pick a u such that:

$$\tau'_f \neq \tau''_f \wedge \tau'_p \not<: \tau''_f \wedge \tau''_r \not<: (\tau'_p \boxdot \tau''_f) \tag{5.16}$$

where τ'_r, τ'_f, τ'_p, τ''_r, and τ''_f are defined such that:

$$\begin{aligned} \Sigma'(u) &: \tau'_r \\ \langle s', \mathcal{T}_\emptyset \rangle \vdash_f u &: \tau'_f \\ \langle s', \mathcal{T}_\emptyset \rangle \vdash_p u &: \tau'_p \\ \Sigma''(u) &: \tau''_r \\ \langle s'', \mathcal{T}_\emptyset \rangle \vdash_f u &: \tau''_f \end{aligned} \tag{5.17}$$

Theorem 4.2 tells us that $\tau''_r <: \tau'_p$, so

$$\tau''_r \not<: \tau''_f \tag{5.18}$$

must in fact hold.

While from the definition of *FailEdge*, (5.14) implies:

$$\exists \Sigma^* \cdot \langle s', s'', \Sigma^* \rangle \in \mathit{EdgeComp}$$

Since τ'_r is the type of an actual value at runtime and there are no values of type $\bot$, implies that

$$\tau'_f = \tau''_f \vee \tau'_p <: \tau''_f \vee (\tau'_p \sqcap \tau''_f) \neq \bot$$

Taken with (5.16), this implies

$$(\tau'_p \sqcap \tau''_f) \neq \bot$$

Collecting the above we have

$$\tau'_p \not<: \tau''_f \wedge (\tau'_p \sqcap \tau''_f) \neq \bot \tag{5.19}$$

We now consider all cases for the last inference rule used in the derivation of $\langle s', \mathcal{T}_\emptyset \rangle \vdash_f u : \tau'_f$ (see Figure 4.5 and Figure 4.6).

We note that fEND and fRAISE are not applicable since τ'_f is the type of u at s', and there is an execution s'' that occurs after s'. Likewise, fINIT is not applicable. fTRAIL is also not applicable since $\mathcal{T}_\emptyset$ is empty.

We now consider rules fSET/JIF/STR/INT, except the special case where $P'_{pc'} = \mathsf{LG}\ x$. Under these cases, we can see that rules pLC/INST/USE also match for τ'_p. In this case, τ'_p is the type of a constant such as Bool, Int, etc. If we analyse the type lattice, we note that there are no types between the level of Bool, Int, etc. and $\bot$. Therefore there is no type τ''_f such that $\tau'_p \not<: \tau''_f \wedge (\tau'_p \sqcap \tau''_f) \neq \bot$, which contradicts (5.19) and so none of these rules could have been used to derive $\langle s', \mathcal{T}_\emptyset \rangle \vdash_f u : \tau'_f$.

All the remaining cases are similar to the special case where $P'_{pc'} = \mathsf{LG}\ x$ for fSET.

Case fSET and $P'_{pc'} = \mathsf{LG}\ x$, i.e., u is tos and s' has the form $\langle P', pc' \rangle :: \ldots$

Before considering this case in detail, let us first consider the last inference rule applied in order to get the type ${\tau'_f}^x$ of x (not tos), derived by the judgement $\langle s', \mathcal{T}_\emptyset \rangle \vdash_f x : {\tau'_f}^x$.

Since $P'_{pc} = \mathsf{LG}\ x$, this rule is fLG1. By this rule and the fact that $\{s''\} = \mathrm{next}(s')$,

$$ {\tau'_f}^x = \upsilon' \sqcap \nu' $$

where υ' is defined such that

$$ \langle s'', \{\langle s', x \rangle\} \rangle \vdash_f tos : \upsilon' $$

Hence ${\tau'_f}^x <: \upsilon'$.

In this case (i.e., fSET), u is tos and therefore τ''_f is defined such that $\langle s'', \mathcal{T}_\emptyset \rangle \vdash_f tos : \tau''_f$. By Lemma 4.9 we can conclude that $\upsilon' <: \tau''_f$. Therefore, by transitivity we conclude that

$$ {\tau'_f}^x <: \tau''_f \tag{5.20} $$

We now consider ${\tau'_r}^x$ (the runtime type of x), which is defined such that $\Sigma'(x) : {\tau'_r}^x$ and we consider the judgement for the runtime type τ''_r of tos where $\Sigma''(tos) : \tau''_r$.

By the μPython semantics, we conclude that ${\tau'_r}^x = \tau''_r$. Therefore, since we know from (5.18) that $\tau''_r \not<: \tau''_f$, we can also conclude that ${\tau'_r}^x \not<: \tau''_f$. Also since we know from (5.20) that ${\tau'_f}^x <: \tau''_f$, we can now conclude that

$$ {\tau'_r}^x \not<: {\tau'_f}^x \tag{5.21} $$

From Theorem 5.5 we know that a state that has been executed several times using the checked semantics maintains an error-preserving simulation. This means that since $\langle \Sigma_0, \varepsilon \rangle \dashrightarrow^* \langle \Sigma', S' \rangle$, then $\langle \Sigma', S' \rangle \in StateComp$, i.e.

$$ {\tau'_r}^x <: {\tau'_f}^x $$

We have therefore found a contradiction with (5.21), as required. □

We have therefore proven that preemptive type checking is at least linearly optimal in terms of type error preemption. Our optimality condition guarantees that type errors are preempted *at worst* at the beginning of the branch where the type error would be raised. In practice, type information and assertions are propagated through control flow splits and joins to earlier points. Hence, linear optimality is not as restrictive as it first appears.

5.4 Type check insertions

In this section we describe an algorithm that transforms bytecode programs by inserting type checks and explicit errors in such a way that the transformed program implements the checked

$P' \longleftarrow \varepsilon$
for $pc \longleftarrow 0..\text{size}(P) - 1$:
  $s \longleftarrow \lfloor \langle P, pc \rangle :: s \rfloor_N$
  for $s' \in \text{next}(s)$:
    if $P_{pc} = \mathsf{JIF}\ \ pc' \wedge s' = \langle P, pc' \rangle :: ... \wedge \langle s, s' \rangle \in \mathit{FailEdge}$:
      $\mathit{extend}(P', \underline{\text{failIfFalse}})$
    if $P_{pc} = \mathsf{JIF}\ \ pc' \wedge s' = \langle P, pc + 1 \rangle :: ... \wedge \langle s, s' \rangle \in \mathit{FailEdge}$:
      $\mathit{extend}(P', \underline{\text{failIfTrue}})$
    if $\langle \varepsilon, s \rangle \in \mathit{FailEdge}$:
      $\mathit{extend}(P', \mathsf{raise})$
    if $P_{pc} \notin \{\mathsf{JIF}\ \ pc', \mathsf{CF}\ \ f, \mathsf{JP}\ \ pc'\}$:
      $\mathit{extend}(P', P_{pc})$
    for $x \in \mathbb{V}$:
      let τ_p be such that $\langle s, \mathcal{T}_\emptyset \rangle \vdash_p x : \tau_p$
      let τ_f be such that $\langle s, \mathcal{T}_\emptyset \rangle \vdash_f x : \tau_f$
      let τ'_f be such that $\langle s', \mathcal{T}_\emptyset \rangle \vdash_f x : \tau'_f$
      if $\neg(\tau_f = \tau'_f \vee \tau_p <: \tau'_f)$:
        if $P_{pc} = \mathsf{JIF}\ \ pc' \wedge s' = \langle P, pc' \rangle :: ...$:
          $\mathit{extend}(P', \underline{\text{checkIfFalse}}(x, \tau_p \sqcap \tau'_f))$
        if $P_{pc} = \mathsf{JIF}\ \ pc' \wedge s' = \langle P, pc + 1 \rangle :: ...$:
          $\mathit{extend}(P', \underline{\text{checkIfTrue}}(x, \tau_p \sqcap \tau'_f))$
        if $P_{pc} \neq \mathsf{JIF}\ \ pc'$:
          $\mathit{extend}(P', \underline{\text{check}}(x, \tau_p \sqcap \tau'_f))$
    if $P_{pc} = \mathsf{CF}\ \ f$:
      $\langle Q, 0 \rangle :: ... \longleftarrow s'$
      $\mathit{extend}(P', \underline{\text{call}}(\mathit{specialise}(Q, s)))$
  if $P_{pc} = \mathsf{JIF}\ \ pc' \vee P_{pc} = \mathsf{JP}\ \ pc'$:
    $\mathit{extend}(P', P_{pc})$

Figure 5.2: Algorithm for inserting type checks in μPython programs, expressed as a function $\mathit{specialise}(P, s)$ that returns an updated program P'.

semantics. An important point to note, however, is that the checked semantics is defined in terms of edges of the truncated CFG, and that nodes in this graph do not correspond uniquely to program locations. That is, each program location may occur many times as the currently executing instruction in different nodes of the graph. For this reason, the bytecode transformation takes as a parameter the particular truncated call stack against which we are inserting checks. If the same program location is reached with a different call stack, then a specialised copy of the program bytecode is created with the relevant assertions for that different call stack inserted. Of course, call sites must be updated to call these specialised programs also.

The algorithm is given in Figure 5.2. It iterates over every instruction of the program, extending the call stack with this instruction as the current one. It then considers edges in the truncated CFG from this point in order to implement the *FailEdge* and *StateComp* predicates. The algorithm makes use of several bytecode macros that are underlined in the algorithm and defined in the Figure 5.3. These are expanded to a list of bytecode instructions. Procedure *extend*,

failIfFalse =

```
    SG tmp
    LG tmp
    JIF l1
    JP l2
l1: raise
l2: LG tmp
```

failIfTrue =

```
    SG tmp
    LG tmp
    JIF l1
    raise
l1: LG tmp
```

call (Q) =

```
    SG tmp
    LC Q
    SG tmpf
    LG tmp
    CF tmpf
```

check (x, τ) =

```
    SG tmp
    LG x
    isInst τ
    JIF l1
    JP l2
l1: raise
l2: LG tmp
```

checkIfFalse (x, τ) =

```
    SG tmp
    LG tmp
    JIF l1
    JP l3
l1: LG x
    isInst τ
    JIF l2
    JP l3
l2: raise
l3: LG tmp
```

checkIfTrue (x, τ) =

```
    SG tmp
    LG tmp
    JIF l2
    LG x
    isInst τ
    JIF l1
    JP l2
l1: raise
l2: LG tmp
```

Figure 5.3: Macros for type checking insertions, where *tmp* and *tmpf* are fresh variables.

which takes a program and a list of instructions, appends the instructions to the end of the given program. When inserting any instructions into a program, the targets of any jump instructions in this program are rearranged to reflect the inserted instructions.

5.5 A worked example

In this section, we go through the μPython example from Chapter 3 (see Figure 3.5), which can raise a TypeError depending on the branch taken at line 4. This compiles to M and P^f, defined as

$$M = [\underset{0}{\mathsf{LC}}\ P^f; \underset{1}{\mathsf{SG}}\ f; \underset{2}{\mathsf{LC}}\ *; \underset{3}{\mathsf{JIF}}\ 7; \underset{4}{\mathsf{LC}}\ \text{'42'}; \underset{5}{\mathsf{SG}}\ x; \underset{6}{\mathsf{JP}}\ 9; \underset{7}{\mathsf{LC}}\ 42; \underset{8}{\mathsf{SG}}\ x; \underset{9}{\mathsf{CF}}\ f; \underset{10}{\mathsf{RET}}]$$

$$P^f = [\underset{0}{\mathsf{LG}}\ x; \underset{1}{\mathsf{intOp}}; \underset{2}{\mathsf{RET}}]$$

We show how preemptive type checking works at each stage and how the type error is preempted at the earliest possible point. The type checking process starts with a control flow analysis; its results are shown in Figure 5.4.

We then show how we conclude that the edge $\langle\langle M, 4\rangle, \langle M, 5\rangle\rangle$ is in $\mathit{FailEdge}$. This means that if the execution moves from $\langle M, 4\rangle$ to $\langle M, 5\rangle$, the program will eventually raise a TypeError or

s	$\langle M,0\rangle$	$\langle M,1\rangle$	$\langle M,2\rangle$	$\langle M,3\rangle$	$\langle M,4\rangle$
line	0	0	2	2	3
instr.	LC P^f	SG f	LC *	JIF 7	LC '42'
prev.	ε	$\langle M,0\rangle$	$\langle M,1\rangle$	$\langle M,2\rangle$	$\langle M,3\rangle$
next.	$\langle M,1\rangle$	$\langle M,2\rangle$	$\langle M,3\rangle$	$\{\langle M,4\rangle,\langle M,7\rangle\}$	$\langle M,5\rangle$
s	$\langle M,5\rangle$	$\langle M,6\rangle$	$\langle M,7\rangle$	$\langle M,8\rangle$	$\langle M,9\rangle$
line	3	3	5	5	6
instr.	SG x	JP 9	LC 42	SG x	CF f
prev.	$\langle M,4\rangle$	$\langle M,5\rangle$	$\langle M,3\rangle$	$\langle M,7\rangle$	$\{\langle M,8\rangle,\langle M,6\rangle\}$
next.	$\langle M,6\rangle$	$\langle M,9\rangle$	$\langle M,8\rangle$	$\langle M,9\rangle$	$\langle P^f,0\rangle::\langle M,9\rangle$
s	$\langle P^f,0\rangle::\langle M,9\rangle$	$\langle P^f,1\rangle::\langle M,9\rangle$	$\langle P^f,2\rangle::\langle M,9\rangle$	$\langle M,10\rangle$	
line	1	1	1	6	
instr.	LG x	intOp	RET	RET	
prev.	$\langle M,9\rangle$	$\langle P^f,0\rangle::\langle M,9\rangle$	$\langle P^f,1\rangle::\langle M,9\rangle$	$\langle P^f,2\rangle::\langle M,9\rangle$	
next.	$\langle P^f,1\rangle::\langle M,9\rangle$	$\langle P^f,2\rangle::\langle M,9\rangle$	$\langle M,10\rangle$		

Figure 5.4: Control Flow for the μPython example

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle\langle M,10\rangle,\{\langle\langle P^f,2\rangle::\langle M,9\rangle,x\rangle,\langle\langle P^f,1\rangle::\langle M,9\rangle,x\rangle,...\}\rangle\vdash_f x:\top}\text{fEND}}{\langle\langle P^f,2\rangle::\langle M,9\rangle,\{\langle\langle P^f,1\rangle::\langle M,9\rangle,x\rangle,\langle\langle P^f,0\rangle::\langle M,9\rangle,x\rangle,...\}\rangle\vdash_f x:\top}\text{fRET}}{\langle\langle P^f,1\rangle::\langle M,9\rangle,\{\langle\langle P^f,0\rangle::\langle M,9\rangle,x\rangle,\langle\langle M,9\rangle,x\rangle,...\}\rangle\vdash_f tos:\mathsf{Int}/x:\top}\text{fintOp1/2}}{\langle\langle P^f,0\rangle::\langle M,9\rangle,\{\langle\langle M,9\rangle,x\rangle,\langle\langle M,6\rangle,x\rangle,...\}\rangle\vdash_f x:\mathsf{Int}}\text{fLG1}}{\langle\langle M,9\rangle,\{\langle\langle M,6\rangle,x\rangle,\langle\langle M,5\rangle,tos\rangle,...\}\rangle\vdash_f x:\mathsf{Int}}\text{fCF2}}{\langle\langle M,6\rangle,\{\langle\langle M,5\rangle,tos\rangle\}\rangle\vdash_f x:\mathsf{Int}}\text{fRET/JP}}{\langle\langle M,5\rangle,\mathcal{T}_\emptyset\rangle\vdash_f tos:\mathsf{Int}}\text{fSG2}$$

$$\dfrac{'42':\mathsf{Str}}{\langle\langle M,4\rangle,\mathcal{T}_\emptyset\rangle\vdash_p tos:\mathsf{Str}}\text{pLC1}\qquad\dfrac{}{\langle\langle M,4\rangle,\mathcal{T}_\emptyset\rangle\vdash_f tos:\top}\text{fSET}$$

Figure 5.5: Derivations of present and future use types at $\langle M,4\rangle$ and $\langle M,5\rangle$. In each rule the side-conditions are not shown. The rules are applied to the location at the top of the call stack.

diverge. From the definition of *FailEdge*, we need to show that

$$\forall\Sigma'\cdot\langle\langle M,4\rangle,\langle M,5\rangle,\Sigma'\rangle\notin EdgeComp \tag{5.22}$$

We have derivations of the following in Figure 5.5,

$$\langle\langle M,4\rangle,\mathcal{T}_\emptyset\rangle\vdash_f tos:\top\quad\langle\langle M,4\rangle,\mathcal{T}_\emptyset\rangle\vdash_p tos:\mathsf{Str}\quad\langle\langle M,5\rangle,\mathcal{T}_\emptyset\rangle\vdash_f tos:\mathsf{Int}$$

Since $\mathsf{Int}\neq\top$, $\mathsf{Int}\not<:\mathsf{Str}$, and the fact that there can be no τ_r such that $\tau_r<:\bot$, we know that (5.22) holds. Similarly, we also conclude that $\langle\langle M,3\rangle,\langle M,4\rangle\rangle\in FailEdge$.

The edge in (5.22) represents the transition from line 4 to line 5 in the source code. The checked semantics would therefore raise an Exception at that point. Now we insert type checks in

```
def f():
  return intOp(x)
if *:
  raise
  x ='42'
else :
  x = 42
f()
```

Figure 5.6: The transformed μPython example with preemptive type checking.

M. Since this is the program at the outermost scope, the specialisation argument is ε and $specialise(M, \varepsilon)$ is called. According to the definition of $FailEdge$, $specialise$ should insert a failure assertion at each edge $\langle\langle M, 3\rangle, \langle M, 4\rangle\rangle$ and $\langle\langle M, 4\rangle, \langle M, 5\rangle\rangle$. However, in our implementation we optimise by only inserting raise at the first point in the sequence of failing edges. Therefore the transformed bytecode for M is:

$$M' = [\mathsf{LC}\ P^f; \mathsf{SG}\ f; \mathsf{LC}\ *; \underline{\mathrm{failIfTrue}}; \mathsf{JIF}\ 7 + n; \mathsf{LC}\ \text{'42'}; \mathsf{SG}\ x; \mathsf{JP}\ 9 + n; \mathsf{LC}\ 42; \mathsf{SG}\ x; \mathsf{CF}\ f]$$

where the inserted code is underlined and n is the length of the instructions in $\underline{\mathrm{failIfTrue}}$. This is equivalent to the high-level program shown in Figure 5.6. The check is therefore inserted at the *earliest point* at which we can guarantee that the execution *will* end in a TypeError.

It is interesting to compare this example, say, with the approach used in gradual typing with unification based inference [98]. Since variable x is assigned both a Str and an Int in different locations, and is used as an Int, x would be inferred to have type Dyn and a type error could only be raised at the application of intOp. This is typical for other type systems which allow this program to be statically type checked [22, 5, 117]. Other static analysis approaches for dynamic languages would reject this program outright [10, 18, 6].

5.6 Conclusion

In this chapter we have formalised the type checking mechanism behind preemptive type checking. We have defined this in terms of a checked semantics for μPython that implements the type checking mechanism using inferred type information from the type inference defined in Chapter 4.

We have proven correctness and optimality properties for the checked semantics and presented an algorithm for inserting explicit type checks into a μPython program. Such a program behaves like a program running under a checked semantics when interpreted under an unchecked semantics. We have also illustrated the algorithms presented in this chapter on an example μPython program.

Chapter 6

From μPython to full Python

In this chapter we describe how we make use of the algorithms developed in Chapters 4 and 5 to build a tool that implements preemptive type checking. Our tool supports a larger subset of the Python language than μPython. We then evaluate this tool with some synthetic examples and some benchmarks from the computer languages benchmarks game [4].

6.1 Introduction

We implemented the preemptive type checking tool as a Python 3.3 library that can be loaded with the target program. It can be invoked at runtime, typically during the initialisation of a program, to transform an existing function in such a way as to implement the semantics of preemptive type checking. This design decision makes our library easy to use, as we will see in Section 6.2. Despite the fact that the analysis is actually performed at runtime, the techniques used are static analysis techniques and the analysis is meant to be invoked once.

We have based our implementation on Python 3.3 and we support a number of features, including:

- local and global variables (Section 6.5),
- the evaluation stack (Section 6.6),
- control structures such as while-loops,
- polyadic functions, anonymous functions,
- tuples, and operators without overloading, and
- some standard library functions, which we annotate with type information.

We also allow a user to explicitly add type annotations to any functions using function annotations [115].

We use the implementation described in this chapter to demonstrate the usefulness of preemptive type checking. We show this on both synthetic examples and selected benchmarks from the computer language benchmarks game [4]. This benchmark suite compares measurements of programs written in different programming languages.

6.2 Architecture of the tool

We choose to implement our tool for Python in Python mainly to improve usability. In order to use our tool we simply load the library in the source file under analysis and invoke the transformer at runtime on the entry point function, such as a user defined function `main` or any other function. At this point, the analysis performed will be a static analysis. The tool takes full advantage of Python's reflection and limited metaprogramming capabilities. This design decision also makes it possible to allow features not supported by our tool to be used during the initialisation of a program. These include code loading and creation of functions.

High level overview

Our type checking process is integrated with the runtime environment. Unlike most analysers, our type checker does not require the program's source code. Instead, our type analysis works directly on a live program and environment, introspecting and analysing the environment for the currently executing program. Our type checking mechanism is called on a particular function, for example `main`. This function is created and initialised by the standard interpreter. We refer to this phase as the initialisation phase (see Figure 6.2), and in this phase the full power of the Python language can be used. The semantics of the language during this phase are not affected and therefore the program might raise a type error. We show this in an example in Figure 6.1, where the program is executed using the standard semantics up to line 10. Then, the type checking library is invoked on a particular function, for example `main`, as in line 10. Then, a version of `main` with inserted type checks is introduced in the environment at line 13. This is subsequently called at line 15.

The analysis phase, which partly implements preemptive type checking, splits the problem into different stages. As outlined in Figure 6.2 and Figure 6.3, the analysis starts with a control flow analysis. This is followed by a type analysis, where the present and future use types of any variable at any point are calculated. Given this information, the position and kind of type checks that need to be inserted can be established.

It is then possible to emit the bytecode with the type checks inserted. Using the information gained by the analysis, we simply copy the bytecode in the original function that is being type

```
1  from typer import Analyser
2
3  def main():
4      # Python, with some restricted features
5      .
6
7  if __name__=='__main__':
8      # Full Python language up to here.
9      # We first analyse the function initialised above.
10     a=Analyser(main)
11     # We transform the function such that it
12     # implements preemptive type checking semantics.
13     a.emit()
14     # We call the transformed function.
15     _main()
```

Figure 6.1: Phases of the type checking process, outlined in the user code.

checked or any function called from within and interleave the type checks. Emitting bytecode with these type checks inserted is an optional step; the user can simply get a printout of the warnings that pinpoint potential type errors in the original code without actually running the program.

When the function that goes through this process is executed, the simple type checks inserted in the function make sure that any type error is preempted as early as possible with an informative error message.

6.3 Using the type checker on existing programs

In this section we show how to make use of our type checker. The entry point to the analysis is the class `Analyser` which takes a `callable` object such as the main function, and an integer truncation level. The `Analyser` first constructs the truncated CFG and then iterates over all nodes in order to calculate the present and future use types for the accumulator. All type calculations are thus cached during the iteration across the CFG so that present and future use types for all necessary variables in all states are established. The type checks that need to be inserted in the code are also calculated. Warnings can be printed to the screen via the `printWarnings` method available in the `Analyser` class. This prints out a description of the type checks that need to be inserted in the bytecode.

After this, an implementation of *specialise* as in Figure 5.2 is used to transform the program into a type preempted version of the bytecode. This resides in the method `emit` in the `Analyser` class. This method also takes the current `globals()` dictionary, and updates the necessary functions in this dictionary to work with preemptive type checking. The `Analyser` class can also issue messages explaining the potential type failures in the given function. This includes the line numbers of the fault locations. We do this by passing error messages with exceptions that record the file names and line numbers of program locations that have caused a preemptive type error, along with the expected and actual types.

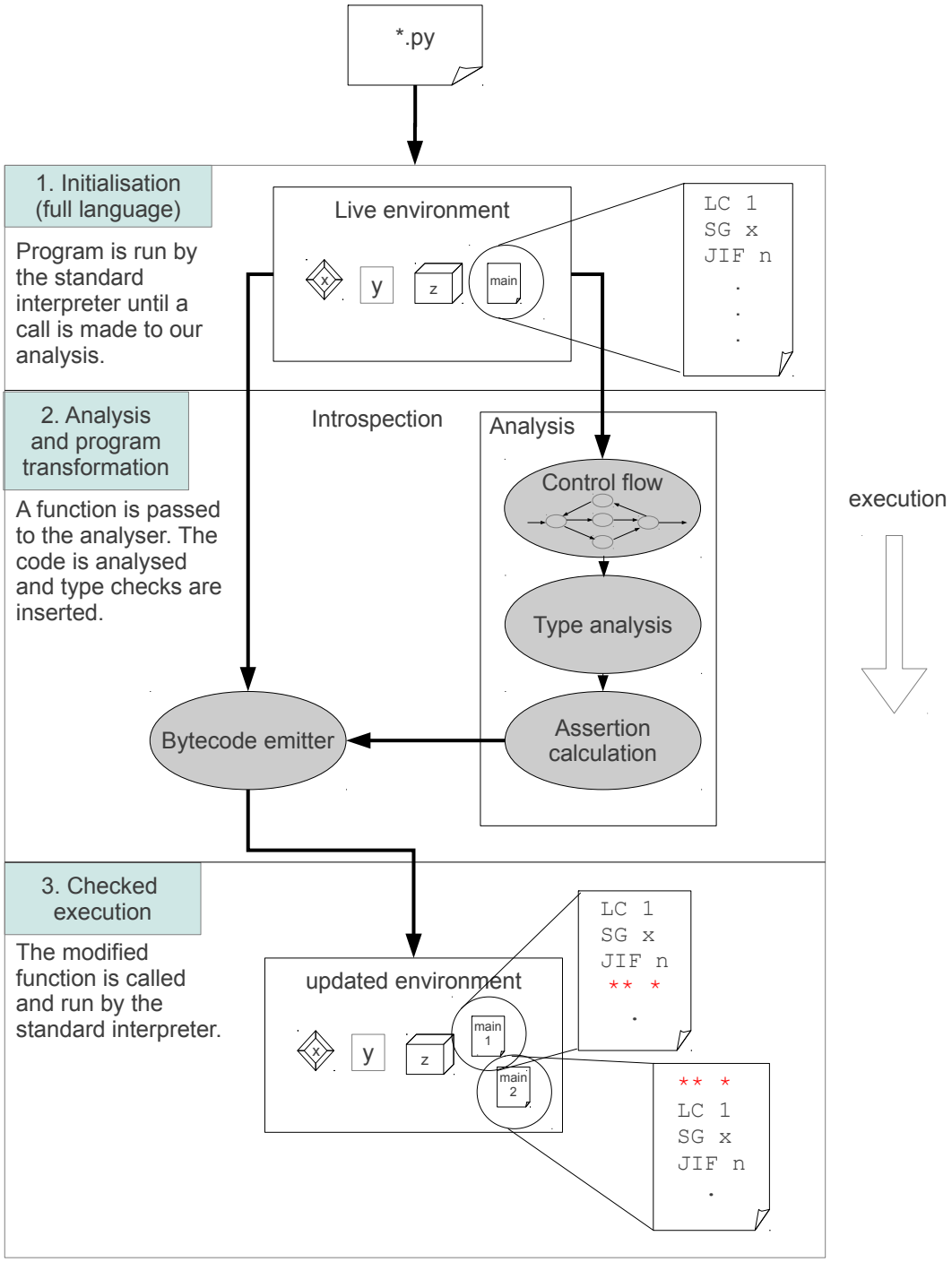

Figure 6.2: Outline of type checking process.

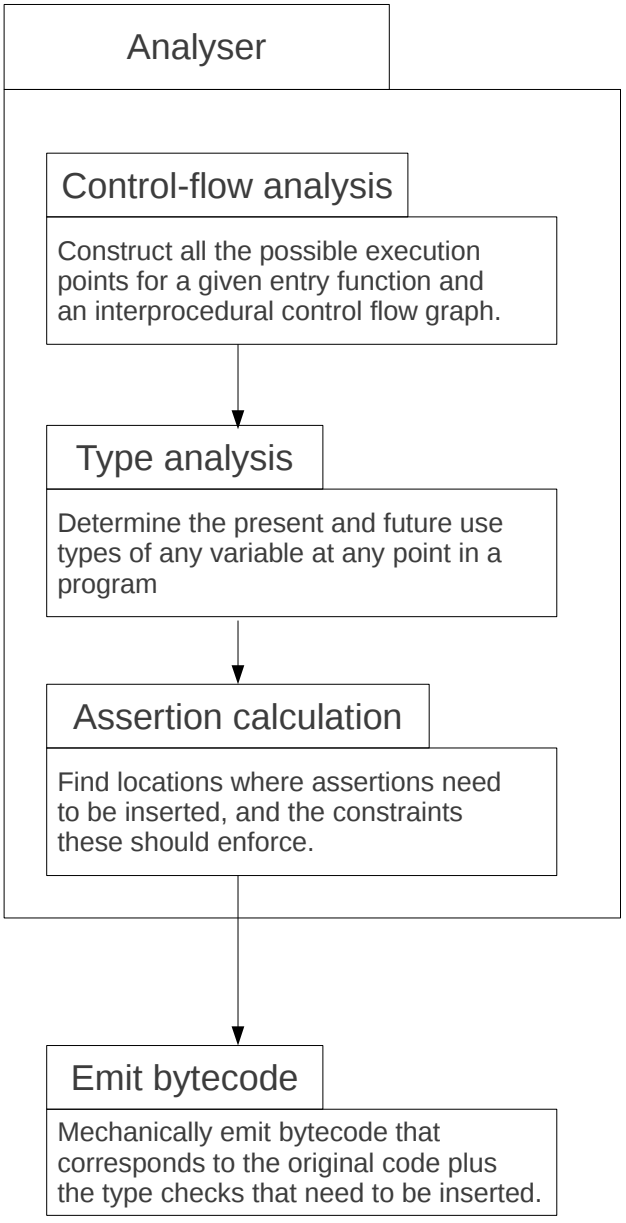

Figure 6.3: Conceptual structure.

To use the implementation, the static analysis of a function (for example called `main`) is invoked by:

```
a=Analyser(main,2)
a.emit(globals())
```

This transforms the function `main`, and all other functions it calls, into specialised versions that implement preemptive type checking. Then, calling the specialised version of `main` activates the preemption to catch any runtime type errors. If it can be statically determined that a type error is unavoidable, `main` will raise an immediate exception on calling.

We only support a limited subset of Python, which makes it difficult to use our library with real world code. We mitigate this problem by supporting optional type annotations that are used to indicate the type of a function to our type inference. This feature makes use of function annotations [115], a recent feature of the Python language. These can be easily applied to existing code:

```
def foo(x:Number,l:MutableSequence) -> NoneType:
    ...
    # unsupported feature used here
    ...
    ...
```

Whenever our analyser encounters a function that is type annotated, it does not analyse its body and therefore we can use any feature supported by Python in the body of such a function.

Instruction objects

Our implementation builds an object structure that follows the structure of the program. We have therefore designed a class structure to represent the various Python instructions. Each individual instruction class maps to a Python bytecode instruction. The responsibilities of each instruction object include:

- Calculating the next execution point given the current execution point.
- Implementing the typing rules associated with an instruction.
- Calculating the stack displacement associated with an instruction.

In terms of storage requirements, instruction objects are quite small. These mainly store the operand of the instruction and a reference to the current execution point. They are however responsible for approximately half of the implementation of the whole tool. The rest of the logic is encapsulated in the `Analyser` class, which links together the instruction objects. This has a special factory method that, given a program execution point, constructs a corresponding instruction object and keeps these in a pool. By extension, since execution points are linked in

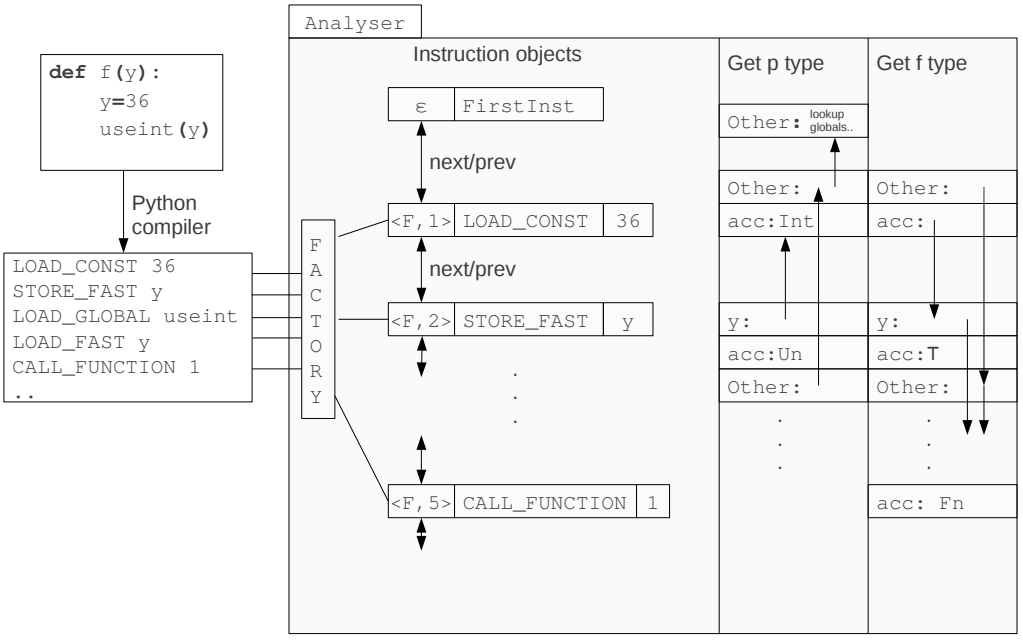

Figure 6.4: Instantiation and interaction of instruction objects.

a graph, instruction objects are also. We can see the interaction between instruction objects in Figure 6.4. In this diagram we can see how the individual instruction objects correspond to the actual instructions in the program. We can also see the interaction between the different instruction objects. For example, each execution point that an instruction object holds is associated to other execution points through the prev and next operations. In the example in Figure 6.4, we see that to get the present type of y at execution point $\langle F, 2\rangle$, one needs to get the present type of tos at $\langle F, 1\rangle$.

Instruction objects follow a hierarchical subclass structure and therefore all instruction objects have a common interface. Each Python instruction that we support has its corresponding Python class. There are also two special kinds of instruction classes `FirstInst` and `LastInst`, which follow a variant of the Null Object [61] design pattern.

Analyser class

The analyser class is mainly responsible for gluing together the logic provided by the instruction objects, calculating type checks to insert and emitting specialised bytecode. As can be seen in Figure 6.4, instruction objects are instantiated and contained in instances of the `Analyser` class. As opposed to any instruction class, the `Analyser` class is a heavyweight class. It contains all internal caches and the calculated type checks.

6.4 Control flow analysis

As we have seen in Chapter 4, the type inference mechanism and its correctness are independent of the choice of the control flow analysis algorithm. Our type inference mechanism requires an implementation for next and prev over execution points, and hence a control flow graph, where nodes in these graph are actually execution points. As long as the control flow analysis returns an overapproximation of the actual control flow that happens at runtime, the soundness of the inference algorithm would hold. The more precise the control flow analysis algorithm is, the more precise the inferred present and future use types can be. Moreover, the more precise the inferred types are, the less type checks need to be inserted into the resulting code.

Clearly, as our system is built upon a static control flow analysis, we need an implementation for this. As in the theory, our implementation is also parametric to the implementation of the control flow analysis. There are several algorithms that could be used, including the well known k-CFA [93, 94]. Unfortunately, for languages such as Python there are no off-the-shelf implementations available that can perform control flow analysis. There is, however, a toolkit [14] that performs control flow analysis on languages with less dynamic features than Python, such as C. This toolkit took 50 person-years of effort over eight elapsed years to build [14]. The engineering effort required to implement a full-fledged control flow analyser for the full Python language is therefore beyond our capabilities. We have instead implemented a simple version.

For the control flow analysis to take place, we have to first extract and parse the bytecode from the function that we are analysing. For this purpose, we use BytePlay, a Python bytecode parser, which we ported to support Python 3.3. This involved converting the functionality from Python 2 to Python 3, but it also involved supporting the new bytecodes and the new bytecode structure. We use this library to parse, analyse, and repackage the bytecode. As a proof of concept, we use a simplified version of next and prev in which we assume that all function definitions are either made in the preamble to the main program or are single use anonymous functions.

There are various frameworks that make use of intermediate program representations in *static single assignment* form [63, 87]. In order to support such a representation we would need to introduce appropriate rules in Figures 4.3–4.6 for ϕ instructions. We believe that analysing programs in SSA form would not facilitate the implementation of our algorithms. As we shall see in the next section, Python makes a distinction between local and global variables, and we handle both in our implementation. The control flow graphs that are followed when determining the types of these variables are different. For example, in order to determine the type of a local variable, one does not need to follow an interprocedural control flow graph. To keep the design simple and the implementation fast, we construct two different control flow graphs: a global inter-procedural control flow graph and disjointed intra-procedural control flow graphs for the main function and all other functions called within. We shall explain how the global control flow graph is constructed, as this involves more work.

```
oldpoints⟵ ∅
oldedges=⟵ ∅
points⟵ {ε, ⟨M, 0⟩ :: ε}
edges⟵ {⟨ε, ⟨M, 0⟩ :: ε⟩}
def addedge(s, s′):
    points⟵ points ∪ {s′}
    edges⟵ {⟨s, s′⟩}
def inc(s):
    ⟨P, pc⟩ :: s* ⟵ s
    return ⟨P, pc + 1⟩ :: s*
while oldedges≠edges:
    oldpoints⟵points
    oldedges⟵edges
    for s ⟵ points:
        ⟨P, pc⟩ :: s* ⟵ s
        if P_pc is a call function:
            P′ ⟵ code of target function
            addedge(s, ⌊⟨P′, 0⟩ :: s⌋_N)
        elif P_pc is a return from function:
            for ⟨s′, s″⟩ ⟵edges:
                if s″ = ⟨P, 0⟩ :: s*:
                    addedge(s,inc(s′))
        elif P_pc is a jump:
            pc′ ⟵ target of instruction
            addedge(s, ⟨P, pc′⟩ :: s*)
            if P_pc is a conditional jump:
                addedge(s,inc(s))
        else:
            addedge(s,inc(s))
```

Figure 6.5: Algorithm for constructing the intra-procedural control flow graph

Our algorithm for constructing the control flow graph is defined in Figure 6.5. We represent a control flow graph as a set of edges, where every edge is a pair of execution points. We also keep a set of all execution points, and maintain indices for fast lookup of next and previous points given a particular point. Our algorithm for constructing the control flow graph initially starts with two execution points: these correspond to ε and $\langle M, 0\rangle :: \varepsilon$ and a single edge between them. This incrementally constructs a control flow graph until a fixpoint is reached on the set of edges. Hence, for every point s we get the next point s' and add s' to our global set of points. We also add $\langle s, s'\rangle$ to the global set of edges. Here is how we determine the next point given a point s. The following steps are repeated for all execution points:

1. If the instruction at s is not a jump, a call function or a return instruction, the next point is the same as s, except that the program counter is incremented by one.

2. If the instruction at s is an unconditional jump, the next point is constructed by looking at the target of the jump. In the case of a conditional jump, we also follow step 1.

3. If the instruction at s is a call function instruction, we statically determine the code of the function that is being called. In our simple implementation, we assume that functions are only defined once and always before being used. We append this function together with a program counter value of one to s. We truncate this new point to the execution point depth N.

4. If the instruction at s is a return instruction, we look at all the edges and find an edge which leads to the entry point s'' of the current function. s'' matches s in everything except the program counter. In this case s' is the next point in program order of s''.

Throughout this stage we also maintain the lookup indices.

6.5 Type analysis

In this section, we describe the process of determining present and future use types of any variable at any point in a program. A main aspect of this process is the interaction between instruction objects when determining the present and future use type at the point that these are associated with. In Figure 6.4 for example, we show as arrows the interaction between instruction objects when getting a particular type.

We represent instructions as classes, and instances of instructions in the program as objects. Each instruction object contains methods (`gtp` or `gtf`) to get the present or future use type of any variable at the location associated with the instruction. In turn, these methods call other methods connected with these instructions via prev and next.

We tried two different approaches for the type analysis. In the first approach, we implemented the type rules in a functional programming style and we kept a trail inside the `Analyser` class. The result was a program that recursively called the respective functions that implemented the type rules. We found that this approach is the fastest approach for finding the type of a single variable at a single point, as the type information is gathered lazily and incrementally. This approach would be suitable for an analyser in an IDE, where the programmer would want to know the type of a single variable at a single point.

The second approach we tried, which is the one we base the results on, finds the types of all variables at all points by starting with the assumption that all types are of type $\perp$. Then, we iterate through every point and every variable mentioned in the program that is being analysed and calculate its type. This process is repeated until we reach a fixpoint on the types for all variables for all points.

An important reason why a trail based approach should be considered in practice is the fact that this kind of type inference is guaranteed to terminate (see Theorem 4.1). We have not proved this for an implementation that tries to find a global fixpoint on the inferred types. Nonetheless, since this approach scales better in cases where we want to find the types of all variables at all execution points, the results and timings that we present are based on this implementation.

Local variables, global variables and evaluation stacks

In Python, local variables can only be redefined within the scope in which they are defined, while global variables can be redefined globally. In the case of mutable objects, however, one can redefine a field of an object defined in another scope or perhaps an entry in a list. Since we do not support container types at this stage, function calls do not have any effect on locals.

We support both local and global variables with a class structure that represents names. These include variable names and also positions in the evaluation stack. Therefore, there are three main concrete classes. `Local` represents local variable names, `Global` represents global variable names and `StackOffset` represents positions within the evaluation stack. In the case of a stack we store the position of the stack as an integer. In the case of a variable name, we represent the visibility of the variable (local or global) and also the actual variable name (such as `x` and `y`). Since locals and globals are stored in different dictionaries, there are no name collision issues between local and global variables.

In order to support local variables, we extend the type rules such that these also consider local variables. In order to determine the type of a local variable, we do not need to traverse the global control flow graph. Therefore we traverse an intra-procedural control flow graph when looking up the type of a local variable. In the case of present types, if we reach the beginning of the program contained in the current function, we then look at the arguments of the function. If the local variable is passed as an argument, its type can be determined by looking at the type of the value passed at the corresponding stack position when calling the function. Local variables that are not defined inside a function or not passed as arguments have type Un.

6.6 Modelling a stack

The Python virtual machine is a stack based machine. The evaluation stack serves as working memory and is read and manipulated by a large portion of the bytecode instructions. For example, load operations push a single element on to the stack while store operations pop a single element from it.

When an element is pushed or popped on to or from the stack, it displaces all other elements by one position. For example, when executing a `CALL_FUNCTION n` instruction, `n` elements are popped from the stack and the return value is pushed. The operand `n` corresponds to the number of arguments applied to the function.

For the general case, we model the stack by calculating how much the stack has shifted for every execution point. For example, consider the following table:

	instruction	stack shift	prev	next
s	`JUMP_ABSOLUTE`	no change	...	s'
s'	`CALL_FUNCTION 2`	pops 2	s	s''
s''	`LOAD_CONST`	pushes 1	s'	...

In the first column, we have the execution points s, s' and s''. The corresponding instructions at these points are in the second column. The third column describes the current instruction's effect to the stack. If we try to infer the present type of the element stored at stack position 4 after executing the instruction at s'', we need to take into consideration these effects. In this case we need to infer the present type of stack position 3 at s' and hence stack position 5 at s. Conversely, the future use type of stack position 3 before executing the instruction at s is the same as stack position 3 at s' and stack position 1 at s''. We observe that in the 100+ bytecode instructions in Python 3.3, this shift can be calculated statically in bytecode generated by the Python compiler.

The Python interpreter performs some preliminary checks on the bytecode before this bytecode is run. One of these is to make sure that if the control flow is split, then the stack depth when the control flow is rejoined is even. In case of a control flow split or join, the existing machinery for joining types therefore automatically handles the different types that may be present at different stack positions under the different branches.

The arguments to a function and the return value are also passed over the stack. For example, the statement `z=f(x,2)` (where `f` is a global variable) is translated into the following bytecode instructions. In this example, we also show what the contents of the stack are at each location.

```
0 LOAD_GLOBAL f          Stack: f]
1 LOAD_FAST x            Stack: x,f]
2 LOAD_CONST 2           Stack: 2,x,f]
3 CALL_FUNCTION          Stack: result]
4 STORE_FAST z           Stack: ]
```

6.7 Type check insertion

Our implementation of the type checking mechanism is similar to the mechanism described in Section 5.4. Since our implementation works on Python rather than μPython, we can use more advanced Python features together with reflection, to implement the type check insertion mechanism. As in Section 5.4, we may need to map multiple execution points to the same code locations and we do so by specialising functions according to the execution point of their call site.

The type check insertion mechanism does not physically insert checks in the bytecode but simply keeps track of the insertions that need to happen at particular execution points. The actual insertion is performed by the bytecode specialisation algorithm, which is described later on.

```
for ⟨s, s′⟩ ⟵ all edges:
  if ⟨s, s′⟩ ∈ FailEdge:
    Insert code at s′ to fail if previous point is s
    continue
  for s ⟵ prev(s′):
    Insert code after s to store s as the previous point
    for u ⟵ 𝕍⁺:
      τ_p ⟵ the p type of u at s
      τ_f ⟵ the f type of u at s
      τ′_f ⟵ the f type of u at s′
      if τ_f ≠ τ′_f and τ_p ≮: τ′_f:
        Insert code at s′ to fail if previous point is s and
            the runtime type τ_r of u is not a subtype of τ_p⊓τ′_f
        for s ⟵ prev(s′):
          Insert code after s to store s as the previous point
```

Figure 6.6: Type checking insertion mechanism, this takes in a set of failing edges *FailEdge* and returns all the insertions to be made

The type check insertion algorithm, expressed in Figure 6.6, goes through all control flow edges $\langle s, s' \rangle$. If $\langle s, s' \rangle \in \mathit{FailEdge}$ then we insert code to raise a preemptive type error at execution point s'. This checks that the previous execution point was s. In order to do this, we insert instructions at all code locations corresponding to $\mathrm{prev}(s')$ to store a representation of the execution point in a reserved global variable. This is a different approach to the algorithm described in Section 5.4. Similarly, code is also inserted to perform the necessary type checks on any variable, as required. This is inserted only in cases where there was a change in the future use type between s and s' and the present type is not a subtype of the future use type.

From our implementation, we found that the simplest and most efficient way to implement the assertion insertion mechanism is to produce a specialised version of the same function according to the different truncated runtime call stacks. In order to do this, we introduce the concept of *specialisation points*. A specialisation point s^* is a further truncation of an execution point and therefore contains all the elements of an execution point except for the topmost element. Specialisation points represent call sites, and when $\langle P, pc \rangle$ is appended to a specialisation point s^*, an execution point is constructed that represents a point inside P, when called from s^*. Therefore, in order to get a set of all the different specialisations for a particular program P, we have the following set comprehension:

$$\text{specialisation points for } P = \{s^* \mid \forall s \cdot \langle P, pc \rangle :: s^* = s\}$$

The algorithm for specialising a particular program P for a specialisation point s^* (i.e. variable containing P called at execution point that ends with s^*) is shown in Figure 6.7. In order to make use of the specialised versions in each function, we need to modify the call site of any function, so that instead of calling the original function, we call the specialised function instead. A call site in Python contains a `CALL_FUNCTION` instruction. There could potentially be a number of different functions that are called from a single call site. In order to maintain the same semantics

```
P' ⟵ [ ]
pc ⟵ 0
for pc ⟵ 0..size(P) − 1:
  s ⟵ ⟨P, pc⟩ :: s*
  if there is code to insert at s:
    P' ⟵ P' + code to insert
  if P_pc is a function call:
     P' ⟵ P' + compiled version of{
       Load the target function code into Q
       Call the specialised version of Q for ⌊s⌋_{N−1}
     }
  P' ⟵ P' + P_pc
  if there is code to insert after s:
    P' ⟵ P' + code to insert
```

Figure 6.7: Algorithm for emitting the bytecode. This takes the specialisation point s^* for P, and all insertions to be made and returns the specialised program P'

in the modified function, we need to replace all `CALL_FUNCTION` instructions with code that dispatches over the actual function being called. This is contained at a stack position that is statically calculated. A pre-calculated specialised version of this function is called instead. This is currently not fully implemented and tested, but since we assume that all functions are declared in a preamble to the function under analysis, no dispatching has to take place. Therefore we can statically determine which function is going to be called at every call site.

In our implementation, optimised versions of these algorithms are used. One of the optimisations is that of determining which variables are relevant at every execution point, which is described in the next section. We now demonstrate the type checking insertion on the example presented in the introduction in Figure 1.2. In this case, the execution point depth is set to 2. Two different specialised versions of `compute` are generated. This is because there are *two possible distinct execution points at the entry* to the function `compute` for an execution point depth of 2. In this case, these correspond to lines 10 and 18 in Figure 1.2.

A disassembly of the bytecode of the transformed version of `main` is shown in Figure 6.8. The first column contains the line number in the original source code. The second column is the offset in the bytecode string, in bytes. The third column is the instruction opcode and the forth column is the operand. The fifth column is a comment describing the operand. In these bytecode listings, we display the inserted or modified instructions in red and mark the corresponding line with a #. From this we can notice that the original line numbers are preserved, even though new bytecode instructions are inserted. Therefore this code still works in the debugger. In Figure 6.8 we can see that two instructions are inserted at the start. These store the first execution point, represented as an empty tuple, in a global variable `previous_stack`. Indeed, all execution points are represented as tuples rather than as stacks, as there is built-in support for this data structure. In this disassembly we can also see that at bytecode offset 79, `main35_compute` is loaded and subsequently called. This is the specialised version of `compute` for the call site

```
#14           0 LOAD_CONST               1 (())
#             3 STORE_GLOBAL             0 (previous_stack)
              6 LOAD_GLOBAL              1 (len)
              9 LOAD_GLOBAL              2 (argv)
             12 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             15 LOAD_CONST               2 (2)
             18 COMPARE_OP               0 (<)
             21 POP_JUMP_IF_FALSE       51

 15          24 LOAD_GLOBAL              3 (abs)
             27 LOAD_GLOBAL              4 (int)
             30 LOAD_GLOBAL              5 (input)
             33 LOAD_CONST               3 ('enter initial value: ')
             36 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             39 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             42 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             45 STORE_GLOBAL             6 (initial)
             48 JUMP_FORWARD            22 (to 73)

 17      >>  51 LOAD_GLOBAL              3 (abs)
             54 LOAD_GLOBAL              4 (int)
             57 LOAD_GLOBAL              2 (argv)
             60 LOAD_CONST               4 (1)
             63 BINARY_SUBSCR
             64 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             67 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             70 STORE_GLOBAL             6 (initial)

 18      >>  73 LOAD_GLOBAL              7 (print)
             76 LOAD_CONST               5 ('outcome:')
#            79 LOAD_GLOBAL              8 (main35_compute)
             82 CALL_FUNCTION            0 (0 positional, 0 keyword pair)
             85 CALL_FUNCTION            2 (2 positional, 0 keyword pair)
             88 POP_TOP
             89 LOAD_CONST               0 (None)
             92 RETURN_VALUE
```

Figure 6.8: Bytecode for the specialised `main` function, i.e., `_main`.

at line 18 in Figure 1.2 and its disassembly can be seen in Figure 6.9. We can see in this listing that at bytecode offsets 13 and 16, we store the current execution point in a special global variable called `previous_stack`. After the conditional jump, a call to function `failfast` is inserted, which takes the current global dictionary. In this function, if `previous_stack` is `((main, 35), (compute, 6))`, an exception is raised, preempting further execution. In this specialised version of `compute`, we know that `x1`, `x2` and `x3` are not numbers, so the subsequent instructions in the branch would raise a type error. The original function `compute` calls itself recursively. Instead, function `main35_compute` calls yet another specialised version of `compute`, `compute34_compute`. Its can be seen in Figure 6.10. This corresponds to the call site at line 10 in Figure 1.2. In this specialised version, our type analysis infers that `x1` and `x2` can be either a number or of type `NoneType`. Therefore type checks need to be inserted in this case. Therefore in the disassembly, from bytecode offsets 22 to 59, functions named `asserttype` are loaded to check the types of `x2` and `x3`. These functions are defined within the `analyser` class, inside function `emit`. These actually form a closure around their definition site and are specialised for the variables and types that they need to check. These functions take the `locals()` and `globals()` dictionaries as parameter.

```
  5           0 LOAD_GLOBAL              0 (initial)
              3 LOAD_CONST               1 (5)
              6 BINARY_MODULO
              7 LOAD_CONST               2 (0)
             10 COMPARE_OP               2 (==)
#            13 LOAD_CONST               3 (((main, 35), (compute, 6)))
#            16 STORE_GLOBAL             1 (previous_stack)
             19 POP_JUMP_IF_FALSE       69

# 6          22 LOAD_CONST               4 (<function Analyser.emit ...
#                                           failfast at 0x7f78d400b830>)
#            25 LOAD_GLOBAL              2 (globals)
#            28 CALL_FUNCTION            0 (0 positional, 0 keyword pair)
#            31 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
#            34 POP_TOP
             35 LOAD_GLOBAL              3 (int)
             38 LOAD_GLOBAL              4 (input)
             41 LOAD_CONST               5 ('enter final value: ')
             44 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             47 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             50 STORE_FAST               3 (fin)

  7          53 LOAD_FAST                0 (x1)
             56 LOAD_FAST                1 (x2)
             59 BINARY_ADD
             60 LOAD_FAST                2 (x3)
             63 BINARY_ADD
             64 LOAD_FAST                3 (fin)
             67 BINARY_ADD
             68 RETURN_VALUE

  9     >>   69 LOAD_GLOBAL              0 (initial)
             72 LOAD_CONST               6 (1)
             75 INPLACE_SUBTRACT
             76 STORE_GLOBAL             0 (initial)

#10          79 LOAD_GLOBAL              5 (compute34_compute)
             82 LOAD_FAST                1 (x2)
             85 LOAD_FAST                2 (x3)
             88 LOAD_GLOBAL              0 (initial)
             91 CALL_FUNCTION            3 (3 positional, 0 keyword pair)
             94 RETURN_VALUE
             95 LOAD_CONST               0 (None)
             98 RETURN_VALUE
```

Figure 6.9: Bytecode for the specialised `compute` function, i.e., `main35_compute`.

6.8 Variables of interest at each point

In our abstract algorithm for inserting the type checks and for calculating $FailEdge$, we quantify over all variables in $\mathbb{V}$ for any arbitrary point s. In order to implement this, we need to calculate a subset of relevant variables. A naive way to do so is to record all variables that appear in a program. This is inefficient because variables can be unused in certain parts of a program. We note that any instruction that can raise a TypeError in Python would do so because there is an element of the wrong type on the stack. At some point this element has to make its way onto the top of the stack. We exploit this to calculate which variables are of interest at each point. One feature of our analyser is that each time a present or future use type is calculated at a point s, we cache the result to improve the performance. Now, if we go through every execution point s and request the present and future use type of the top of the stack, these requests will translate

```
  5           0 LOAD_GLOBAL              0 (initial)
              3 LOAD_CONST               1 (5)
              6 BINARY_MODULO
              7 LOAD_CONST               2 (0)
             10 COMPARE_OP               2 (==)
#            13 LOAD_CONST               3 (((compute, 34), (compute, 6)))
#            16 STORE_GLOBAL             1 (previous_stack)
             19 POP_JUMP_IF_FALSE       94

# 6          22 LOAD_CONST               4 (<function Analyser.emit ...
#                                           asserttype at 0x7f78d400bcb0>)
#            25 LOAD_GLOBAL              2 (globals)
#            28 CALL_FUNCTION            0 (0 positional, 0 keyword pair)
#            31 LOAD_GLOBAL              3 (locals)
#            34 CALL_FUNCTION            0 (0 positional, 0 keyword pair)
#            37 CALL_FUNCTION            2 (2 positional, 0 keyword pair)
#            40 POP_TOP
#            41 LOAD_CONST               5 (<function Analyser.emit ...
#                                           asserttype at 0x7f78d400bf80>)
#            44 LOAD_GLOBAL              2 (globals)
#            47 CALL_FUNCTION            0 (0 positional, 0 keyword pair)
#            50 LOAD_GLOBAL              3 (locals)
#            53 CALL_FUNCTION            0 (0 positional, 0 keyword pair)
#            56 CALL_FUNCTION            2 (2 positional, 0 keyword pair)
#            59 POP_TOP
             60 LOAD_GLOBAL              4 (int)
             63 LOAD_GLOBAL              5 (input)
             66 LOAD_CONST               6 ('enter final value: ')
             69 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             72 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             75 STORE_FAST               3 (fin)

  7          78 LOAD_FAST                0 (x1)
             81 LOAD_FAST                1 (x2)
             84 BINARY_ADD
             85 LOAD_FAST                2 (x3)
             88 BINARY_ADD
             89 LOAD_FAST                3 (fin)
             92 BINARY_ADD
             93 RETURN_VALUE

  9     >>   94 LOAD_GLOBAL              0 (initial)
             97 LOAD_CONST               7 (1)
            100 INPLACE_SUBTRACT
            101 STORE_GLOBAL             0 (initial)

#10         104 LOAD_GLOBAL              6 (compute34_compute)
            107 LOAD_FAST                1 (x2)
            110 LOAD_FAST                2 (x3)
            113 LOAD_GLOBAL              0 (initial)
            116 CALL_FUNCTION            3 (3 positional, 0 keyword pair)
            119 RETURN_VALUE
            120 LOAD_CONST               0 (None)
            123 RETURN_VALUE
```

Figure 6.10: Bytecode for the second specialised `compute` function, i.e., `compute34_compute`.

to other requests of present and future use types of relevant variables at other positions. Any resultant variables in the cache for a particular point are therefore relevant for calculating the type checks to insert at that point.

6.9 Experiments

We now evaluate our implementation of preemptive type checking for a subset of the Python language on both synthetic examples and selected benchmarks from the computer language benchmarks game [4].

We start by describing the process of adapting these examples and benchmarks to our tool, taking note of the general experience of this process and the usefulness of the information gathered by our tool. We also measure key indicators such as the analysis time of our tool and the number of assertions inserted while varying the tool's parameters. Despite our tool not being optimised for performance, we manage to achieve adequate performance on some typical Python scripts. An important result that we present in this chapter is the effect of varying the execution point depth N on the number of inserted assertions. The full results are tabulated in Figure 6.20.

6.9.1 Synthetic examples

We designed some examples that are small enough to show in full but that also demonstrate the technology behind our tool. In particular, these examples demonstrate the effect of varying the execution point depth on the number of inserted assertions. We subdivide this section according to the different examples. In this section we sometimes show how the result of the source code transformations look. However, we have to keep in mind that no source code transformations take place in practice. Instead all the analysis and transformation takes place at bytecode level.

Example: erasefile

The first example, listed in Figure 6.11, is a small program that redefines variable `x` to be either a string or an integer. Function `mayusenum` may or may not "erase" a file and call a function `usenum` on `x`. This function is similar to intOp in μPython, and is defined such that it fails with a TypeError if the argument passed is not a numeric type. We can see from this example that whenever `erasefile` is called, `usenum` is also subsequently called.

We also list the reverse-engineered transformed code in Figure 6.12. As we can see, when the length of the execution points is one, a type check assertion is inserted in `_mayusenum`. This is because at that point, `x` can be either numeric or a string. If the execution points depth is two, our tool produces two specialised versions of function `mayusenum`. One of these versions, `_mayusenum_2` does not need any assertions since `x` would be numeric in this context. In the

```
1  def erasefile():
2      '''Erases file with filename toerase.'''
3      erase(toerase)
4      print('file erased.')
5
6  def mayusenum():
7      global x,toerase
8      if randbool():
9          toerase='xyz'
10         erasefile()
11         usenum(x)
12
13 def main():
14     global x
15     x=''
16     mayusenum()
17     x=5
18     mayusenum()
```

Figure 6.11: Original listing for example erasefile.

```
def _erasefile():
    '''Erases file with filename toerase
    .'''
    erase(toerase)
    print('file erased.')

def _mayusenum():
    global x,toerase
    if randbool():
        # checks that x is numeric
        asserttype(globals(),locals())
        toerase='xyz'
        _erasefile()
        usenum(x)

def _main():
    global x
    x=''
    _mayusenum()
    x=5
    _mayusenum()
```

```
def erasefile():
    '''Erases file with filename toerase
    .'''
    erase(toerase)
    print('file erased.')

def _mayusenum_1():
    global x,toerase
    if randbool():
        # raises controlled exception
        failfast(...)
        toerase='xyz'
        erasefile()
        usenum(x)
def _mayusenum_2():
    global x,toerase
    if randbool():
        toerase='xyz'
        erasefile()
        usenum(x)

def _main():
    global x
    x=''
    _mayusenum_1()
    x=5
    _mayusenum_2()
```

Figure 6.12: Transformed code for the example erasefile with maximum execution point length 1 (left) and 2 (right).

```
def main():
    erasefile()
    if randbool():
        x='abc'
    else:
        x=34
    if randbool():
        usestr(x)
        usenum(x)
    else:
        usenum(x)
        usestr(x)
```

```
def main():
    failfast(...)
    erasefile()
    if randbool():
        x='abc'
    else:
        x=34
    if randbool():
        usestr(x)
        usenum(x)
    else:
        usenum(x)
        usestr(x)
```

Figure 6.13: Original code (left) compared to transformed code (right) for the example erasefile2.

other version, `x` would be a string and therefore we can raise a controlled exception at line 9 in the transformed code, to preempt execution. If we run our tool setting the execution point depth to 1, we get a warning that a type checking assertion needs to be inserted:

```
File "erasefile.py", line 9, in mayusenum
Variable x expected Number
```

On the other hand, if we set the maximum execution point length to 2, we get different warnings. This time we get a confirmation that a failure will occur if the program runs up to that point. Our error message is more informative and contains part of the stack trace:

```
File "erasefile.py", line 16, in main
File "erasefile.py", line 9, in mayusenum
Variable x expected Number but inferred as str
```

Example: erasefile2

In the previous chapter we have proved that preemptive type checking is at least linearly optimal in terms of error detection. If executing a linear sequence of instructions is bound to raise a TypeError, preemptive type checking will detect this at the entry to this block or earlier. This example however demonstrates that in practice, preemptive type checking can detect type errors earlier than at the entry of a basic block. The example in Figure 6.13 shows a program that is bound to raise a TypeError on any branch that is taken. Our tool transforms the program in such a way as to raise a TypeError at the entry point of the `main` function. Therefore, `erasefile()` is not called at all.

Example: erasefile3

We modify the example erasefile2 to produce the example in Figure 6.14. Here, we have two variables `x` and `y` which are alternatively set either a numeric or string value. Technically, all

```
1  def main():
2      erasefile()
3      if randbool():
4          x='abc'
5          y=34
6      else:
7          x=34
8          y='abc'
9      if randbool():
10         usestr(x)
11         usenum(y)
12     else:
13         usenum(x)
14         usestr(y)
```

Figure 6.14: Listing for the example erasefile3.

runs of this of this program should raise a TypeError. Unfortunately, the earliest point where pre-emptive type checking can preempt this type error is at lines 10 or 13 and therefore `erasefile` gets called on line 2. A model checking approach, where all possible runs of the program are simulated, is necessary to guarantee optimality in general.

Example: fixpoint

This particular example listed in Figure 6.15 shows the advantages of fine-tuning the maximum execution point depth setting of our tool. In function `main`, variables `x1`, `x2` and `x3` are initialised to value `None`. Then, function `fixit` is called. Now, function `fixit` is defined such that it can randomly call `usenum` on `x1`, `x2` and `x3` and return, or propagate the value of `x2` to `x1`, `x3` to `x2`, and set `x3` to be an integer, and then recursively call itself. We can see that if the consequent branch is taken, for the first three calls of `fixit`, the program will raise a TypeError, objecting that `None` is not numeric.

If we now look at the transformed program in Figure 6.16 for execution point length 1, we see that our tool inserts a type checking assertion in the consequent branch that makes sure that `x1`, `x2` and `x3` are numeric. If we increase the execution point length to 4 or higher, no type checking assertions are inserted. Instead, specialised versions that insert a failure assertion at the consequent branch for the first three recursive calls of `fixit` are generated.

Example: introduction

This is the example that was used in the introduction, i.e., shown in Figure 1.2. As in the previous example, we note that the larger the execution point depth, the fewer type checks are inserted.

```
def fixit():
    global x1, x2, x3
    if randbool():
        usenum(x1)
        usenum(x2)
        usenum(x3)
        return
    x1=x2
    x2=x3
    x3=5
    return fixit()

def main():
    global x1,x2,x3
    x1=x2=x3=None
    fixit()
```

Figure 6.15: Original listing for example fixpoint

6.9.2 Real world benchmarks

We tested our implementation on a number of Python benchmarks and examples from the Computer Language Benchmarks Game, a standardised benchmark suite for several languages [4]. This benchmark suite is used in various programming language publications [17, 67, 101, 30, 117, 113]. Although the benchmarks are not large, testing our type checker on these is still a valuable exercise as it can expose certain bugs, scalability and usability issues.

Our implementation does not support the whole Python language. However, it supports enough features to run these programs with minimal changes. For example, since we do not support iterators or generators, all `for` loops were converted to `while` loops. Another cosmetic change is that the main module's body was placed in a function. We also simplified the string formatting operations and provided type information for external functions such as `cout`.

Some benchmarks have been ported to Python from original code in statically typed languages. Type errors should thus be rare. However, in one of the four benchmarks that we analysed, `mandelbrot-python3-3`, which plots the Mandelbrot set on a bitmap, failure assertions were inserted at two different points.

We now describe our experience using our tool on the different examples and benchmarks.

Benchmark: mandelbrot-python3-3

This benchmark is a simple program that plots the Mandelbrot set on a bitmap. The main part of the program is a nested loop where loop variables `x` and `y` iterate through 0 to `size` and then a pixel value is produced. This is the only benchmark that we found that raises a TypeError, due to a tuple of bytes being passed to function `cout` instead of a byte string. The original code of this benchmark can be seen in Figure 6.17 on the left. It is likely that the benchmark was not well tested after porting it from Python 2.x to Python 3.x. We reach this conclusion because one of the main differences between these two versions of Python is that unicode strings and

```
def _fixit():
    global x1, x2, x3
    if randbool():
        # check that x1, x2
        # and x3 are numeric
        asserttype(globals(),locals())
        usenum(x1)
        usenum(x2)
        usenum(x3)
        return
    x1=x2
    x2=x3
    x3=5
    return _fixit()

def _main():
    global x1,x2,x3
    x1=x2=x3=None
    _fixit()
```

```
def _fixit1():
    global x1, x2, x3
    if randbool():
        failfast(...)
        usenum(x1)
        usenum(x2)
        usenum(x3)
        return
    x1=x2
    x2=x3
    x3=5
    return _fixit2()
def _fixit2():
    global x1, x2, x3
    if randbool():
        failfast(...)
        usenum(x1)
        usenum(x2)
        usenum(x3)
        return
    x1=x2
    x2=x3
    x3=5
    return _fixit3()
def _fixit3():
    global x1, x2, x3
    if randbool():
        failfast(...)
        usenum(x1)
        usenum(x2)
        usenum(x3)
        return
    x1=x2
    x2=x3
    x3=5
    return _fixit4()
def _fixit4():
    global x1, x2, x3
    if randbool():
        usenum(x1)
        usenum(x2)
        usenum(x3)
        return
    x1=x2
    x2=x3
    x3=5
    return _fixit4()
def _main():
    global x1,x2,x3
    x1=x2=x3=None
    _fixit1()
```

Figure 6.16: Transformed code for example fixpoint with maximum execution point length 1 (left) and maximum execution point length 4 (right)

```
def main():
  cout = sys.stdout.buffer.write
  size = int(sys.argv[1])
  xr_size = range(size)
  xr_iter = range(50)
  bit = 128
  byte_acc = 0

  cout(("P4\n%d %d\n" % (size, size)).
    encode('ascii'))

  size = float(size)
  for y in xr_size:
    fy = 2j * y / size - 1j
    for x in xr_size:
      z = 0j
      c = 2. * x / size - 1.5 + fy

      for i in xr_iter:
        z = z * z + c
        if abs(z) >= 2.0:
          break
      else:
        byte_acc += bit

      if bit > 1:
        bit >>= 1
      else:
        cout((byte_acc,))
        bit = 128
        byte_acc = 0

    if bit != 128:
      cout((byte_acc,))
      bit = 128
      byte_acc = 0

main()
```

```
cout = sys.stdout.buffer.write
def main():
  bit = 128
  byte_acc = 0
  cout(asciiencode('P4\n%d %d\n'%(size,
    size)))
  size = float(argv[2])
  y=0
  x=0
  while y<size:
    fy = 2j * y / size - 1j
    while x<size:
      z = 0j
      c = 2. * x / size - 1.5 + fy
      i=0
      while i<50:
        z = z * z + c
        if abs(z) >= 2.0:
          break
        i+=1
      else:
        byte_acc += bit

      if bit > 1:
        bit >>= 1
      else:
        cout((byte_acc,))
        bit = 128
        byte_acc = 0
      x+=1

    if bit != 128:
      cout((byte_acc,))
      bit = 128
      byte_acc = 0
    y+=1

a=Analyser(main)
a.printWarnings()
a.emit(globals())
_main()
```

Figure 6.17: Original listing of the mandelbrot-python3-3 code (left) vs. manually modified code (right).

byte strings cannot be interchanged. If a byte string is used in place of a unicode string, a TypeError is raised. The benchmark passes a tuple of bytes instead of passing a `bytearray` or `bytes` object to external function `sys.stdout.buffer.write`. When we ran our tool, it immediately flagged up a warning. The only modifications that were performed for this benchmark to work were to convert the `for` loop into a `while` loop and encapsulate all code in the benchmark into a function `main`. We also simplified the printing operation and pulled the assignment to `cout` outside the main program.

Preemptive type checking detects the possible type failures and outputs the following warnings before executing the `main` function:

```
Failure 1 - partial Traceback:
File "mandelbrot-python3-3.py", line 34, in main
Expected bytes or bytearray but found tuple

Failure 2 - partial Traceback:
```

```
File "mandelbrot-python3-3.py", line 40, in main
Expected bytes or bytearray but found tuple
```

These two failures correspond to the lines `cout((byte_acc,))`. Running the original benchmark (on the left) in Python without preemptive type checking raises a TypeError, with the following output:

```
Traceback (most recent call last):
  File "mandelbrot-python3-3.py", line 37, in <module>
    main()
  File "mandelbrot-python3-3.py", line 33, in main
    cout((byte_acc,))
TypeError: 'tuple' does not support the buffer interface
```

However, with our preemptive type checking analysis we got more precise information regarding the type errors, including a second error where `cout` is called with a tuple.

Benchmark: pidigits-python3-2

This program calculates the first N digits of Pi and prints the digits 10-to-a-line, with the running total of digits calculated. This program adapts the step-by-step Rabinowitz and Wagon's spigot algorithm [45]. In order to adapt this benchmark to our tool, we removed the usage of a math library called MPZ and substituted these operations with operations from the standard library. This did not require a lot of modifications as this library is meant as a drop-in replacement. MPZ provides higher precision math operations than the standard Python library, but the semantics of these operations are essentially the same.

As expected, no type errors were flagged by our tool and no assertions needed to be inserted.

Benchmark: fasta

This program generates DNA sequences, by copying from a given sequence and by weighted random selection from 2 alphabets [4]. It converts the expected probability of selecting each nucleotide into cumulative probabilities, matches a random number against those cumulative probabilities to select each nucleotide and uses this linear congruential generator to calculate a random number each time a nucleotide needs to be selected.

This program makes use of a number of standard library calls to `str.join` and also implements a stateful random number generator. Since we do not support attribute access, we replace calls to `str.join` to an equivalent library call that does the same thing. The random number generator uses a Python language feature called generators, which are similar to coroutines. This is a feature that we do not support. Indeed no Python bytecode analysis tool to our knowledge supports this feature. We get around this problem by adding a type annotation to the random

number generator. This way, the analyser ignores the internal implementation details of the generator and uses the type information from the annotation.

Python also features syntax sugar for string slicing and concatenation. Unfortunately, this is not supported by our tool so we replaced these with the same operations from library functions. A shortcoming of our type system that starts to appear when adapting this benchmark is the lack of polymorphic types. Python programmers tend to make heavy use of Python's available data structures. Being able to infer that, for example, a tuple is not simply a tuple but a 2-tuple of a string and a number could considerably increase the accuracy of our inferred types.

We overcome some of these problems by encapsulating library calls and explicitly adding the type information. The result of losing type information is an increase in inserted assertions that would always succeed. We tested this benchmark on our tool and we can see from the results in Figure 6.20 that no type checks were inserted. This means that this benchmark does not have any type errors.

Benchmark: meteor-contest

This program finds solutions to the Meteor Puzzle board [4]. This is made up of 10 rows of 5 hexagonal Cells. There are 10 puzzle pieces to be placed on the board. Each puzzle piece is made up of 5 hexagonal Cells. At 206 lines of code, it is the largest one tested. Although this might seem like a small program, it is the length of a typical Python script. In comparison, our entire tool is implemented in less than 800 lines of Python code. Another thing to keep in mind is that the C++ version of this benchmark is 500 lines of code. We also have to keep in mind that our analysis can be applied to particular functions rather than to the whole program.

This particular benchmark uses more functions from the standard library than the others we tested. Type information was added to these external functions. One of the difficulties in handling this benchmark is the use of data structures containing anonymous functions. These functions are contained in a dictionary, retrieved at runtime and are repeatedly applied in a loop. Since our control flow analysis mechanism is not sophisticated enough to determine which functions are called, we resort to manually unrolling the loop.

A possible failure was statically inferred by our analyser for this benchmark. This occurred in function `findFreeCell`, in Figure 6.18. Our tool produced the following output:

```
Failure 1 - partial Traceback:
File "meteor.py", line 49, in findFreeCell
Expected tuple but found NoneType
...
```

However, when running this benchmark, no type errors were encountered. When preemptive type checking was turned on, no preemptive type errors were raised either. This means that the program execution never reached beyond line 49 in `findFreeCell`. We can immediately

```
45 def findFreeCell(board):
46     for y in range(height):
47         for x in range(width):
48             if board & (1 << (x + width*y)) == 0:
49                 return x,y
```

Figure 6.18: Code snippet from meteor-contest showing possible type error. There are branches where this function does not return a free cell.

see however that if no free cells are found in a board, this function will not return anything. A Python function that does not return anything by default returns `None`. This means that if nothing is returned a type error would occur, as `None` cannot be unpacked in the same way as a tuple. The programmer is therefore assuming an invariant that asserts that a "free cell" will always be found in the "board". Our tool explicitly inserts an assertion that the loop will never terminate without returning from within the loop. If this program is run using preemptive type checking, a preemptive type checking error is raised as soon as the loop at line 47 exits. For this benchmark, all the inserted type checks are expected to hold and some of them occur because type information is lost when items are inserted into lists and retrieved again.

Question: stackoverflow

We firmly believe that preemptive type checking can be of help especially to programmers who are just starting to learn the language. This is especially true if the analysis can be used in a tool that can issue warnings prior to running the code. We therefore tested our implementation on code that was posed by a Python beginner on stackoverflow.com.[1] The code can be seen in Figure 6.19. This user complained that the program initially seems to run and that half way through the interaction with it a TypeError is raised. In this program, the reason a type error occurs is that user input is of type Str but this input is being used in a mathematical expression without converting it to a number.

Since this question was based on Python 2.x, we had to run the standard `2to3` toolchain to automatically convert this to Python 3.x syntax. When we analysed our program using preemptive type checking, our implementation statically produced warnings that corroborate the answer given to this question by Python developers. In particular it indicates all the locations where a type error would occur.

```
Failure 1 - partial Traceback:
File "stackoverflow.py", line 36, in main
Variable level expected Number but found str

Failure 2 - partial Traceback:
File "stackoverflow.py", line 39, in main
Variable level expected Number but found str

Failure 3 - partial Traceback:
File "stackoverflow.py", line 42, in main
```

[1] http://stackoverflow.com/questions/320827/python-type-error-issue

```
status = 1

print "[b][u]magic[/u][/b]"

while status == 1:
    print " "
    print "would you like to:"
    print " "
    print "1) add another spell"
    print "2) end"
    print " "
    choice = input("Choose your option: ")
    print " "
    if choice == 1:
        name = raw_input("What is the spell called?")
        level = raw_input("What level of the spell are you trying to research?")
        print "What tier is the spell: "
        print " "
        print "1) low"
        print "2) mid"
        print "3) high"
        print " "
        tier = input("Choose your option: ")
        if tier == 1:
            materials = 1 + (level * 1)
            rp = 10 + (level * 5)
        elif tier == 2:
            materials = 2 + (level * 1.5)
            rp = 10 + (level * 15)
        elif tier == 3:
            materials = 5 + (level * 2)
            rp = 60 + (level * 40)
        print "research ", name, "to level ", level, "--- material cost = ",
                materials, "and research point cost =", rp
    elif choice == 2:
        status = 0
```

Figure 6.19: Code that raised type errors submitted by a stackoverflow user.

```
Variable level expected Number but found str
```

6.10 Results

In this section we summarise the results of all experiments. These were conducted on an otherwise idle Intel Xeon W3520 workstation running at 2.67GHz. All times given are in milliseconds measured by system calls to get the current time. We set the cutoff time to one hour. We measure the performance of our tool using the following criteria:

Analysis time. This is the time required for our tool to perform the analysis of the program. This includes a control flow analysis, type inference and the calculation of assertions and fail edges. Naturally, since the control flow graph is potentially larger if the maximum length of the execution points is increased, we expect the analysis time to be longer.

Transformation time. This is the time required to take the current program and, given the information gathered from the analysis phase, transform the program and all functions

called from within this program. We expect that the transformation time will depend on the size of the control flow graph.

CFG size. This is the number of nodes in the control flow graph. As the execution point depth is increased, the CFG is also expected to become larger. The size of the CFG also depends on the size of the program.

Number of dynamic checks in the specialised functions. This is the number of estimated type checks present in the code, if performed by the standard interpreter. For example, in the statement $f(x, y)$, a check is made to see whether f is a callable function. If f is a standard library function that expects x and y to have particular types, a check is made for every argument. Therefore this statement requires 3 type checks. We estimate the number of dynamic checks by accumulating this *for every instruction associated with a node* in the CFG.

Number of fail edges. This is the number of failing assertions inserted, i.e., the number of edges in the control flow graph beyond which the program is guaranteed to fail. Failing assertions that raise a controlled exception introduce no runtime overhead as any of these is typically only executed once, if ever.

Number of inserted checks. The number of inserted type checks in the specialised code. Type checking assertions can potentially introduce runtime overheads and therefore the fewer of these need to be inserted, the better.

We present the full results of our benchmarks in Figure 6.20. An important result that we note is that failing assertions are only inserted within the original code if the original code contains latent type errors. From the results in Figure 6.20, we can see that for most Python modules, the performance of the analyser is adequate. In fact, we are able to analyse a program more than 30,000 nodes in the CFG in under half an hour. Inevitably, this program analysis and transformation step will increase the initialisation time, just as a JIT compiler would increase the initialisation time of a program. Most of the runtime of our tool is spent on the control flow analysis and the type inference stages. We expect that the algorithms used can be re-implemented in a faster manner and using a faster programming language, as Python is around $80\times$ slower than C[4].

Another important result that we note is that when the maximum execution point depth is increased, the number of fail edges increases and the number of inserted checks decreases relative to the original number of checks. Ideally, we do not want our type checking mechanism to insert any type checks, as these increase the computation required to run the code. On the other hand, assertions that always fail do not impose a computation expense on the program, as these are typically only executed once, if ever. Therefore, our results are positive because they show that if more computation is dedicated in the analysis phase, the modified program has a smaller number of type checks to compute at runtime.

max. exec. point length	analysis time (ms)	transformation time (ms)	CFG size	dynamic checks	fail edges	inserted checks
erasefile, 23 lines of code						
1	59	0	38	12	0	1
2	89	1	54	18	1	0
3	100	1	62	22	1	0
4	101	1	62	22	1	0
erasefile2, 24 lines of code						
1	50	0	43	17	1	0
2	52	0	43	17	1	0
3	53	0	43	17	1	0
4	53	0	43	17	1	0
fixpoint, 24 lines of code						
1	82	0	40	11	0	3
2	155	1	67	21	1	2
3	249	2	94	31	2	1
4	360	3	121	41	3	0
introduction, 21 lines of code						
1	197	1	67	39	0	3
2	278	2	97	57	1	2
3	414	3	127	75	2	1
4	526	4	157	93	3	0
stackoverflow, 48 lines of code						
1	295	2	157	82	3	0
2	931	3	157	82	3	0
3	298	2	157	82	3	0
4	294	3	157	82	3	0
pidigits-python3-2, 40 lines of code						
1	257	2	131	62	1	0
2	257	2	131	62	1	0
3	257	2	131	62	1	0
4	258	2	131	62	1	0
mandelbrot-python3-3, 46 lines of code						
1	312	2	128	59	2	0
2	313	2	128	59	2	0
3	312	2	128	59	2	0
4	316	2	128	59	2	0
fasta, 96 lines of code						
1	358	2	154	87	0	0
2	631	4	208	120	0	0
3	631	4	208	120	0	0
4	634	5	208	120	0	0
meteor, 206 lines of code						
1	9764	16	813	409	1	10
2	42822	44	1719	869	1	35
3	247689	210	6357	3215	1	179
4	1510357	1294	30945	15587	1	1043

Figure 6.20: Table of results.

Our tool is implemented as a prototype, and therefore its performance and scalability should not be used to judge the suitability of preemptive type checking. We believe preemptive type checking can successfully be implemented for languages that are similar to Python or new languages designed with this type checking mechanism in mind. Our particular tool can still be used on medium sized scripts or critical parts of larger programs, as long as only a limited subset of Python is used. From the performance figures in the table, we note that our tool is very usable for programs up to 200 lines of code. Looking at the table in Figure 6.20, we can easily note that the analysis time seems to be a function of the number of the CFG and the lines of code.

If we discount the control flow analysis step, the reason why our analysis does not scale so well is that a global analysis is very expensive. If we could ignore global variables, our tool would be much faster. We could mitigate this problem by performing an escape analysis for every global variable and discounting large parts of the control flow graph where a global variable is not reassigned. We also ran a profiler on our analyser to determine where most of the time is being spent and it appears that around 10 to 20 percent of the time is being spent calculating the hash code of objects such as instructions, execution points, edges and the like. This is because we rely on set and dictionary operations for most of our algorithms. This would be a perfect kind of application where a system that can generate optimised hashing operations [47] would increase the performance.

6.11 Conclusions

In this chapter we have shown how to write a type checker that implements preemptive type checking, implemented as a Python 3.3 library that can be loaded with the target program. It can be used at runtime to transform an existing function. When this function is then executed, it is executed using preemptive type checking.

Although our implementation is not yet a production quality tool, we have shown that it is possible to implement preemptive type checking. In a language such as Python, we do not even need to modify the interpreter but to implement the tool as a library that can be imported into the users' code. The implementation itself was coded in less than 800 lines of code.

We have also evaluated our implementation on both synthetic examples and selected benchmarks from the computer language benchmarks game [4]. We have shown that all our high level research objectives, defined in Section 1.3 have been met.

We have shown that our simple tool can handle small to medium Python scripts that do not use object oriented features. We have seen how these real world scripts can be manually edited in case these use unsupported features. An important result that we present in this chapter is the relation between the maximum execution point length and the number of inserted assertions. In all the examples and benchmarks used in this chapter, all type errors were caught in advance. Our tool does not produce any false positives, i.e., if a program did not raise any type errors

before using our tool, it did not raise any controlled exceptions when the program was run with preemptive type checking.

Chapter 7

Conclusion and Future work

In this dissertation, we have introduced a new method for type checking dynamically typed programs that combines elements of both static and dynamic type checking. It is described as *preemptive type checking* since the actual type checks happen much earlier than in dynamic typing. We have proven that any program that can run to completion under dynamic typing without raising a type error will also not raise any errors under preemptive type checking. We have also demonstrated an implementation for a subset of Python and have evaluated it on some synthetic examples and also on some benchmarks from the computer languages benchmarks game [4].

In this last chapter we summarise the main contributions, propose further work directions, and conclude our research on preemptive type checking.

7.1 Main contributions

The main contribution of this thesis is the concept of type error preemption for dynamically typed languages. Our type checking mechanism tries to preempt all type errors at the earliest possible point for which a type error is inevitable. This is the most novel contribution, and a problem that we have effectively solved in our work. Preemptive type checking is the only type checking mechanism in which any program that can run to completion under dynamic typing without raising a type error will also not raise any errors under it. We have made a number of smaller but equally novel contributions in order to make it possible to create a type checking mechanism as described in this dissertation.

Our first contribution in Chapter 3 is a small Python like language which we call μPython. This includes the formalisation and definition of μPython source code, together with its bytecode language, compiler and the semantics of its bytecode.

Our second contribution is a type system and analysis technique for μPython. We have proven that the type information obtained by the type analysis is an overapproximation of the actual

types that would appear at runtime. A part of this contribution is the development of the concept of *present* and *future use* types. The distinction between these types can be leveraged in two ways. Firstly, this allows the insertion of type checks at earlier points, which we exploit in preemptive type checking. Secondly, error messages can explain in greater detail why some code should not be allowed to run. Another novel contribution is the concept of trails as part of the inference mechanism. Trails will always guarantee the termination of the type inference algorithm. This is true irrespective of the structure of the type language. Trails are particularly suited when inferring the type of a variable in situations such as the example in the introduction (see Figure 1.1).

We have also contributed to the area of control flow analysis. We introduce the concept of abstracting nodes in the control flow graph as call stacks that are truncated to a finite depth. The memory overhead for low execution point depths N is minimal with an efficient representation, as used in our implementation. When a depth of one is selected, the nodes effectively become program locations.

Our implementation shows that it is possible to build a tool that is sophisticated enough to perform the kinds of analysis and program transformations required to support preemptive type checking. To make this possible, there are several innovations in our implementation. Primarily, we perform a "static analysis at runtime", which can be performed once during initialisation to offset any performance issues. Another innovation is that we substitute the bytecode of functions in memory with specialised versions that have type checks inserted into them. In the future, this significant machinery could be further leveraged to perform performance optimisations by techniques such as partial evaluation.

7.2 Future work directions

Throughout our work, we have mostly focused on type error preemption. However, the machinery developed to make this possible can be easily extended to offer other useful features.

Enhanced error reporting

One of the first extensions is that of passing around the execution point at which a type was introduced with all present and future use types. Using this information, whenever a type incompatibility is detected a more informative error message can be constructed. This could indicate the source (the original assignment) and the sink (the use). For example, the following program on the left could be transformed into the program on the right, where more information is given in the error message.

```
1
2
3
4 x='4'
5 ..
6 ..
7 intOp(x)
```

```
raise PreemptiveTypeError(
  'Expected int at line 7, but
   found a str at line 4')
x='4'
..
..
intOp(x)
```

Runtime type inference - metaprogramming and reflection

Python is not a metaprogramming language in the same sense as Lisp [69], MetaML [104], Jumbo [58] or MetaAspectJ [119] as it is not designed to manipulate syntactic program structures. However, most use cases for a metaprogramming language can be handled using Python's metaprogramming and reflective features such as metaclasses, `eval` and `exec`. There are also other functions such as `getattr` or `setattr` that can alter an arbitrarily named field, determined at runtime. Modules can also be dynamically loaded, and different modules would have differently typed functions.

Because of these features, in general one cannot determine which modules are being imported. We propose that instead of invoking the type checking mechanism once, we can continuously update the type information as metaprogramming operations are invoked. In this way, if a function is dynamically created, we can apply the type inference to it and also insert more assertions at runtime. One could describe this as "just-in-time type checking".

There are some design decisions we already take in order to accommodate for this feature. Since some of the type inference has to be performed at runtime, where we can only inspect the bytecode, this is the natural way to perform type inference. One other design decision is to implement the type inference in the same language as the target language. This facilitates the interaction between the target program and the type inference. To support newly created functions, the algorithm in Figure 6.7 needs to be extended. When dispatching over the functions that have been specialised for a particular point, a newly created function that has not been analysed may be found. In this case, the analysis process needs to be called again on this new function.

If we adopt this functionality, we can support metaprogramming and reflection. We now show some of the functionality available in Python to make this possible.

First class functions and decorators. Functions or classes can be passed around like regular objects. This creates the opportunity to write *decorator* functions. These are higher-order functions that given a function return the same function wrapped with some extra concerns such as logging or caching. There is also syntactic support for decorators [100]. Figure 7.1 shows a Python decorator that performs simple memoisation.

```
def simplememo(fn):
    '''Ignores any arguments to a function'''
    cachename='_'+fn.__name__
    def memofn(self,*args):
        if not hasattr(self,cachename):
            setattr(self,cachename,fn(self,*args))
        return getattr(self,cachename)
    return memofn
```

Figure 7.1: A Python decorator.

Class and function generation. Apart from function decorators, functions and classes can also be directly manipulated. This has been useful in our implementation.

A function's document string, bytecode and argument list can be introspected and new functions and methods can be created. These features however are seldom used, as these require an advanced understanding of the internal details of Python. These features are also not fully supported by alternative Python implementations such as PyPy [87], Jython [57] or IronPython [118].

A commonly used feature however is to rebind methods in classes and objects. This effectively changes the behaviour of the objects at runtime. The rebound methods can be created from existing functions or by wrapping existing functions with extra logic. This is usually done using class decorators [114]. This has been used to implement `AutoEq` and `AutoEqImmutable` in our implementation, see Appendix.

Reflection and Introspection. Having a variable number of positional and keyword arguments makes it possible to create functions with a dynamic interface that can change programmatically. For example, it is possible to substitute any arbitrary function by writing another function that can morph its interface according to the original function. This is quite useful for testing, especially for generating mock objects, classes or functions.

A tool at our disposal for creating classes with a "dynamic" interface is that of descriptors, which are special functions in an object or class that control the way attributes are loaded. These are also responsible for *binding* functions to particular objects, thus effectively turning simple functions into *bound methods*. A typical way to offer a dynamic interface is to override the `__getattr__()` and `__setattr__()` methods when defining a class. When an attribute of an object is accessed using the dot operator, a call is made to the `__getattr__()` method in the object and the request can then be handled by it.

A typical use case for overriding methods such as `__getattr__()` is the implementation of generic proxy classes. A generic proxy class can intercept any requests and forward these, perhaps via a network link. The actual classes are called using the equivalent Python functions `getattr` and `setattr`. This does not require the physical generation of class stubs based on the actual classes.

eval() and exec(). The functions `eval()` and `exec()` evaluate and execute Python expressions and statements respectively. The input to these functions can be either a string representing Python code, an AST object, or a code object that represents compiled bytecode. Since strings can be created programmatically, these two functions clearly give Python metaprogramming capabilities.

Generating source code as Python strings and passing these strings to functions such as `eval()` is a very crude way to make use of metaprogramming. The use of these features is generally discouraged [86], especially when working in a team.

Metaclasses. Metaclasses in Python [56] are classes that inherit from the class `type`. Classes in Python are instances of `type`. The class body is evaluated just as a function block. The resulting local variables are placed in a dictionary like object as dictated by the metaclass and referred to as the class dictionary. By default this is a dictionary that does not preserve order. A metaclass (for example `type`) is then called and the class name, a list of base classes and the class dictionary are passed to the constructor. The constructor is responsible for creating the actual class.

It is possible to override the constructor of a metaclass. This gives the programmer power to change the semantics of class creation, and ultimately of object-oriented programming. Metaclasses are rarely used in practice, as most functionality can usually be achieved using class decorators. The main difference between metaclasses and class decorators is that class decorators do not allow any changes to the type of data structure used by the class dictionary.

Dynamic code loading. All imports in Python are done dynamically. The `import` statement in Python dynamically loads the module and executes it line by line. The resulting names are loaded into a new namespace or the current namespace, depending on the usage of the `import` statement. This feature allows a program to selectively load the required modules depending on the current context and allows a good level of flexibility that traditionally requires metaprogramming.

Supporting other languages and more language features

Our proof of concept implementation of preemptive type checking was specifically written for Python. It is a challenge to create a type system that supports object oriented programming in this language. Although there are type systems that support object oriented programming in JavaScript [11, 106], these are not necessarily applicable to Python. Python's object system makes it extremely difficult to statically infer any information about objects, due to features such as descriptors. Preemptive type checking can however be easily applied to JavaScript. The semantics of JavaScript [48] is better understood than Python's, and static type systems have been applied to JavaScript with various degrees of success [11, 106, 49].

```
1  class A:
2      def foo(self):
3          return 'a'
4  class B:
5      def foo(self):
6          return 34
7  testset={A(),B()} # put A and B objects in a set
8  min(t.foo() for t in testset) # find the minimum value of foo()
9  .
10 .
11 TypeError: unorderable types: str() < int()
```

Figure 7.2: Calling a method on objects in a set.

So far, our type inference supports local and global variables, basic control flow, the runtime stack and function declarations (nested or otherwise). There are however other features that would make the supported language more usable.

Built-ins: It would be beneficial to manually type a larger subset of built-ins. This simply involves adding a module with initial types of the built-ins.

Objects and closures: Objects are similar to records, but, since Python is a dynamically typed language, these do not have a rigid structure. In fact, if any function is called and a particular object is accessible within the scope (perhaps it is passed as argument), any of its fields could be mutated. To give better support for object mutation, closures ideally should be supported as well. Introducing structural types [80] to the type language might be a way to support objects.

Parametric polymorphism: The type language, together with the type operations such as meet and join can be extended to support parametric polymorphism.

Containers and iterators: Most loops in Python are performed over iterators. In order to support these language features, we need to investigate whether our type system could support parametric polymorphism first.

Control flow analysis: The analysis carried out by preemptive type checking is parametric with respect to the control flow analysis. The better the control flow analysis, the better the results we can get. Ultimately, the reason why a lot of features of Python are not supported in our implementation is that we cannot perform a proper control flow analysis. In the example in Figure 7.2, at line 8, it is difficult to determine that the result from `foo()` can be either a string or an integer. The control flow analysis must determine that the call to `foo` to any object in `testset` can be either `foo` defined at line 2 or `foo` at line 5.

Type-directed runtime optimisation

A lot of machinery has been developed to make preemptive type checking possible. Some of this machinery can be leveraged to increase the performance of the Python code that has been analysed. Since this analysis is carried out at runtime, some of the functions and values in the

global variables would be resolved. Also, we can use the results from our type inference process to generate optimised versions of the functions under test. We could start with some simple optimisations on the bytecode itself, such as:

- Partially evaluating the bytecode with the runtime information available and the information inferred from our sophisticated static analysis.
- Inlining functions; our bytecode emitter can be modified to do so.
- Eliminating dead code.

These optimisations are easy to implement with the current machinery but will not yield a tremendous performance increase. We expect a much higher performance increase if instead of emitting bytecode, we could generate C code that implements the functionality of the bytecode. This would interact with the original program via the Python C API. Generating C code is only possible if the correct types of the variables and a call graph can be resolved. In effect, we could turn our implementation into a Just-In-Time compiler. This extension to our work would entail a modest amount of engineering effort, but this way the cost of developing a control flow analyser and type analyser for preemptive type checking can be amortised with the performance advantages that JIT compilation brings.

7.3 Concluding Remarks

Dynamically typed languages are used in a wide range of sophisticated applications, such as JavaScript on the browser, web servers or on NoSQL databases. Languages such as Python or JavaScript are used to implement full-sized enterprise applications [74], daemons and also a number of server side scripts. The reliance on this kind of languages has increased significantly throughout the last 10 years and is projected to increase even further. Traditionally, programs could be type-checked either statically or dynamically. The latter method is by definition the only option available for dynamically typed languages such as Python as computing a static type safety guarantee is not possible. However, the former method could give a type safety guarantee before even running the program. With preemptive type checking, we have broken the dichotomy between static and dynamic type checking, by trying to push the type checking as early as possible.

Our work is not the first attempt to reconcile both facets of static and dynamic typing. For example, gradual typing [96] makes it possible to mix static and dynamically typed languages together. However, gradual typing does not guarantee anything about the dynamically typed portion of a program. Soft typing [22] introduces the concept of narrowing functions, whereby a dynamically typed program is transformed into a statically typed program with explicit casts. This, however, reduces the expressiveness of the language. The existing implementations [116]

have not been applied to dynamically typed languages with a global state and re-definable functions.

We have developed a new method for type checking dynamically typed programs, where we try to preempt type errors as early as possible. Amongst other things, this helps the programmer find type errors in his code. Programs can raise controlled exceptions much earlier in case of a type error, thus reducing the time required to test a system. We have also demonstrated how such a system can be effectively implemented for a subset of the real Python language and shown its usefulness on some examples.

References

[1] Mars Climate Orbiter Mishap Investigation Board Phase I Report. Technical report, NASA, 1999.

[2] ECMA-262: ECMAScript Language Specification. Technical report, ECMA International, 2011.

[3] TIOBE Programming Community Index. Technical report, TIOBE Software, 2012.

[4] The Computer Language Benchmarks Game, 2013.

[5] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[6] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type Inference of SELF: Analysis of Objects with Dynamic and Multiple Inheritance. In *Proceedings of ECOOP*, pages 247–267, 1993.

[7] Amal Ahmed, Robert Bruce Findler, Jacob Matthews, and Philip Wadler. Blame for all. In *Proceedings of STOP*, 2009.

[8] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proceedings of POPL*, pages 163–173, 1994.

[9] Jong-hoon An, Avik Chaudhuri, and Jeffrey S. Foster. Static Typing for Ruby on Rails. In *Proceedings of ASE*, pages 590–594, November 2009.

[10] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Proceedings of DLS*, pages 53–64, 2007.

[11] Christopher Anderson and Paola Giannini. Towards type inference for JavaScript. In *Proceedings of ECOOP*, pages 428–452, 2005.

[12] John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, June 2003.

[13] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of PLDI*, volume 35, pages 1–12, 2000.

[14] Ira D. Baxter, Christopher Pidgeon, and Mehlich Mehlich. DMS: program transformations for practical scalable software evolution. In *Proceedings of ICSE*, pages 625–634, 2004.

[15] Gavin Bierman, Erik Meijer, and Mads Torgersen. Adding dynamic types to C#. In *Proceedings of ECOOP*, 2010.

[16] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25. ACM, 2009.

[17] Carl Friedrich Bolz and Laurence Tratt. The Impact of Meta-Tracing on VM Design and Implementation. *Science of Computer Programming*, 2013.

[18] Alan H. Borning and Daniel H. H. Ingalls. A Type Declaration and Inference System for Smalltalk. In *Proceedings of POPL*, pages 133–141, 1982.

[19] Gilad Bracha. Pluggable type systems. In *OOPSLA workshop on revival of dynamic languages*, 2004.

[20] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings of OOPSLA*, pages 215–230, 1993.

[21] Luca Cardelli. Type systems. *ACM Computing Surveys*, 26(1), 1996.

[22] Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of PLDI*, pages 278–292, 1991.

[23] Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for JavaScript. *ACM SIGPLAN Notices*, 47(10):587, November 2012.

[24] Alonzo Church. A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic*, 5(2):56, June 1940.

[25] Consel Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proceedings of POPL*, pages 493–501, 1993.

[26] Pascal Costanza. Dynamic vs. Static Typing — A Pattern-Based Analysis. *Proceedings of Workshop on Object-oriented Languages*, 2004.

[27] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of POPL*, 1977.

[28] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of POPL*, pages 207–212, 1982.

[29] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[30] Matthew Davis, Peter Schachte, Zoltan Somogyi, and Harald Sondergaard. Towards region-based memory management for Go. In *Proceedings of MSPC*, pages 58–67, 2012.

[31] Rodrigo B. de Oliveira. The Boo programming language, 2005.

[32] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct blame for contracts: no more scapegoating. In *Proceedings of POPL*, pages 215–226, 2011.

[33] Bruce Eckel. Strong Typing vs. Strong Testing. In Joel Spolsky, editor, *The Best Software Writing I*, pages 67–77. Apress, 2005.

[34] Evan R. Farrer. A Quantitative Analysis of Whether Unit Testing Obviates Static Type Checking for Error Detection. *M.Sc Thesis, California State University*, 2011.

[35] Jerome A. Feldman. *A Formal Semantics for Computer-Oriented Languages*. PhD thesis, Carnegie Institute of Technology, 1964.

[36] Mathias Felleisen. From Soft Scheme to Typed Scheme: 20 Years of Scripts to Program Conversion (Invited talk). *Proceedings of STOP*, 2009.

[37] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of ICFP*, pages 48–59, September 2002.

[38] Cormac Flanagan. Hybrid type checking. In *Proceedings of POPL*, pages 245–256, 2006.

[39] Python Software Foundation. *Python*. Available online: http://www.python.org.

[40] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. In *Proceedings of OOPSLA*, pages 283–300, 2009.

[41] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *Symposium on Applied Computing*, pages 1859–1866, 2009.

[42] Yoshihiko Futamura. Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation*, pages 381–391, 1999.

[43] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boriz Zbarsky, Jason Orendorff, Jesse Rederman, Edwin Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of PLDI*, pages 465–478, 2009.

[44] Andreas Gal and Michael Franz. Incremental dynamic code generation with trace trees. *Technical Report ICS-TR*, 2006.

[45] Jeremy Gibbons. Unbounded Spigot Algorithms for the Digits of Pi. *The Mathematical Association of America*, 2005.

[46] Adele Goldberg, David Robson, and Michael Harrison. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1983.

[47] Neville Grech, Julian Rathke, and Bernd Fischer. JEqualityGen: generating equality and hashing methods. In *Proceedings of GPCE*, 2010.

[48] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *Proceedings of ECOOP*, pages 1–25, 2010.

[49] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing local control and state using flow analysis. *Programming Languages and Systems*, pages 256–275, 2011.

[50] Jungwoo Ha, Mohammad R. Haghighat, Shengnan Cong, and Kathryn S. McKinley. A concurrent trace-based just-in-time compiler for JavaScript. In *Proceedings of OOPSLA*, 2009.

[51] Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for JavaScript. *Proceedings of PLDI*, pages 239–250, 2012.

[52] Robert Harper, Bruce F. Duba, and David Macqueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(04):465–480, November 1993.

[53] Phillip Heidegger and Peter Thiemann. Recency Types for Dynamically-Typed, Object-Based Languages. In *Proceedings of FOOL*, 2009.

[54] Fritz Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994.

[55] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of OOPSLA*, pages 132–146, 1999.

[56] David Joiner. Python Enhancement Proposal (PEP) 3115 – Metaclasses in Python 3000. *Available online: http://www.python.org/dev/peps/pep-3115/*, 2007.

[57] Juneau Juneau, Jim Baker, Leo Soto, Victor Ng, and Frank Wierzbicki. *The definitive guide to Jython: Python for the Java platform*. Springer-Verlag, 2010.

[58] Sam Kamin, Lars Clausen, and Ava Jarvis. Jumbo: run-time code generation for Java and its applications. In *Code generation and optimization: feedback-directed and runtime optimization*, pages 48–56, 2003.

[59] Andrew John Kennedy. *Programming languages and dimensions*. PhD thesis, University of Cambridge, 1996.

[60] Andrew John Kennedy. Relational parametricity and units of measure. In *Proceedings of POPL*, 1997.

[61] Joshua Kerievsky. *Refactoring to patterns*. Addison-Wesley, 2005.

[62] Dénes Knig. Theorie der Endlichen und Unendlichen Graphen: Kombinatorische Topologie der Streckenkomplexe. *Leipzig: Akad. Verlag.*, 1936.

[63] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of International Symposium on Code Generation and Optimization*, number c, pages 75–86, 2004.

[64] Barbara Liskov and Stephen Zilles. Programming with abstract data types. *ACM Sigplan Notices*, pages 50–59, 1974.

[65] Mika V. Mantyla and Casper Lassenius. What types of defects are really discovered in code reviews? *IEEE Transations on Software Engineering*, 35(3):430–448, 2009.

[66] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In *Proceedings of ICFP*, pages 136–149, August 1997.

[67] Stefan Marr and Theo D'Hondt. Identifying a unifying mechanism for the implementation of concurrency abstractions on multi-language virtual machines. *Objects, Models, Components, Patterns*, 2012.

[68] Yukihiro Matsumoto. *Ruby*. Available online at: www.ruby-lang.org, 1995.

[69] John McCarthy. Recursive functions symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, April 1960.

[70] Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *Proceedings of Workshop on Revival of Dynamic Languages*, 2004.

[71] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, Hemel Hempstead, 1992.

[72] Microsoft. TypeScript, 2012.

[73] Matt Might, Yannis Smaragdakis, and David Van Horn. Resolving and Exploiting the k-CFA Paradox. In *Proceedings of PLDI*, pages 305–315, 2010.

[74] Tommi Mikkonen and Antero Taivalsaari. Using JavaScript as a Real Programming Language. 2007.

[75] David C. Morrill. Traits: A new way of adding properties to Python classes. In *Proceedings of PyCon*. Available online at: http://code.enthought.com/projects/traits/, 2003.

[76] MSDN. Using Type dynamic (C# Programming Guide), 2011.

[77] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer-Verlag, 1999.

[78] Sven-Olof Nyström. A soft-typing system for Erlang. In *Proceedings of Erlang Workshop*, pages 56–71, 2003.

[79] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Inc, 1st edition, November 2008.

[80] Benjamin C. Pierce. Nominal and Structural Type Systems. In *Types and programming languages*, chapter 19, pages 247–264. The MIT Press, first edition, 2002.

[81] Benjamin C. Pierce. *Types and programming languages*. The MIT Press, first edition, 2002.

[82] François Pottier. A modern eye on ML type inference. *Lecture notes for the APPSEM Summer School*, 2005.

[83] Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 27(6):1047–57, June 2000.

[84] Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The Ins and Outs of gradual type inference. In *Proceedings of POPL*, pages 481–494, New York, New York, USA, 2012. ACM Press.

[85] Didier Remy. Typechecking records and variants in a natural extension of ML. In *Proceedings of POPL*, pages 77–88, 1989.

[86] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do. In *Proceedings of ECOOP*, pages 52–78, 2011.

[87] Armin Rigo and Samuele Pedroni. PyPy's approach to virtual machine construction. In *Proceedings of OOPSLA*, pages 944–953. ACM, 2006.

[88] John Rose. JSR 292: Supporting Dynamically Typed Languages on the Java Platform.

[89] Bertrand Russell. *The principles of mathematics*. WW Norton & Company, 1996.

[90] Michael Salib. *Starkiller: A Static Type Inferencer and Compiler for Python*. Masters Thesis, Department of Electrical Engineering and Computer Science, MIT., 2004.

[91] Manuel Serrano and Pierre Weis. Bigloo: a portable and optimizing compiler for strict functional languages. *Static Analysis*, 1995.

[92] Tim Sheard and James Hook. Type safe meta-programming. Technical report, Oregon Graduate Institute, 1994.

[93] Olin Shivers. Control-flow analysis in Scheme. In *Proceedings of PLDI*, pages 164–174, 1988.

[94] Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, 1991.

[95] Olin Shivers. Higher-order control-flow analysis in retrospect: lessons learned, lessons abandoned. *ACM SIGPLAN Notices*, 39(4):257–269, 2004.

[96] Jeremy Siek and Walid Taha. Gradual typing for functional languages. In *Proceedings of Scheme and Functional Programming Workshop*, 2006.

[97] Jeremy Siek and Walid Taha. Gradual typing for objects. In *Proceedings of ECOOP*, 2007.

[98] Jeremy Siek and Manish Vachharajani. Gradual typing with unification-based inference. In *Proceedings of DLS*, 2008.

[99] Jeremy Siek and Philip Wadler. Threesomes, with and without blame. In *Proceedings of STOP*, pages 34–46, New York, New York, USA, 2009. ACM Press.

[100] Kevin D. Smith, Jim J. Jewett, Skip Montanaro, and Anthony Baxter. Python Enhancement Proposal (PEP) 318 – Decorators for Functions and Methods. *Available online: http://www.python.org/dev/peps/pep-0318/*, 2003.

[101] Vincent St-Amour, Sam Tobin-Hochstadt, and Matthias Felleisen. Optimization coaching: optimizers learn to communicate with programmers. In *Proceedings of OOPSLA*, pages 163–178, 2012.

[102] Rok Strniša, Peter Sewell, and Matthew Parkinson. The Java module system: core design and semantic definition. In *Proceedings of OOPSLA*, volume 277, pages 499–513, 2007.

[103] Gerald Jay Sussman and Guy L Steele Jr. Scheme: An interpreter for extended lambda calculus. *MEMO 349, MIT AI LAB*, 1975.

[104] Walid Taha. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, October 2000.

[105] Satish R. Thatte. Quasi-static typing. In *Proceedings of POPL*, pages 367–381, 1989.

[106] Peter Thiemann. Towards a type system for analyzing javascript programs. In *Programming Languages and Systems*, pages 408–422, 2005.

[107] Christian Tismer. Continuations and Stackless Python. In *Proceedings of PyCon*, 2000.

[108] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *Proceedings of DLS*, pages 964–974, 2006.

[109] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *Proceedings of POPL*, pages 395–406, 2008.

[110] Laurence Tratt. Dynamically typed languages. *Advances in Computers*, March 2009.

[111] Michael M Vitousek, Shashank Bharadwaj, and Jeremy G Siek. Towards Gradual Typing in Python, 2012.

[112] Philip Wadler and R Findler. Well-typed programs can't be blamed. *Programming Languages and Systems*, pages 0–31, 2009.

[113] Kevin Williams, Jason McCandless, and David Gregg. Dynamic interpretation for dynamic scripting languages. *Trinity College Dublin, Department of Computer Science, Technical Report*, 2009.

[114] Collin Winter. Python Enhancement Proposal (PEP) 3129 – Class Decorators. *Available online: http://www.python.org/dev/peps/pep-3129/*, 2007.

[115] Collin Winter and Tony Lownds. Python Enhancement Proposal (PEP) 3107. *Available online: http://www.python.org/dev/peps/pep-3107/*, 2006.

[116] Andrew K. Wright and Robert Cartwright. A practical soft type system for scheme. In *ACM Transactions on Programming Languages and Systems*, volume 19, pages 87–152, January 1997.

[117] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In *Proceedings of POPL*, pages 377–388, 2010.

[118] Junfeng Zhang. IronPython: A fast Python implementation for .NET and Mono. In *Proceedings of PyCon*, 2004.

[119] David Zook, Shan Shan Huang, and Yannis Smaragdakis. Generating AspectJ Programs with Meta-AspectJ. In *Proceedings of GPCE*. Springer-Verlag, October 2004.

Implementation listing

In this appendix we include the implementation of the type checking mechanism. It includes everything except the bytecode parsing and repackaging module, a library of typed functions, and some utilities. We have not included a similarly sized module with unit and integration tests.

```
from itertools import chain
from collections import defaultdict
from util import *
from mytypes import *
import random

PREVIOUS_STACK='previous_stack'
MAX_STACK=8

class TypeErrorAssertion(AssertionError): pass

def typetostr(t):
    'Returns a "pretty" textual representation of a type.'
    if isinstance(t,type):
        return str(t).split("'")[1]
    if isinstance(t,str):
        return t
    if isinstance(t,set):
        return ' or '.join(typetostr(tt) for tt in t)
    assert False,t

def pointtostr(point):
    res=[]
    for p,pc in point:
        for i in range(pc,-1,-1):
            op,operand=p.code[i]
            if op==byteplay.SetLineno:
                res.append('File "%s", line %d, in %s'%(p.filename,operand,p.name))
                break
    return '\n'.join(res)

def typejoin(*types):
    '''Returns a join of the types given as arguments.'''
    # optimization, chose 2 on purpose
    if len(types)==1: return types[0]
    res=set()
    for t in types:
        if isinstance(t,set):
            res|=t
        else:
            res.add(t)
```

```
        if Top in res: return Top
        res-={Bot}
        if len(res)==1: return res.pop()
        if not res: return Bot
        return res

    def typemeet(u,v):
        '''Returns a meet of the types given as arguments.'''
        if u == v: return u
        if u==Top: return v
        if v==Top: return u
        res=(u if isinstance(u,set) else {u}) & (v if isinstance(v,set) else {v})
        if not res: return Bot
        if len(res)==1: return res.pop()
        return res

    def skipinvalid(point):
        *rst, (p,pc)=point
        while not isopcode(p.code[pc][0]):
            pc+=1
        assert pc<len(p.code)
        rst.append((p,pc))
        return tuple(rst)


    class Name:
        '''Base class for variable names/stack position hierarchy'''
        isstack,islocal,isglobal=False,False,False
        def shift(self,n):
            return self

    @AutoEQImmutable
    class VName(Name):
        '''Base class for a variable name.'''
        def __init__(self,x):
            self.x=x
        def __key__(self):
            yield self.x
        def __repr__(self):
            return self.x
    class Global(VName): isglobal=True
    class Local(VName): islocal=True

    @AutoEQImmutable
    class StackOffset(Name):
        '''Represents the offset on a stack.'''
        isstack,islocal,isglobal=True,False,False
        def __init__(self,n):
            self.n=n
        def shift(self,n):
            if n==0: return self
            else: return StackOffset(self.n+n)
        def __key__(self):
            yield self.n
        def __str__(self):
            if self.n:
                return 'StackOffset(%d)'%self.n
            else: return ''
        __repr__=__str__
```

```
tos=StackOffset(0)
tos1=StackOffset(1)

class BaseInst:
    checks=0
    changesprog=False
    calls=False
    def copyp(self):
        pass
    def copyf(self):
        pass
    def __init__(self,analyser,s):
        self.analyser=analyser
        self.s=s

class FirstInst(BaseInst):
    '''Represents the instruction at first execution point (),
    which in reality does not exist.'''
    calls=True
    def gtp(self,x):
        if not x.isglobal:
            return Un
        # x is a global
        globs=self.analyser.globs
        if x.x in globs:
            val=globs[x.x]
        elif x.x in self.analyser.builtins:
            val=self.analyser.builtins[x.x]
        else:
            return Un
        return self.analyser.gettype(val)

    def gtf(self,x):
        return Top

    def _getnext(self):
        return [self.analyser.initpoint]

    def _getnextloc(self):
        return [None]
class LastInst(BaseInst):
    '''Represents the instruction at first execution point None,
    which in reality does not exist.'''
    def gtf(self,x):
        return Top
    def gtp(self,x):
        return Un
    def _getnext(self):
        return []
    _getnextloc=_getnext


class Inst(BaseInst):
    '''This class represents a generic instruction object.
    All subclasses of Inst correspond to actual bytecode instructions.

    This contains all the functionality for inferring the types at its curent
    point.'''
```

```
    def __init__(self,analyser,s):
        super().__init__(analyser,s)
        (p,pc)=self.s[-1]
        self.operand=p.code[pc][1]
    def shiftp(self,x):
        '''Depending on the stack shifting of the current instruction
        shifts the stack offset for the forwards analysis'''
        return x.shift(-self.stackshift)
    def shiftf(self,x):
        '''Depending on the stack shifting of the current instruction
        shifts the stack offset for the backwards analysis'''
        return x.shift(self.stackshift)
    def getfunction(self,n):
        '''Returns the function loaded at stack position n'''
        assert n!=0
        a=self.analyser
        prev=list(a.getprevloc(self.s))
        assert len(prev)==1,prev
        prev=a.getinst(prev[0])
        return prev.getfunction(n-self.stackshift)

    def _getnext(self):
        '''Default implementation that returns the next execution point.
        By default this is the next instruction in program order.'''
        *next, (p,pc)=self.s
        next.append((p,pc+1))
        return [skipinvalid(next)]
    _getnextloc=_getnext
    def gtp(self,x):
        if x.isstack and (x.n<0 or x.n>=MAX_STACK):
            return Un
        return self.analyser.envp.get((self.s,x),Bot)
    def gtf(self,x):
        if x.isstack and (x.n<0 or x.n>=MAX_STACK):
            return Top
        return self.analyser.envf.get((self.s,x),Bot)
    def copyf(self):
        a=self.analyser
        envf=a.envf
        res=self._copyf()
        for x in a.vars:
            envf[self.s,x]=(res[x] if x in res
                            else self.gtfnext(self.shiftf(x)))
    def _copyf(self):
        return {}
    def copyp(self):
        a=self.analyser
        envp=a.envp
        res=self._copyp()
        for x in a.vars:
            spot=self.s,x
            if x in res:
                envp[spot]=res[x]
            else:
                x=self.shiftp(x)
                if x.islocal or x.isstack:
                    envp[spot]=self.gtpprevloc(x)
                else:
                    envp[spot]=self.gtpprevglob(x)
```

```
219     def _copyp(self):
            return {}
        def gtpprevloc(self,x):
            '''Returns a union of the p types of x at the previous points
            on the intra procedural control flow graph'''
            a=self.analyser
            s=self.s
            if a.isentry(s):
                if x.islocal:
                    args=s[-1][0].args
229                 name=x.x
                    if name in args:
                        index=len(args)-1-args.index(name)
                        prev=a.getprev(s)
                        return typejoin(*[a.getinst(s_).gtpprevloc(StackOffset(index))
                                          for s_ in prev])
                return Un
            prev=a.getprevloc(s)
            return typejoin(*[a.getinst(s_).gtp(x) for s_ in prev])

239     def gtpprevglob(self,x):
            '''Returns a union of the p types of x at the previous points
            on the inter procedural control flow graph'''
            a=self.analyser
            prev=a.getprev(self.s)
            return typejoin(*[a.getinst(s_).gtp(x) for s_ in prev])

        def gtfnext(self,x):
            a=self.analyser
            nxt=a.getnext(self.s)
249         return typejoin(*[a.getinst(s_).gtf(x) for s_ in nxt])
        def __repr__(self):
            return '%s %s at %s'%(type(self),self.operand,self.s)
        __str__=__repr__

    class LOAD_CONST(Inst):
        stackshift=1
        def getfunction(self,n):
            if n!=0:
                return super().getfunction(n)
259         return self.operand

        def _copyp(self):
            return {tos:self.analyser.gettype(self.operand)}

    class MAKE_FUNCTION(Inst):
        stackshift=-1
        def _copyp(self):
            return {tos:Callable}
    class DUP_TOP(Inst):
269     stackshift=1
        def _copyf(self):
            return {tos:typemeet(self.gtfnext(tos),self.gtfnext(tos1))}
    class CALL_FUNCTION(Inst):
        calls=True
        @property
        def checks(self):
            return 1+(0 if self.changesprog else self.operand)
        @property
```

```
    def stackshift(self):
        return -self.operand
    def getfunction(self,n):
        assert n!=0
        a=self.analyser
        prev=list(a.getprevloc(self.s))
        assert len(prev)==1,prev
        prev=a.getinst(prev[0])
        return prev.getfunction(n+self.operand)

    @simplememo
    def getcalledfn(self):
        a=self.analyser
        prev=a.getprevloc(self.s)
        assert len(prev)==1,self.s
        res=a.getinst(list(prev)[0]).getfunction(self.operand)
        return res

    @property
    def changesprog(self):
        return not getattr(self.getcalledfn(),'__annotations__',False)

    def _getnext(self):
        if not self.changesprog:
            return self._getnextloc()
        # get called function
        f=self.getcalledfn()
        p_=ExtraCode.from_code(f.__code__)
        return [skipinvalid((self.s+((p_,0),))[-self.analyser.accuracy:])]

    def _copyp(self):
        a=self.analyser
        fn=self.getcalledfn()
        if getattr(fn,'__annotations__',False):
            return {tos:fn.__annotations__['return']}
        else:
            nxt=a.getnextloc(self.s)
            assert len(nxt)==1
            nxt=list(nxt)[0]
            return {tos:a.getinst(nxt).gtpprevglob(tos)}
    def _copyf(self):
        fn=self.getcalledfn()
        a=self.analyser
        s=self.s
        n_args=self.operand
        res={}
        if hasattr(fn,'__code__'):
            args=ExtraCode.from_code(fn.__code__).args
            assert len(args)==n_args,fn
            for n in range(n_args):
                name=args[n_args-n-1]
                if getattr(fn,'__annotations__',False):
                    assert name in fn.__annotations__,'all or none'
                    res[StackOffset(n)]=fn.__annotations__[name]
                else:
                    res[StackOffset(n)]=self.gtfnext(Local(name))
        res[StackOffset(n_args)]=Callable
        for x in a.vars:
            if x.islocal or (x.isstack and x.n>n_args):
```

```
                res[x]=typejoin(*[a.getinst(nxt).gtf(self.shiftf(x))
                                  for nxt in a.getnextloc(self.s)])
        return res


class POP_JUMP_IF_FALSE(Inst):
    checks=1
    stackshift=-1
    def _getnext(self):
        return super()._getnext()+JUMP_ABSOLUTE._getnext(self)
    _getnextloc=_getnext

class STORE_GLOBAL(Inst):
    stackshift=-1
    def _copyp(self):
        return {Global(self.operand):self.gtpprevglob(tos)}
    def _copyf(self):
        x=Global(self.operand)
        return {x:Top, tos:self.gtfnext(x)}
class STORE_FAST(Inst):
    stackshift=-1
    def _copyp(self):
        return {Local(self.operand):self.gtpprevglob(tos)}
    def _copyf(self):
        x=Local(self.operand)
        return {x:Top,tos:self.gtfnext(x)}
class LOAD_GLOBAL(Inst):
    stackshift=1
    def _copyf(self):
        x=Global(self.operand)
        return {x:typemeet(self.gtfnext(tos),self.gtfnext(x))}
    def _copyp(self):
        return {tos:self.gtpprevglob(Global(self.operand))}
    def getfunction(self,n):
        if n!=0:
            return super().getfunction(n)
        a=self.analyser
        f=self.operand
        if f in a.globs:
            return a.globs[f]
        if f in a.builtins:
            return a.builtins[f]
        raise Exception('%s not found'%f)
class LOAD_FAST(Inst):
    stackshift=1
    def _copyf(self):
        x=Local(self.operand)
        return {x:typemeet(self.gtfnext(tos),self.gtfnext(x))}
    def _copyp(self):
        return {tos:self.gtpprevloc(Local(self.operand))}

class POP_TOP(Inst): stackshift=-1
class NOP(Inst): stackshift=0
class POP_BLOCK(NOP): pass
class SETUP_LOOP(NOP): pass
class BREAK_LOOP(NOP):
    def _getnext(self):
        *rst, (p,pc)=self.s
        for _pc in range(pc-1,0,-1):
```

```
            if p.code[_pc][0]==byteplay.SETUP_LOOP:
                label=p.code[_pc][1]
                for _pc in range(pc+1,len(p.code)):
                    if p.code[_pc][0]==label:
                        return [skipinvalid(rst+[(p,_pc)])]
                assert False
        assert False
        _getnextloc=_getnext

class JUMP_ABSOLUTE(NOP):
    def _getnext(self):
        *rst, (p,pc)=self.s
        for i,label in enumerate(p.code):
            if label==(p.code[pc][1],None):
                rst.append((p,i))
                return [skipinvalid(rst)]
        assert False
    _getnextloc=_getnext
class JUMP_FORWARD(JUMP_ABSOLUTE): pass
class POP_JUMP_IF_TRUE(POP_JUMP_IF_FALSE): pass
class RETURN_VALUE(Inst):
    stackshift=0
    changesprog=True
    def _getnext(self):
        a=self.analyser
        *start, (p,pc)=self.s
        start.append((p,0))
        return chain(*(a.getinst(fro)._getnextloc() for fro in a.getprev(skipinvalid(
    start))))
    def _getnextloc(self):
        return []
    def _copyf(self):
        return dict((x,Top) for x in self.analyser.vars
                 if x.islocal or (x.isstack and x.n>0))
class INPLACE_ADD(Inst):
    checks=2
    stackshift=-1
    def _copyp(self):
        return {tos:Number}

    def _copyf(self):
        return {tos:Number,tos1:Number}

class BINARY_MODULO(Inst):
    checks=2
    stackshift=-1
    def _copyp(self):
        return {tos:self.gtpprevloc(tos1)}
    def _copyf(self):
        return {tos:{bytes,Number,str,tuple},tos1:{Number,str}}


BINARY_OR=BINARY_LSHIFT=BINARY_RSHIFT=BINARY_AND=INPLACE_RSHIFT=BINARY_FLOOR_DIVIDE=
    INPLACE_SUBTRACT=BINARY_TRUE_DIVIDE=BINARY_MULTIPLY=INPLACE_MULTIPLY=BINARY_POWER=
    BINARY_SUBTRACT=BINARY_LSHIFT=BINARY_ADD=INPLACE_ADD

class UNARY_NEGATIVE(Inst):
    stackshift=0
    checks=1
```

```
        def _copyp(self):
            return {tos:Number}
        def _copyf(self):
            return {tos:Number}

    class BUILD_TUPLE(Inst):
        @property
459     def stackshift(self):
            return -self.operand+1
        def _copyp(self):
            return {tos:tuple}
        def _copyf(self):
            return dict((StackOffset(n),Top) for n in range(self.operand))

    class UNPACK_SEQUENCE(Inst):
        checks=1
        @property
469     def stackshift(self):
            return self.operand-1

        def _copyp(self):
            return dict((StackOffset(n),Top) for n in range(self.operand))
        def _copyf(self):
            return {tos:tuple}

    class COMPARE_OP(INPLACE_ADD):
        def _copyp(self):
479         return {tos:bool}
    class BINARY_SUBSCR(Inst):
        stackshift=-1
        checks=1
        def _copyp(self):
            return {tos:Top}
        def _copyf(self):
            return {tos:Number,tos1:{MutableSequence,str}}

    class STORE_SUBSCR(Inst):
489     stackshift=-3
        def _copyf(self):
            return {tos:Number,tos1:MutableSequence,StackOffset(2):Top}


    class Analyser:
        '''This class is the entry point for the analysis.'''
        def __init__(self,main, accuracy=2):
            self.accuracy=accuracy
            self.main=main
499         self.globs=main.__globals__
            self.builtins=__builtins__
            self.initpoint=((ExtraCode.from_code(self.main.__code__),1),)
            self.localedges=set()
            self.nextlocaldict=defaultdict(set)
            self.prevlocaldict=defaultdict(set)
            self.edges=set()
            self.nextdict=defaultdict(set)
            self.prevdict=defaultdict(set)
            self.getnext=self.nextdict.__getitem__
509         self.getprev=self.prevdict.__getitem__
            self.getnextloc=self.nextlocaldict.__getitem__
```

```
        self.getprevloc=self.prevlocaldict.__getitem__
        self.points={(),self.initpoint,None}
        self.trail=set()
        self.envp={}
        self.envf={}
        self.failedges={}
        self.instdict={():FirstInst(self,()),None:LastInst(self,None)}
        self.assertions=defaultdict(list)
        self.vars=set()
        self.calcedges()
        for p in self.points:
            inst=self.getinst(p)
            if isinstance(inst,(STORE_GLOBAL,LOAD_GLOBAL)):
                self.vars.add(Global(inst.operand))
            if isinstance(inst,(STORE_FAST,LOAD_FAST)):
                self.vars.add(Local(inst.operand))
        for n in range(MAX_STACK):
            self.vars.add(StackOffset(n))
        self.calcassertions()

    def printwarnings(self):
        i=1
        for warn in self.failedges.values():
            print('Failure',i,'- partial Traceback:')
            print(warn)
            print()
            i+=1
        i=1
        for (fro,to),ass in self.assertions.items():
            print('Assertion ',i)
            print(pointtostr(to))
            for x,tp,t in ass:
                print('Variable %s inferred %s but expected %s'%(x,typetostr(tp),
typetostr(t)))
            i+=1
    def isentry(self,s):
        return s[-1][1]==1 and self.getinst(list(self.getprev(s))[0]).calls
    def gettype(self,val):
        for typ in (Callable,Number,MutableSequence,type(val)):
            if isinstance(val,typ): return typ

    def getinst(self,point):
        si=self.instdict
        if point not in si:
            p,pc=point[-1]
            si[point]=globals()[str(p.code[pc][0])](self,point)
        return si[point]

    def calcedges(self):
        '''This method constructs the global inter-procedural CFG
        and local intra-procedural CFGs'''
        oldlen=-1
        spa=self.points.add
        points=self.points
        sea=self.edges.add
        se=self.edges
        sn=self.nextdict
        sp=self.prevdict
        slea=self.localedges.add
```

```
569         sle=self.localedges
            sln=self.nextlocaldict
            slp=self.prevlocaldict
            while oldlen!=len(se):
                oldlen=len(se)
                for point in set(points):
                    for nextpoint in self.getinst(point)._getnext():
                        edge=(point,nextpoint)
                        if edge in se:
                            continue
579                     spa(nextpoint)
                        sea(edge)
                        sn[point].add(nextpoint)
                        sp[nextpoint].add(point)
                    if sln[point]:
                        continue
                    for nextpoint in self.getinst(point)._getnextloc():
                        edge=(point,nextpoint)
                        if edge in sle:
                            continue
589                     slea(edge)
                        sln[point].add(nextpoint)
                        slp[nextpoint].add(point)
            # optimisation, instructions have all been constructed
            # replace factory with call to dictionary
            self.getinst=self.instdict.get


        def emit(self,globs):
            newfns=set()
599         b=byteplay # alias
            insertions=defaultdict(list)
            def insertss(s):
                # inserts before
                insertions[s]+=[
                    (b.LOAD_CONST,s),
                    (b.STORE_GLOBAL,PREVIOUS_STACK)
                ]
            def insertfail(s,sprev):
                failmsg=self.failedges[sprev,s]
609             def failfast(globs):
                    if globs[PREVIOUS_STACK]!=sprev:
                        return
                    raise TypeErrorAssertion(failmsg)
                insertions[s]+=[
                    (b.LOAD_CONST,failfast),
                    (b.LOAD_GLOBAL,'globals'),
                    (b.CALL_FUNCTION,0),
                    (b.CALL_FUNCTION,1),
                    (b.POP_TOP,None)
619             ]
            def insertassert(s,assertion,sprev):
                def asserttype(globs,locs):
                    x,_,t=assertion
                    if globs[PREVIOUS_STACK]!=sprev:
                        return
                    dic=locs if x.islocal else globs
                    if x.x not in dic:
                        tr=Un
```

```
                else:
                    tr=self.gettype(dic[x.x])
                if typemeet(tr,t)!=Bot:
                    return
                raise TypeErrorAssertion('Future type error due to %s at %s, expected
    %s got %s'%(x,s,typetostr(t),typetostr(tr)))
            insertions[s]+=[
                (b.LOAD_CONST,asserttype),
                (b.LOAD_GLOBAL,'globals'),
                (b.CALL_FUNCTION,0),
                (b.LOAD_GLOBAL,'locals'),
                (b.CALL_FUNCTION,0),
                (b.CALL_FUNCTION,2),
                (b.POP_TOP,None)
             ]
        def getfname(s):
            *rst,(p,pc)=s
            fname='_'.join(p.name+str(pc) for p,pc in rst)
            fname+='_'+p.name
            return fname

        for point in self.points:
            if point:
                *rst,(p,pc)=point
                rst.append(p)
                newfns.add(tuple(rst))
        # insert marker at entry point
        insertions[self.initpoint]+=[
                (b.LOAD_CONST,()),
                (b.STORE_GLOBAL,PREVIOUS_STACK)
        ]
        for edge in self.edges:
            fro,to=edge
            if edge in self.failedges:
                for s in self.getprev(to):
                    insertss(s)
                insertfail(to,fro)
            if edge in self.assertions:
                for s in self.getprev(to):
                    insertss(s)
                for assertion in self.assertions[edge]:
                    insertassert(to,assertion,fro)
        self.delfns={PREVIOUS_STACK}
        self.checks=0
        for fn in newfns:
            *rst,p=fn
            newbc=ExtraCode.from_byteplay_code(p)
            # create new bytecode
            newbc.code=[]
            for pc in range(len(p.code)):
                s=tuple(rst+[(p,pc)])
                if isopcode(p.code[pc][0]):
                    if self.getinst(s):
                        self.checks+=self.getinst(s).checks
                newbc.code+=insertions[s]
                # change called function reference
                op,operand=p.code[pc]
                if op==b.CALL_FUNCTION:
                    _s=list(self.getnext(s))[0]
```

```
                    if self.isentry(_s):
                        # change called function
                        newbc.code[-operand-1]=(b.LOAD_GLOBAL,getfname(_s))
689                 newbc.code.append(p.code[pc])
                self.delfns.add(getfname(s))
                globs[getfname(s)]=FunctionType(newbc.to_code(),globs)

    def clearfns(self,globs):
        for fn in self.delfns:
            if fn in globs:
                del globs[fn]
    def calcassertions(self):
        # compute all environments
699     def traverse(point,allpoints,prefn,nxtfn,gt):
            if point not in allpoints:
                return
            gt(point)
            allpoints.remove(point)
            for point in nxtfn(point):
                traverse(point,allpoints,prefn,nxtfn,gt)
        oldenvp=None
        iterations=0
        while oldenvp!=self.envp:
709         iterations+=1
            oldenvp=dict(self.envp)
            # traverse forwards direction
            allpoints=set(self.points)
            traverse((),allpoints,self.getprev,self.getnext,
                 lambda p : self.getinst(p).copyp())
            assert not allpoints
#        print('fixpoint p in %d iterations'%iterations)
        del oldenvp
        oldenvf=None
719     while oldenvf!=self.envf:
            oldenvf=dict(self.envf)
            # traverse backwards direction
            traverse(None,set(self.points),self.getnext,self.getprev,
                 lambda p : self.getinst(p).copyf())
        del oldenvf
        gi=self.getinst
        # calculate failedge
        for edge in self.edges:
            fro,to = edge
729         froinst=gi(fro)
            for x in self.vars:
                if x.islocal and froinst.changesprog: continue
                if x.isstack and x.n>0: continue
                _tf=gi(to).gtf(x)
                if _tf==Top: continue
                tp=froinst.gtp(x);tf=froinst.gtf(x)
                if tf!=_tf and typemeet(tp,_tf)==Bot:
                    self.failedges[edge]='%s\nVariable %s expected %s but found %s'%(
                        pointtostr(to),x,typetostr(_tf),typetostr(tp))
739     # add to failedge
        oldsize=0
        while oldsize!=len(self.failedges):
            oldsize=len(self.failedges)
            for edge in self.edges:
                if edge in self.failedges:
```

```
                continue
            # if all next edges are failedges then this one is
            fro,to=edge
            if to is not None and all((to,nxt) in self.failedges
749                 for nxt in self.getnext(to)):
                self.failedges[edge]='\n'.join({
                    self.failedges[(to,nxt)]
                    for nxt in self.getnext(to)})
    # calculate assertions to insert
    # fail edges is at its maximum here
    for edge in self.edges:
        if edge in self.failedges:
            continue
        fro,to=edge
759     if len(self.getnext(fro))==1:
            continue
        froinst=gi(fro)
        toinst=gi(to)
        for x in self.vars:
            if x.islocal and froinst.changesprog: continue
            if x.isstack: continue
            _tf=toinst.gtf(x)
            tp=froinst.gtp(x);tf=froinst.gtf(x)
            meet=typemeet(tp,_tf)
769         if tf!=_tf and meet!=tp and meet!=Bot:
                self.assertions[edge].append((x,tp,meet))
    # remove redundant failedge
    oldsize=9999999999
    oldfailedges=dict(self.failedges)
    while oldsize!=len(self.failedges):
        oldsize=len(self.failedges)
        for edge in dict(self.failedges):
            # if all previous edges are failedges then this one
            # need not be
779         fro,to=edge
            if fro!=() and all(
                (prev,fro) in oldfailedges
                for prev in self.getprev(fro)):
                self.failedges.pop(edge)
```