Preemptive type checking

Neville Grech[a,b], Bernd Fischer[c], Julian Rathke[d]

[a] *University of Athens*
[b] *University of Malta*
[c] *University of Stellenbosch*
[d] *University of Southampton*

Abstract

Dynamically typed languages are very well suited for rapid prototyping, agile programming methodologies and rapidly evolving software. However, programmers can still benefit from the ability to detect type errors in their code early, in particular if this does not impose restrictions on their programming style.

In this paper we describe a new type checking system that identifies potential type errors in such languages through a flow-sensitive static analysis. 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, the mechanism inserts type checks at strategic points in the original program. We prove that these checks are inserted as early as possible and 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. Instead, it can be invoked at a very late stage, after the compilation to bytecode and initialisation of the program. We demonstrate an implementation of this for the Python language, and its effectiveness on a number of standard benchmarks.

Keywords: Program Analysis, Program Transformation, Type Theory, Dynamic Typing

1. Introduction

Dynamically typed languages such as Python are some of the most popular languages in use today [1]. In these languages type errors are typically not detected prior to execution. Instead, they manifest themselves as runtime errors or exceptions, causing systems to fail. However, the earlier type errors are detected, the earlier the program can be corrected, and it has indeed been shown that programs that are written in a dynamically-typed language tend to contain more latent errors than those that are written in statically-typed languages [2].

Figure 1 shows an example Python program taken and slightly adapted from the official tutorial for the Spark platform.[1] The program processes large CSV-files: it splits the file into fields, counts the number of individual field occurrences in a *map-reduce* style, sorts the (field, count)-pairs in parallel, using a number `tasks` of processors, and finally saves the results. It uses a generic configuration loading library `app.getConfig` (not shown here) to set up its parameters. However, the program can fail with a runtime type error in `sortByKey` if `app.getConfig` does not return an integer value as its third return value, but this error manifests itself (if at all) only after the expensive file input and map-reduce operations.

A defensive developer would therefore insert a runtime type check before the computation of `counts`, but in a large system the number of type checks to be inserted can become overwhelming, and their optimal

Email addresses: `me@nevillegrech.com` (Neville Grech), `bfischer@cs.sun.ac.za` (Bernd Fischer), `jr2@ecs.soton.ac.uk` (Julian Rathke)

[1] Available at `https://github.com/apache/spark/blob/master/examples/src/main/python/wordcount.py`

```
 1  @preemptive
 2  def main() {
 3      (url_in, url_out, tasks) = app.getConfig()
 4
 5      file = sc.textFile(url_in)
 6
 7      counts = file.flatMap(lambda line: line.split(",")) \
 8                  .map(lambda word: (word, 1))            \
 9                  .reduceByKey(lambda a, b: a + b)        \
10                  .sortByKey(numTasks = tasks)
11
12      counts.saveAsTextFile(url_out)
13  }
```

Figure 1: Adapted Python Spark word count example program with a possible type error at line 10.

```
 1  from sys import argv
 2
 3  def compute(x1=None,x2=None,x3=None):
 4      global initial
 5      if initial==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(int(input('enter initial value: ')))
16      else:
17          initial=abs(int(argv[1]))
18      print('outcome: ',compute())
19
20  if __name__=='__main__':
21      main()
```

Figure 2: Synthetic Python example with type errors for initial value inputs '2' and '1'.

placement can be difficult to decide. In this paper, we describe an approach that automatically inserts such runtime type checks. More specifically, we introduce the notion of *preemptive type checking* 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 but before it actually happens; in some cases, this can be even before the program execution starts. The analysis 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 "doomed" [3], i.e., guaranteed to raise a TypeError exception.

In the example, preemptive type checking infers that the successful execution of the `sortByKey` operation depends on the value stored in `tasks` being an integer. In addition, it infers that it can insert this type check within the `getConfig` method directly, specialized for the invocation at line 3. Preemptive type checking also informs the user about the presence of these type checks (as warnings). This is useful to help the developer discover unintended behaviors in his program.

Proponents of dynamically typed languages often argue that type errors can be detected by unit testing. Preemptive type checking can also help to shorten this testing process. For instance, consider the (synthetic) example program shown in Figure 2. This needs to be tested thoroughly in order to find all its type errors. When we run the program without preemptive type checking, we notice that depending on the input, the program will either raise a TypeError exception or work as expected, for example:

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

$ python foo.py
```

```
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 3: Transformed version of the `compute` function.

```
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'
```

Clearly, finding the exact sequence of inputs to find type errors, even in this simple program, can prove to be a challenge. Standard test case generation techniques, such as boundary value analysis, do not help in this case either.

With preemptive type checking we can reduce the testing effort and identify type errors much quicker. Our analysis infers that `x1` and `x2` are each independently either of type `NoneType` or integers, depending on the control flow taken by the program. The 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 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 can see that for this particular example there are no false positives and all errors can occur when this program is executed. We will show that this is indeed the case for all non-diverging programs.

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 the function `compute` three times. In this paper we describe an effective technique that uses *trails* (see Section 3.3) to perform a flow-sensitive type inference. An important aspect of preemptive type checking is that it transforms the `compute` function so that any type errors are preempted (see Figure 3). Hence, our generated runtime checks also statically indicate the locations where type errors can potentially originate. In contrast, there are no implementations of soft typing [4] or gradual typing [5] that handle this example. For example, gradual typing implementations would insert a type check right before the additions of `x1` and `x2`, but this means that the user input still has to take place before the type error can be raised. However, these type systems aim to solve other problems than type error preemption and actual implementations could potentially combine both gradual typing and preemptive type checking.

Preemptive type checking can also entirely subsume a "must-fail" version of static type checking, i.e., type errors that will manifest themselves in any run of the program. Consider for example a different (but also wrong) version of the main method of the previous example:

```
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())
```

In this version the program will start executing but, depending on the arguments passed to the program, fail at either line `main:15` or or line `main:17`, due to missing type conversions at these lines. For example, using the standard Python interpreter we can observe the following behavior:

```
$ 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'
```

The runtime environment raises a TypeError when line 15 is executed.

In this example, preemptive type checking transforms the program as follows: `main` function is 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 prevents the program from executing at all and reduces the time required to test this program since in this case no user input needs to be given for the error to be raised. Notice also that the type check is inserted at an optimal point, i.e., at the earliest point where we can detect that the program will raise a type error, which in this case is at the beginning of the function. However, note that this is a special case of preemptive type checking. We will mostly be concerned with type errors that only occur under some runs.

Contributions. In summary, in this paper, we make four main contributions:

Inference model for present/future use types: We give a detailed description and the inference rules of a new type checking system that identifies potential type errors in dynamically typed languages through a flow-sensitive static analysis. Its core idea is to compute 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.

Type error preemption: We describe an algorithm that takes the inferred past and future types and inserts runtime type checks that force the termination of the program execution as soon as it can be detected that a type error is inevitable but before it actually happens.

Proofs of correctness and optimality: We formally prove, for a bytecode-based dynamically typed core calculus modeled on Python, that our inference rules are sound (i.e., that the inferred past types are over-approximations of the runtime types). We also prove that the type checks are inserted as early as possible and 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.

Implementation and evaluation: We sketch a publicly available implementation of the preemptive type checking algorithm as a Python 3.3 library, and give details of its application to a number of real-world application examples.

This paper is a revised and extended version of an earlier conference contribution that appeared in the proceedings of ICTAC 2013 [6]. The main differences are that we now give full formal proofs for all theorems in Section 3 and Section 4. We also present a worked example in Section 5 that better illustrates the approach and a more detailed evaluation of the approach in Section 7.

Outline. We proceed by describing μPython, a core calculus of the dynamically-typed language Python in Section 2, and formalising the type system and the corresponding bytecode level type inference, including correctness proofs, in Section 3. We describe the theoretical details of the type checking mechanism in Section 4 and present a worked example in Section 5. Although the theory is presented for μPython the techniques presented are applicable for any similar dynamically typed language, or indeed larger subsets of Python as in our implementation. We describe an implementation of preemptive type checking in Section 6, including assertion insertion, and evaluate it on a number of real-world Python benchmarks in Section 7. We discuss related work in Section 8 and conclude in Section 9.

2. The μPython language

In this section 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 analysis.

High-Level Syntax. For illustrative purposes we present in Figure 4 a high-level syntax of μPython, but the actual type analysis is performed at the bytecode level. The primitive types of the language are standard except perhaps for the types Un of uninitialised variables (i.e., the type of U) and Fn of functions. μPython supports function definitions, conditional statements, assignments, and while loops. In μPython, expressions are either function calls, constants, or variables. Valid expressions are also valid statements. 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 generic 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. Function definitions are semantically just assignments of anonymous, single argument functions to variable names. Functions can be redefined at any point and within any control flow structure or scope. μPython supports higher order functions and hence functions are first class citizens.

Bytecode. Our type analysis is defined on the μPython bytecode. For presentation purposes we use a simplified machine model consisting of a store (for mapping variables to constants), an integer-valued program counter and a single accumulator acc rather than a full evaluation stack with an accumulator. Note that the full Python VM is a stack-based machine and our implementation of preemptive type checking fully supports evaluation stacks. We use the metavariables u, v to range over names including acc. 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

Statements:

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

Expressions:

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

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 4: High-level syntax of the μPython language

$$\begin{array}{rclllcl}
instr & ::= & \mathsf{LC}\ c & \text{(load constant)} & \quad & \mid & \mathsf{intOp} \\
& \mid & \mathsf{LG}\ x & \text{(load global)} & & \mid & \mathsf{strOp} \\
& \mid & \mathsf{SG}\ x & \text{(store global)} & & \mid & \mathsf{isInst}\ \tau \\
& \mid & \mathsf{JP}\ n & \text{(unconditional jump)} & & \mid & \mathsf{raise} \\
& \mid & \mathsf{JIF}\ n & \text{(jump if false)} & & & \\
& \mid & \mathsf{CF}\ f & \text{(call function)} & & & \\
& \mid & \mathsf{RET} & \text{(return from call)} & & &
\end{array}$$

Figure 5: The μPython bytecodes

$$
\begin{array}{rcll}
\langle \emptyset, \varepsilon \rangle & \to & \langle \Sigma_I, \langle M, 0 \rangle :: \varepsilon \rangle & \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \to & \mathsf{End} & \text{if } P_{pc} = \mathsf{RET}, S = \varepsilon \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \to & \langle \Sigma, S \rangle & \text{if } P_{pc} = \mathsf{RET}, S \neq \varepsilon \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \to & \langle \Sigma \oplus (acc \mapsto c), \langle P, pc + 1 \rangle :: S \rangle & \text{if } P_{pc} = \mathsf{LC}\ c \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \to & \langle \Sigma \oplus (acc \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 & \to & & \text{if } P_{pc} = \mathsf{SG}\ x \\
\multicolumn{3}{c}{\langle \Sigma \oplus (x \mapsto \Sigma(acc)) \oplus (acc \mapsto \mathsf{U}), \langle P, pc + 1 \rangle :: S \rangle} & \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \to & \langle \Sigma, \langle P, pc' \rangle :: S \rangle & \text{if } P_{pc} = \mathsf{JP}\ pc' \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \to & \langle \Sigma \oplus (acc \mapsto \mathsf{U}), \langle P, n \rangle :: S \rangle & \text{if } P_{pc} = \mathsf{JIF}\ n, \Sigma(acc) = \mathsf{false} \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \to & \langle \Sigma \oplus (acc \mapsto \mathsf{U}), \langle P, pc + 1 \rangle :: S \rangle & \text{if } P_{pc} = \mathsf{JIF}\ n, \Sigma(acc) = \mathsf{true} \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \to & \mathsf{TypeError} & \text{if } P_{pc} = \mathsf{JIF}\ n, \neg\Sigma(acc) : \mathsf{Bool} \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \to & \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 & \to & \mathsf{TypeError} & \text{if } P_{pc} = \mathsf{CF}\ f, \neg\Sigma(f) : \mathsf{Fn} \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \to & \langle \Sigma \oplus (acc \mapsto \mathsf{U}), \langle P, pc + 1 \rangle :: S \rangle & \text{if } P_{pc} = \mathsf{intOp}, \Sigma(acc) : \mathsf{Int} \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \to & \langle \Sigma \oplus (acc \mapsto \mathsf{U}), \langle P, pc + 1 \rangle :: S \rangle & \text{if } P_{pc} = \mathsf{strOp}, \Sigma(acc) : \mathsf{Str} \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \to & \mathsf{TypeError} & \text{if } P_{pc} = \mathsf{intOp}, \neg\Sigma(acc) : \mathsf{Int} \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \to & \mathsf{TypeError} & \text{if } P_{pc} = \mathsf{strOp}, \neg\Sigma(acc) : \mathsf{Str} \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \to & \langle \Sigma \oplus (acc \mapsto \mathsf{true}), \langle P, pc + 1 \rangle :: S \rangle & \text{if } P_{pc} = \mathsf{isInst}\ \tau, \Sigma(acc) : \tau \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \to & \langle \Sigma \oplus (acc \mapsto \mathsf{false}), \langle P, pc + 1 \rangle :: S \rangle & \text{if } P_{pc} = \mathsf{isInst}\ \tau, \neg\Sigma(acc) : \tau \\
\langle \Sigma, \langle P, pc \rangle :: S \rangle & \to & \mathsf{Exception} & \text{if } P_{pc} = \mathsf{raise}
\end{array}
$$

Figure 6: Semantics of the μPython Bytecode

set of constants to now include constants of type Fn made of finite sequences of bytecode instructions. For technical convenience we also add a constant U of type Un.

The actual bytecodes we use are given in Figure 5. We assume well-formed bytecode where jumps only refer to actual program locations and every program has a RET instruction at its final location. Loading places values in the accumulator, while storing moves a value from the accumulator to a variable. The load instructions LC c and LG x load a constant c resp. a global variable x onto the top of the stack; conversely, the store instruction SG x stores the accumulator in the global variable x. There are four instructions that change the program counter. The jump instructions JP n and JIF n jump unconditionally and conditionally (i.e., if the top of the stack contains the value false) to a given location n. The instructions CF f and RET implement a function call and return mechanism. The instructions intOp, strOp, and raise echo the corresponding high-level expressions and isInst writes a Boolean into the accumulator depending on whether this contains a value of the given type. Note that most operations consume the accumulator value as part of their execution. The CF instruction is of interest: to execute this the machine finds the sequence of instructions P' mapped to f in the store and pushes this program on to the call stack, with the program counter at zero.

Reduction Semantics. We formalise μPython's semantics by the rules for single execution steps of the abstract machine shown in Figure 6. The states of the machine, $\mathit{State}^{\to}$, are of the form $\langle \Sigma, S \rangle$ (where the *environment* Σ is a mapping from names, including acc, to constants and S is a *call stack* of $\langle$program, program counter$\rangle$ pairs) or one of the termination states TypeError, Exception, or End. We assume that the machine begins in an "empty" state $\langle \emptyset, \varepsilon \rangle$. The only step applicable at this point loads $\langle M, 0 \rangle :: \varepsilon$ onto the call stack, where M is the main program. This step also sets the store to Σ_I, an initial store that contains mappings for built-ins and that maps all other names to U. We write 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 primitive type τ.

3. Type inference for μPython

In imperative dynamically typed languages, variables can be reassigned with values of different types at runtime, and used differently under different conditions too. With preemptive type checking, to determine whether a type error may occur we need to establish, for any given point of execution, two pieces of information: the type a variable actually has, which is determined by the values it can hold at the location, and the type with which a variable may be used in the future, which is determined by the operations along the possible control flows from the location. We call these the *present* and *future use* types. Flow-sensitive analyses such as Andersen-style analysis [7] or abstract interpretation [8] are typically used to model the former. However, modeling the latter in such styles is less natural, so we use a type inference approach in this work to model both types in a unified way. To establish the present types, 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 the analysis infers an over-approximation. 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 extend the grammar of types to be

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

and define the subtyping order $<:$ inductively

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

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

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. At any given program execution point we will check that the present and future use types are compatible, by which we simply mean that the present type is a subtype of the future use type. If this is no longer the case then we have detected a type error.

3.1. Program execution points

Our type analysis establishes the type of any variable at any program execution point. However, the variables in the outer scope of a function can have different types for different invocations of that function. Therefore, the entire call stack and not simply the code location 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. Unfortunately, even for finite programs, the CFG of all possible program execution points could be infinite. We therefore over-approximate the CFG by *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 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. Given a truncated execution point s we write $\text{prev}(s)$ for the set of nodes from which s can be reached in the truncated CFG of the program. Similarly, $\text{next}(s)$ denotes the set of nodes which can be reached from s. The μPython interpreter 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 $\text{prev}(s)$ and $\text{next}(s)$ functions in order to find the present and future use types of variables. Both functions return

$$\begin{tikzcd}
\langle \Sigma_0, S_0 \rangle \arrow[r, "\rightarrow"] \arrow[d, "\lfloor . \rfloor"'] & \langle \Sigma'_0, S'_0 \rangle \arrow[d, "\lfloor . \rfloor"] \\
s \arrow[r, bend left=10] \arrow[r, shift left=1, "\mathrm{next}(.)" description] \arrow[r, shift right=1] \arrow[r, bend right=10] & \left\{ \begin{matrix} s'_0 \\ s'_1 \end{matrix} \right\} \\
\langle \Sigma_1, S_1 \rangle \arrow[r, "\rightarrow"'] \arrow[u, "\lfloor . \rfloor"] & \langle \Sigma'_1, S'_1 \rangle \arrow[u, "\lfloor . \rfloor"']
\end{tikzcd}$$

Figure 7: An illustration of the correspondence between stacks and execution points, where $\{s'_0, s'_1\} \subseteq \mathrm{next}(s)$.

finite sets of all possible previous and next execution points, respectively, of a given execution point. Hence, for any state of a running program, if $\langle \Sigma, S \rangle \rightarrow \langle \Sigma', S' \rangle$ then

$$s \in \mathrm{prev}(s') \text{ and } s' \in \mathrm{next}(s) \tag{1}$$

It should be noted that the converse of (1) does not hold in the sense that if $s \in \mathrm{prev}\, s'$ (or similarly for next) then it is not necessarily the case that $\langle \Sigma, S \rangle \rightarrow \langle \Sigma', S' \rangle$ for some Σ. It therefore follows that our analysis, based on the truncated CFG, is an over-approximation as the truncated flow graph may contain transitions that are not witnessed by an underlying transition in the CFG proper. As we see later, this over-approximation does not affect soundness of our analysis adversely but rather it simply reduces the precision of the inferred types.

Figure 7 illustrates the relationship between execution points and stacks. Here, the 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 at least to contain $\{s'_0, s'_1\}$.

The simplest way to truncate the call stack is to retain only the last element, i.e., 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.

3.2. Trails

Since the types associated with variables depend on the program execution points, 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 program execution point.

For illustration, we initially consider type judgements to be 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 the execution point s. We could then consider the following rule:

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

This rule states that if the instruction executed at s is any one of LC, JIF, or 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. Unfortunately, this also means that the proof tree becomes a potentially infinite structure, and the simplistic mechanism considered so far will not work in practice. We show this by applying this rule on a small program M. In M, we load a random 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:

$$\begin{tikzcd}
\varepsilon \arrow[r] & \langle M,0\rangle \arrow[r] & \langle M,1\rangle \arrow[r] \arrow[l, bend left] & \langle M,2\rangle
\end{tikzcd}$$

We now attempt to infer the present type of x, where x is not defined in M, after executing the instruction at execution point $\langle M, 2\rangle$. 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:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\varepsilon \vdash x : \mathsf{Un}}\,\textsf{INIT} \qquad \dfrac{\dots}{\langle M,1\rangle \vdash x : \mathsf{Un} \sqcup \dots}\,\textsf{PREV}}{\langle M,0\rangle \vdash x : \mathsf{Un} \sqcup \dots}\,\textsf{PREV}}{\langle M,1\rangle \vdash x : \mathsf{Un} \sqcup \dots}\,\textsf{PREV}}{\langle M,2\rangle \vdash x : \mathsf{Un} \sqcup \dots}\,\textsf{PREV}$$

However, the structure of the tree will repeat itself due to the cycle between $\langle M, 1\rangle$ and $\langle M, 0\rangle$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\varepsilon \vdash x : \mathsf{Un}}\,\textsf{INIT} \qquad \dfrac{\dfrac{\dfrac{}{\varepsilon \vdash x : \mathsf{Un}}\,\textsf{INIT} \qquad \dfrac{\dots}{\langle M,1\rangle \vdash x : \mathsf{Un} \sqcup \dots}\,\textsf{PREV}}{\langle M,0\rangle \vdash x : \mathsf{Un} \sqcup \dots}\,\textsf{PREV}}{\langle M,1\rangle \vdash x : \mathsf{Un} \sqcup \dots}\,\textsf{PREV}}{\langle M,0\rangle \vdash x : \mathsf{Un} \sqcup \dots}\,\textsf{PREV}}{\langle M,1\rangle \vdash x : \mathsf{Un} \sqcup \dots}\,\textsf{PREV}}{\langle M,2\rangle \vdash x : \mathsf{Un} \sqcup \dots}\,\textsf{PREV}$$

In order to address this problem, we introduce a mechanism called *trails*, in the type judgements for both present and future use types. We therefore define the type inference mechanism 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 program execution points (together with the variables that triggered the visit) and are used to ensure termination of the inference, by adding a side condition to the inference rules that the trail cannot contain the current truncated program execution point and variable.

The judgement $\langle s, \mathcal{T}_\emptyset\rangle \vdash_p u : \tau$ (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.

3.3. Type inference rules

We now define the type inference rules with trails for both present and future use types. These rules are given in Figures 8–11.

The axioms for inferring $\vdash_p$ (cf. Figure 8) account for situations in which the present type of a variable is fully determined by the current instruction. This is usually the case when the instruction leaves a value with a specific known type in the variable. For example, after loading a constant (i.e., rule pLC) the accumulator

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

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

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

Figure 8: Inference rules for the $\vdash_p$ judgement (axioms). All rules assume that $s = \langle P, pc \rangle :: ....$

$$\frac{\langle s, acc \rangle \notin \mathcal{T} \quad P_{pc} = \mathsf{LG}\ x \quad \langle s_i, \mathcal{T} \cup \{\langle s, acc \rangle\} \rangle \vdash_p x : \tau_i}{\langle s, \mathcal{T} \rangle \vdash_p acc : \bigsqcup \tau_i}\ \mathsf{pLG}$$

$$\frac{\langle s, x \rangle \notin \mathcal{T} \quad P_{pc} = \mathsf{SG}\ x \quad \langle s_i, \mathcal{T} \cup \{\langle s, x \rangle\} \rangle \vdash_p acc : \tau_i}{\langle s, \mathcal{T} \rangle \vdash_p x : \bigsqcup \tau_i}\ \mathsf{pSG}$$

$$\frac{\langle s, u \rangle \notin \mathcal{T} \quad \langle s_i, \mathcal{T} \cup \{\langle s, u \rangle\} \rangle \vdash_p u : \tau_i}{\langle s, \mathcal{T} \rangle \vdash_p u : \bigsqcup \tau_i}\ \mathsf{pDEFAULT}$$

Figure 9: Inference rules for the $\vdash_p$ judgement. pDEFAULT applies if none of the other rules apply. All rules assume that $s = \langle P, pc \rangle :: ...$ and s_i ranges across prev(s).

is known to have the type of the constant that has just been loaded. Similarly, rule pINST details that after an instance check has been executed the accumulator is known to have a Boolean value loaded. Both rules pTRAIL and pRAISE infer $u : \bot$, but the two judgements are used differently in the type inference. In the case of pTRAIL, it will be joined with another judgement, because the side condition $\langle s, u \rangle \in \mathcal{T}$ restricts its use to loops; hence, $u : \bot$ reflects the fact that no further information can be gleaned by following the loop a second time around. The judgement pRAISE is only applied for an instruction that stops the program, hence no information flows through that execution point. The rule pUSE covers the multiple cases of bytecode instructions that consume the accumulator value and replace it with the uninitialised value U, the type of which is of course Un.

The inference rules in Figure 9 handle the remaining cases. It is perhaps easiest to understand the rule pDEFAULT. This rule applies in the case where no other rule applies and is intended primarily for the situation in which the next instruction does not affect the type of the variable of interest. In particular, to determine the present type of a variable x or the accumulator acc where this is not affected by the next instruction, provided we have not already seen this truncated point and variable combination before, we simply consider the types of the same variable in each of the previous truncated execution points (given by the prev function) and form the union of these types. Of course, we also update the trail information as we do so.

Where the type of variable of interest does depend on the next instruction we have specific rules to capture these cases. Firstly, rule pLG considers the case in which the next instruction is a load command for the variable x and we are interested in the type of acc. In this case, clearly the type of the accumulator after executing this instruction will depend on the type of x. Of course, the type of x depends on the particular path of control taken to arrive at this point so we consider the various types of x in all previous execution points and take the union across these. Complementary to the previous rule, pSG deals with the case where the next instruction is SG x and we are considering the type of x. Clearly the type of x depends on the type

$$\frac{}{\langle \varepsilon, \mathcal{T} \rangle \vdash_f u : \top}\ \mathsf{fINIT} \qquad \frac{P_{pc} = \mathsf{RET}}{\langle \langle P, pc \rangle :: \varepsilon, \mathcal{T} \rangle \vdash_f u : \top}\ \mathsf{fEND} \qquad \frac{\langle s, u \rangle \in \mathcal{T}}{\langle s, \mathcal{T} \rangle \vdash_f u : \bot}\ \mathsf{fTRAIL}$$

$$\frac{P_{pc} = \mathsf{SG}\ x \quad \langle s, x \rangle \notin \mathcal{T}}{\langle s, \mathcal{T} \rangle \vdash_f x : \top}\ \mathsf{fSG1} \qquad \frac{P_{pc} = \mathsf{raise} \quad \langle s, u \rangle \notin \mathcal{T}}{\langle s, \mathcal{T} \rangle \vdash_f u : \top}\ \mathsf{fRAISE}$$

$$\frac{P_{pc} \in \{\mathsf{LC}\ c, \mathsf{LG}\ x, \mathsf{isInst}\ \tau\} \quad \langle s, acc \rangle \notin \mathcal{T}}{\langle s, \mathcal{T} \rangle \vdash_f acc : \top}\ \mathsf{fSET} \qquad \frac{P_{pc} = \mathsf{JIF}\ pc' \quad \langle s, acc \rangle \notin \mathcal{T}}{\langle s, \mathcal{T} \rangle \vdash_f acc : \mathsf{Bool}}\ \mathsf{fJIF}$$

$$\frac{P_{pc} = \mathsf{strOp} \quad \langle s, acc \rangle \notin \mathcal{T}}{\langle s, \mathcal{T} \rangle \vdash_f acc : \mathsf{Str}}\ \mathsf{fSTR} \qquad \frac{P_{pc} = \mathsf{intOp} \quad \langle s, acc \rangle \notin \mathcal{T}}{\langle s, \mathcal{T} \rangle \vdash_f acc : \mathsf{Int}}\ \mathsf{fINT}$$

Figure 10: Inference rules for the $\vdash_f$ judgement (axioms). All rules assume that $s = \langle P, pc \rangle :: ....$

of *acc* at the previous truncated execution points and as above we take the union across the types of *acc* at all previous execution points.

For the rules for $\vdash_f$ we have the axioms in Figure 10 and inference rules in Figure 11. Many of the axioms assign a future use type of $\top$ to a variable. This reflects the fact that there are no constraints on the type of the variable coming from the future program execution, for example cases where that variable is just about to be overwritten are represented by rules fSET and fSG1, cases where the programming is terminating are represented by fEND and fRAISE and the case where no constraint is present is represented by fINIT. The rule fTRAIL plays the same role as in the system for present types. Otherwise, constraints are generated in the type by immediate use of a variable. These immediate uses comprise the conditional jumps and the explicit use operations (cf. rules fJIF, fSTR, and fINT).

Turning attention now to Figure 11 we have a number of cases to consider: again, we have a straightforward "catch all" rule in fDEFAULT that applies only when no other rule applies. This rule simply looks at the future types of the variable of interest at the next execution points (using function next) and forms the union of these. Therefore the future use type records possible future uses along *some* path. More intererstingly, rule fLG concerns the situation where the next instruction is a load command for the variable x and we are interested in the future use type for x. In this case, since we are loading x into the accumulator, any constraints on the future type of the accumulator must be transferred to x; in addition, x remains intact so there may be future uses of it to account for as well. Hence, x's type must reflect multiple constraints and we therefore define a *meet operation* on types, written as $\dot\sqcap$. Unlike $\sqcup$, which yields actual (union) types, the meet operation does not yield actual (intersection) types, but rather is defined via the following elimination rules (applied left-to-right in top-down order):

$$\tau \dot\sqcap (\tau_1 \sqcup \tau_2) = (\tau \dot\sqcap \tau_1) \sqcup (\tau \dot\sqcap \tau_2)$$
$$(\tau_1 \sqcup \tau_2) \dot\sqcap \tau = (\tau_1 \dot\sqcap \tau) \sqcup (\tau_2 \dot\sqcap \tau)$$
$$\tau \dot\sqcap \top = \tau \quad \tau \dot\sqcap \tau = \tau \quad \top \dot\sqcap \tau = \tau$$
$$\tau_1 \dot\sqcap \tau_2 = \bot$$

For instance, the last rule is applied if all the previous rules do not apply. We make a similar use of this meet operation in the rule fCF, where we consider the case where the next instruction is a function call and the variable of interest is the function being called. We can track the constraints on the type at the next execution points in a standard way but of course in order for the instruction to have succeeded we must augment these constraints with the added constraint that the variable is in fact a function. The rule fSG2 is similar to fLG, except of course in this case the accumulator value is consumed, so further uses, which could can affect the future type, are impossible and no meet operation is required. Finally, rule fRET is very

$$\frac{\langle s,x\rangle \notin \mathcal{T} \quad P_{pc} = \mathsf{LG}\ x \quad \langle s_i, \mathcal{T} \cup \{\langle s,x\rangle\}\rangle \vdash_f acc : \upsilon_i \quad \langle s_i, \mathcal{T} \cup \{\langle s,x\rangle\}\rangle \vdash_f x : \nu_i}{\langle s,\mathcal{T}\rangle \vdash_f x : \bigsqcup(\upsilon_i \dot{\sqcap} \nu_i)}\ \mathsf{fLG}$$

$$\frac{s = \langle P,pc\rangle :: \langle P',n\rangle :: ... \quad \langle s,u\rangle \notin \mathcal{T} \quad P_{pc} = \mathsf{RET} \quad \langle s_i, \mathcal{T} \cup \{\langle s,u\rangle\}\rangle \vdash_f u : \tau_i}{\langle s,\mathcal{T}\rangle \vdash_f u : \bigsqcup \tau_i}\ \mathsf{fRET}$$

$$\frac{\langle s,acc\rangle \notin \mathcal{T} \quad P_{pc} = \mathsf{SG}\ x \quad \langle s_i, \mathcal{T} \cup \{\langle s,acc\rangle\}\rangle \vdash_f x : \tau_i}{\langle s,\mathcal{T}\rangle \vdash_f acc : \bigsqcup \tau_i}\ \mathsf{fSG2}$$

$$\frac{\langle s,f\rangle \notin \mathcal{T} \quad P_{pc} = \mathsf{CF}\ f \quad \langle s_i, \mathcal{T} \cup \{\langle s,f\rangle\}\rangle \vdash_f f : \tau_i}{\langle s,\mathcal{T}\rangle \vdash_f f : \bigsqcup(\tau_i \dot{\sqcap} \mathsf{Fn})}\ \mathsf{fCF}$$

$$\frac{\langle s_i, \mathcal{T} \cup \{\langle s,u\rangle\}\rangle \vdash_f u : \tau_i}{\langle s,\mathcal{T}\rangle \vdash_f u : \bigsqcup \tau_i}\ \mathsf{fDEFAULT}$$

Figure 11: Inference rules for the $\vdash_f$ judgement. fDEFAULT applies if none of the previous rules apply. All rules assume that $s = \langle P, pc\rangle :: ....$

similar to the default rule except for the side-condition that the (truncated) call stack must contain at least two elements, otherwise rule fEND would apply.

Note that the trail sets $\mathcal{T}$ are finitely bounded because the call stacks are truncated to a fixed depth and because there are, for a given program, only finitely many code locations and variables. For a given program, we write $\mathcal{T}_U$ to denote the maximum trail containing all truncated execution point/variable pairs. In fact, because the trail sizes strictly decrease in non-leaf rules, because all rules have finitely many hypotheses, and by König's Lemma, it is guaranteed that the application of the type inference rules terminates and thus, for any s, u, the judgements $\langle s, \mathcal{T}_\emptyset\rangle \vdash_p u : \tau$ and $\langle s, \mathcal{T}_\emptyset\rangle \vdash_f u : \tau'$ hold for some τ, τ'.

3.4. Correctness of Inference Algorithm

We now show that the type inference rules are correct. 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 1. *Consider a derivation $\langle \emptyset, \varepsilon\rangle \xrightarrow{n} \langle \Sigma, S\rangle \to \langle \Sigma', S'\rangle$, where Σ, Σ' are environments and S, S' are call stacks, and the judgement $\langle s, \mathcal{T}_\emptyset\rangle \vdash_p u : \tau_p$, where $s = \lfloor S\rfloor_N, N \in \mathbb{N}^+$, is a finite truncation of the stack S, and assume $\Sigma'(u) : \tau_r$. Then, the inferred type is an over-approximation of the runtime type, i.e.,*

$$\tau_r <: \tau_p \tag{2}$$

Proof. We proceed by induction on n.
Base case. For $n = 0$, we have $\langle \Sigma, S\rangle = \langle \emptyset, \varepsilon\rangle$, and hence $s = \varepsilon$. Since $\langle \emptyset, \varepsilon\rangle$ 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 (2) 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 (2) holds since the subtype operator is reflexive.

Inductive case. We assume that the claim holds for some $n > 0$, i.e., that 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 single step, i.e., $\langle \Sigma', S' \rangle \to \langle \Sigma'', S'' \rangle$. 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

$$\tau'_r <: \tau'_p \tag{3}$$

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

We start by noting that some rules are not applicable. pINIT relies on the call stack being empty, but with $n > 0$ we know that s' is non-empty because there is no step that yields an empty stack. pTRAIL rule is also not applicable because its precondition requires the variable u for execution point s' to be in the trail but since $\mathcal{T}_\emptyset$ is empty we have that $\langle s', u \rangle \notin \mathcal{T}_\emptyset$. Finally, pRAISE is not applicable because we cannot execute another step after reducing to a TypeError.

We continue by analysing the cases that apply to the remaining axioms shown in Figure 8. We only show details for pLC but the remaining cases all follow the same pattern, with Bool and Un taking the place of the type τ.

Case pLC. u is *acc*, s' has the form $\langle P', pc' \rangle :: \ldots$ and $P'_{pc'}$ is LC c. Hence, the inferred type τ'_p is obtained using the judgement $\langle s', \mathcal{T}_\emptyset \rangle \vdash_p acc : \tau'_p$. By pLC, τ'_p is such that $c : \tau'_p$ where 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 *acc*, s' has the form $\langle P', pc' \rangle :: \ldots$ and $P'_{pc'}$ is LC c. From this we see that $\Sigma'(acc)$ 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 (3) holds as required.

We now focus on the recursive rules shown in Figure 9. The proofs for these cases follow a common pattern and we only elaborate on one case.

Case pLG. u is *acc*, s' has the form $\langle P', pc' \rangle :: \ldots$ and $P'_{pc'}$ is LG x. Hence, the inferred type τ'_p is obtained using the judgement $\langle s', \mathcal{T}_\emptyset \rangle \vdash_p acc : \tau'_p$. By pLG, we have $\tau'_p = \bigsqcup \tau_i$ such that for all $s_i \in \text{prev}(s')$,

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

By definition of prev, 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

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

Since τ''_p must be one of the τ_i joined together to yield τ'_p, we know that $\tau''_p <: \tau'_p$ and we can use Lemma 1 (see below) 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 transitivity of the subtype operator we therefore know that $\tau_r <: \tau'_p$.

From the semantics of μPython (cf. Figure 6) for this case we know that $\Sigma(x)$ is $\Sigma'(acc)$. Hence, $\tau_r = \tau'_r$ and therefore we conclude that $\tau'_r <: \tau'_p$ as required in (3). □

The following technical lemma relates the types of a variable which can be inferred with and without using the program point information contained in the trail. Its proof is in the appendix.

Lemma 1 (p-Bounding)**.** *For any variables u, v, truncated execution points s, s', and trails $\mathcal{T}, \mathcal{T}'$ such that $\mathcal{T}' \subseteq \mathcal{T}$. If*

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

then also

$$\tau <: \tau'$$

The next lemma states that with fewer elements in a trail we get a more general type. Its proof is also shown in the appendix.

Lemma 2. *For any variable u, execution point s and trails $\mathcal{T}$, $\mathcal{T}'$ such that $\mathcal{T}' \subseteq \mathcal{T}$, $\tau <: \tau'$ where*

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

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 1. *A relation $R^{\Uparrow}$ on $\langle \Sigma, S \rangle$ is called a diverge-error relation 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 $\langle \Sigma, S \rangle \to$ TypeError.*

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

Definition 2. *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. Imagine an extra reduction rule that raises a type error is added to the system whenever the current state $\langle \Sigma, S \rangle \notin StateComp$. However, we will see in the next section that *StateComp* can be refined to make better use of static type information.

Theorem 2. *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 relation, i.e., if $\langle \Sigma, S \rangle \notin StateComp$ then either

- $\langle \Sigma, S \rangle \to$ TypeError or
- $\langle \Sigma, S \rangle \to \langle \Sigma', S' \rangle$ 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., $\Sigma(u) : \tau_r$ and $\langle s, \mathcal{T}_\emptyset \rangle \vdash_f u : \tau_f$ but $\tau_r \not<: \tau_f$. We choose this u and consider the last rule used to infer the $\vdash_f$ judgement.

We can first rule out the case that the last applied rule was fTRAIL as it only applies to a non-empty trail but we have an empty trail here. We can further rule out the cases fSET, fEND, fRAISE, and fSG1 because their application would imply $\tau_f = \top$, and thus yield an immediate contradiction, since $\tau_r <: \top$ holds for all τ_r.

The remaining axioms, fJIF, fSTR, and fINT follow the same pattern, so we will use the case matching rule fJIF as an example.

Case fJIF. u is *acc*, s has the form $\langle P, pc \rangle :: ...$ and P_{pc} is JIF n. In this case, $\tau_f =$ Bool must hold therefore and $\tau_r \not<: \tau_f$ means that the type τ_r of the value held in *acc* before the instruction is not a subtype of Bool and hence not Bool. This means that $\neg\Sigma(acc)$: Bool. From the semantics of μPython (see Figure 6), we see that we get a type error if we execute the current instruction (a conditional jump) and acc is not of type Bool, i.e.,

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

as required.

We now proceed to analyse the rules in Figure 11. All cases, except fLG and fCF, which we shall tackle later, follow the same pattern. We therefore elaborate the case for $\mathsf{fDEFAULT}$ and omit the other cases.
Case fDEFAULT. s has the form $\langle P, pc \rangle :: ...$ and no other premise applies. We know from the reduction semantics that a unique $\langle \Sigma', S' \rangle$ state exists such that $\langle \Sigma, S \rangle \to \langle \Sigma', S' \rangle$, so it suffices to show that $\langle \Sigma', S' \rangle \notin \mathit{StateComp}$ 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 $\mathsf{fDEFAULT}$, $\tau_f = \bigsqcup \tau_i$ such that:

$$\langle s_i, \mathcal{T} \cup \{\langle s, u \rangle\} \rangle \vdash_f u : \tau_i$$

From the definition of next (1), we know that at least one of the execution points returned by $\mathrm{next}(s)$ is a truncation of the runtime call stack S'. Hence $s' \in \mathrm{next}(s)$. 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 3 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 6) 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 fLG and fCF are more intricate. These also follow similar patterns, so we will look at the case for fLG.
Case fLG. u is x, s has the form $\langle P, pc \rangle :: ...$ and P_{pc} is $\mathsf{LG}\ x$. 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 fLG, $\tau_f = \bigsqcup \upsilon_i \sqcap \nu_i$ such that:

$$\langle s_i, \mathcal{T} \cup \{\langle s, x \rangle\} \rangle \vdash_f acc : \upsilon_i$$
$$\langle s_i, \mathcal{T} \cup \{\langle s, x \rangle\} \rangle \vdash_f x : \nu_i$$

By definition of next we know that at least one of the execution points returned by $\mathrm{next}(s)$ is a truncation of the runtime call stack S'. Hence $s' \in \mathrm{next}(s)$.

Let υ'' and ν'' be such that

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

Given that $\mathcal{T}_\emptyset \subseteq \mathcal{T}_\emptyset$, we use our result from Lemma 3 and conclude that

$$\upsilon' <: \tau_f \sqcup \upsilon'' \tag{4}$$
$$\nu' <: \tau_f \sqcup \nu'' \tag{5}$$

where υ' and ν' are such that

$$\begin{aligned}\langle s', \mathcal{T}_\emptyset\rangle &\vdash_f acc : \upsilon' \\ \langle s', \mathcal{T}_\emptyset\rangle &\vdash_f x : \nu'\end{aligned}$$

We combine (4) and (5) into

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

which we can rearrange as

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

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

$$(\upsilon' \dot{\sqcap} \nu') <: \tau_f$$

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

$$\begin{aligned}\tau_r &\not<: \upsilon' \\ \tau_r &\not<: \nu'\end{aligned}$$

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

$$\begin{aligned}\Sigma'(acc) &: \tau_r \\ \Sigma'(x) &: \tau_r\end{aligned}$$

Since $\tau_r \not<: \upsilon'$ or $\tau_r \not<: \nu'$, the runtime type of *acc* or the runtime type of x is not a subtype of its future use type. We therefore conclude that $\langle \Sigma', S'\rangle \notin \mathit{StateComp}$ as required. $\square$

The next lemma is the future use types equivalent of Lemma 1. 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 compared to $\mathcal{T}$. Its proof is shown in the appendix.

Lemma 3 (f-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*

$$\begin{aligned}\langle s, \mathcal{T}'\rangle &\vdash_f v : \tau \\ \langle s', \mathcal{T}\rangle &\vdash_f u : \tau' \\ \langle s', \mathcal{T} \cup \{\langle s, v\rangle\}\rangle &\vdash_f u : \tau''\end{aligned} \tag{6}$$

The next lemma is the future use types equivalent of Lemma 2. Since it follows the same pattern as Lemma 2, we omit its proof.

Lemma 4. *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*

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

4. Type checking for μPython

The naive runtime type check *StateComp* in the previous section simply checks whether the current runtime type of a variable is a subtype of the statically inferred f-type. However, we have also statically calculated the p-types as a sound approximation of the runtime types and we can leverage this to obtain a type check that can be partially evaluated statically. This predicate is defined on edges in the truncated CFG.

Definition 3. *The edge compatibility predicate EdgeComp holds at $\langle s, s', \Sigma'\rangle$ if for all variables u, such that*

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

then

$$\tau_p <: \tau'_f \quad \textit{or} \quad \tau'_r <: \tau_p \tilde{\sqcap} \tau'_f$$

In other words, as the program moves from a state s to a state s', there is no error to report if either (1) the statically approximated runtime type is a subtype of future uses, or (2) the actual new runtime type of a variable is within the future use set (modulated by the present type). Clearly, only (2) requires the inspection of the runtime types. Even then, if $\tau_p \tilde{\sqcap} \tau'_f$ is $\perp$, we know statically that the predicate must fail as there are no constants of type $\perp$. 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.

Definition 4. *The fail edge predicate FailEdge holds at $\langle s, s'\rangle$ if $s \in \mathrm{prev}(s')$ and either $\forall \Sigma' \cdot \langle s, s', \Sigma'\rangle \notin$ EdgeComp or $\{\langle s', s''\rangle \mid s'' \in \mathrm{next}(s')\} \subseteq$ FailEdge.*

Fail edges are used to denote pairs of points at which the program will always eventually raise a type error. These are similar to *doomed program points* [3], specific to type errors. 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. Thus τ_p is not a subtype of τ'_f and $\tau_p \tilde{\sqcap} \tau'_f = \perp$. Since there is no primitive type that is a subtype of $\perp$, 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'.

The checked semantics makes direct use of the original semantics. A step on a state $\langle \Sigma, S\rangle$ in the original semantics may reduce to a state $\langle \Sigma', S'\rangle$, i.e., $\langle \Sigma, S\rangle \rightarrow \langle \Sigma', S'\rangle$. In the checked semantics, $\langle \Sigma, S\rangle$ may also reduce to $\langle \Sigma', S'\rangle$. However, $\langle \Sigma, S\rangle$ may also reduce to a preemptive type error exception instead, i.e., $\langle \Sigma, S\rangle \dashrightarrow \mathsf{Exception}$. We note that $\langle \Sigma, S\rangle$ is overloaded, and *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.

Definition 5. *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* $\mathsf{Exception}$ *such that:*

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

Definition 6. *A relation $R^{\leq}$ on $State^{\rightarrow} \times State^{\dashrightarrow}$, which relates only identical non-terminating states (i.e., if $\langle \Sigma, S\rangle R^{\leq} \langle \Sigma_c, S_c\rangle$ then $\Sigma = \Sigma_c$ and $S = S_c$) is called an error-preserving simulation if the following holds for all $\langle \Sigma, S\rangle \in \mathrm{dom}(R^{\leq})$:*

- $\langle \Sigma, S\rangle \not\rightarrow \mathsf{TypeError}$
- *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*
  - $\langle \Sigma, S\rangle \dashrightarrow \langle \Sigma', S'\rangle \wedge \langle \Sigma', S'\rangle R^{\lesssim} \langle \Sigma', S'\rangle$ *or*
  - $\langle \Sigma, S\rangle \dashrightarrow \mathsf{Exception} \wedge \langle \Sigma', S'\rangle \in \Uparrow$

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

We now prove that preemptive type checking is *complete*, i.e., 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 3. *Let* R^{SC} *be defined as*

$$\{\langle \Sigma, S\rangle, \langle \Sigma, S\rangle \mid \langle \emptyset, \varepsilon\rangle \xrightarrow{*} \langle \Sigma, S\rangle \wedge \langle \Sigma, S\rangle \in \mathit{StateComp}\} \tag{7}$$

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

$$\begin{gathered} \Sigma(u) : \tau_r \\ \langle s, \mathcal{T}_\emptyset\rangle \vdash_f u : \tau_f \end{gathered} \tag{8}$$

where $s = \lfloor S \rfloor$.

From the definition of error-preserving simulation in Definition 6, we need to prove that *all* of the following hold:

$$\langle \Sigma, S\rangle \not\rightarrow \mathsf{TypeError} \tag{9}$$

$$\text{if } \langle \Sigma, S\rangle \rightarrow \mathsf{End} \qquad \text{then } \langle \Sigma, S\rangle \dashrightarrow \mathsf{End} \tag{10}$$

$$\text{if } \langle \Sigma, S\rangle \rightarrow \mathsf{Exception} \qquad \text{then } \langle \Sigma, S\rangle \dashrightarrow \mathsf{Exception} \tag{11}$$

We also need to prove that the following hold:

$$\text{if } \langle \Sigma, S\rangle \rightarrow \langle \Sigma', S'\rangle \text{ then } \langle \Sigma, S\rangle \dashrightarrow \langle \Sigma', S'\rangle \wedge \langle \Sigma', S'\rangle R^{SC} \langle \Sigma', S'\rangle \tag{12}$$

$$\text{or} \quad \langle \Sigma, S\rangle \dashrightarrow \mathsf{Exception} \wedge \langle \Sigma, S\rangle \in \Uparrow \tag{13}$$

By definition of the checked semantics, if $\langle \Sigma, S\rangle \rightarrow$ End then $\langle \Sigma, S\rangle \dashrightarrow$ End. Therefore we have shown that (10) holds as required. The same is true for Exception, i.e., (11).

We now proceed to prove that a type error cannot be raised, i.e., (9). We prove this by contradiction, i.e., we assume $\langle \Sigma, S\rangle \rightarrow$ TypeError holds and find a contradiction. We analyse all cases of the μPython semantics where $\langle \Sigma, S\rangle \rightarrow$ TypeError.

Case fJIF. u is acc, s has the form $\langle P, pc\rangle :: ...$ and P_{pc} is JIF n and $\neg(\Sigma(acc) : \mathsf{Bool})$. From (8), 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 (8):

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

This contradicts the assumption of the current case.

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

We now consider cases where $\langle \Sigma, S\rangle \rightarrow \langle \Sigma', S'\rangle$. Since we know that $\langle \emptyset, \varepsilon\rangle \xrightarrow{*} \langle \Sigma, S\rangle$, we can conclude that $\langle \emptyset, \varepsilon\rangle \xrightarrow{*} \langle \Sigma', S'\rangle$. We also need to show that either (12) *or* (13) holds. We proceed by case analysis on $\dashrightarrow$ for the cases where $\langle \Sigma, S\rangle \rightarrow \langle \Sigma', S'\rangle$.

Case 2. *$\langle s, s', \Sigma' \rangle \notin$ EdgeComp. From the definition of our checked μPython semantics in Definition 5 for this case, we can conclude that*

$$\langle \Sigma, S \rangle \dashrightarrow \mathsf{Exception} \tag{14}$$

By analysing the definition of EdgeComp, i.e. Definition 3, the current case implies that there is a u such that $\tau'_r \not<: \tau_p \sqcap \tau'_f$, where

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

Now since we know from Theorem 1 that $\tau'_r <: \tau_p$, we can say that there is a u such that $\tau'_r \not<: \tau'_f$. This means that $\langle \Sigma', S' \rangle \notin$ StateComp (see Definition 2).

Hence we know from Theorem 2 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 (14), we have shown that (13) holds as required.

Case 3. *$\langle s, s' \rangle \in$ 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 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 $\langle s, s' \rangle \in$ FailEdge, so either $\langle s, s', \Sigma' \rangle \notin$ EdgeComp and hence $\langle \Sigma, S \rangle \in \Uparrow$, or $\langle s', s'' \rangle \in$ FailEdge for all $s'' \in \mathrm{next}(s')$, as required.

Case 4. *$\langle s, s', \Sigma' \rangle \in$ EdgeComp. From Definition 5 of our checked μPython semantics, we can conclude that*

$$\langle \Sigma, S \rangle \dashrightarrow \langle \Sigma', S' \rangle \tag{15}$$

In order to prove that (12) holds, we need to show that:

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

holds, that is,

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

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

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

we have

$$\tau'_r <: \tau'_f \tag{16}$$

In order to prove (16), we refer to the definition of EdgeComp, which states that

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

where

$$\begin{gathered} \langle s', \mathcal{T}_\emptyset \rangle \vdash_f u : \tau'_f \\ \langle s, \mathcal{T}_\emptyset \rangle \vdash_p u : \tau_p \\ \Sigma'(u) : \tau'_r \end{gathered} \tag{17}$$

Since Theorem 1 guarantees $\tau'_r <: \tau_p$, combining with the above we conclude that $\tau'_r <: \tau'_f$, as required. □

Corollary 1. *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 about the properties of preemptive type checking. What this signifies is that any terminating program that is run using the checked semantics can never raise a type error, but reduces to End or Exception.

Corollary 2. *Consider a maximal trace* $\langle \emptyset, \varepsilon\rangle \overset{*}{\dashrightarrow} N \not\dashrightarrow$. *Then* N *is either* End *or* Exception.

Proof. We note immediately that $\langle \emptyset, \varepsilon\rangle \in \mathit{StateComp}$ holds by virtue of rule fINIT of Figure 10. Therefore we have $\langle \emptyset, \varepsilon\rangle\, R^{SC} \langle \emptyset, \varepsilon\rangle$ and hence by the above corollary we have $\langle \emptyset, \varepsilon\rangle \lesssim \langle \emptyset, \varepsilon\rangle$. 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. □

4.1. Optimality

Now that we have shown the correctness of our type inferencer, we would like to establish that our type inference system is optimal in the sense that the checked semantics report an Exception as soon as the control flow reaches a point where all possible further execution steps in the unchecked semantics lead to a TypeError state. However, since our analysis considers variables individually, we can only prove that our inference system satisfies a milder form of optimality in general, along execution sequences in which there are no branches of control flow. 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.

Definition 7. *A reduction step* $\langle \Sigma, S\rangle \to \langle \Sigma', S'\rangle$ *is called linear if* $\mathrm{next}(s) = \{s'\}$. *A sequence* $\langle \Sigma, S\rangle \overset{*}{\to} \langle \Sigma', S'\rangle$ *is called linear if each step in the sequence is linear.*

Theorem 4 (Linear optimality)**.** *Consider a state* $\langle \emptyset, \varepsilon\rangle$ *that is executed a number of times using our checked semantics until it reaches a state* $\langle \Sigma, S\rangle$.

$$\langle \emptyset, \varepsilon\rangle \overset{*}{\dashrightarrow} \langle \Sigma, S\rangle$$

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

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

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

Proof. We prove this by contradiction. We assume that the checked semantics does not find type errors in a linearly optimal manner, and therefore $\langle \Sigma, S\rangle \dashrightarrow \langle \Sigma', S'\rangle$, but $\langle \Sigma', S'\rangle \dashrightarrow$ 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 *FailEdge*, we can conclude that $\langle s, s'\rangle \in \mathit{FailEdge}$. By the checked semantics, in this case $\langle \Sigma, S\rangle \dashrightarrow$ Exception so we have found a contradiction in our hypothesis. From this point onwards we therefore assume that $\langle s', s''\rangle \notin \mathit{FailEdge}$.

Since $\langle \Sigma, S\rangle \dashrightarrow \langle \Sigma', S'\rangle$ and $\langle \Sigma', S'\rangle \dashrightarrow$ Exception, from the definition of the checked μPython semantics in Definition 5, together with our previous assumption, we know:

$$\langle s', s''\rangle \notin \mathit{FailEdge} \tag{18}$$
$$\langle s', s'', \Sigma''\rangle \notin \mathit{EdgeComp} \tag{19}$$

where $\langle \Sigma', S'\rangle \to \langle \Sigma'', S''\rangle$. From the definition of *EdgeComp*, (19) implies that we can pick a u such that:

$$\tau_p' \not<: \tau_f'' \wedge \tau_r'' \not<: (\tau_p' \sqcap \tau_f'') \tag{20}$$

where τ_r', τ_p', τ_r'', and τ_f'' are defined such that $\Sigma'(u) : \tau_r'$, $\langle s', \mathcal{T}_\emptyset\rangle \vdash_p u : \tau_p'$, $\Sigma''(u) : \tau_r''$ and $\langle s'', \mathcal{T}_\emptyset\rangle \vdash_f u : \tau_f''$.

From Theorem 1 we have concluded that $\tau_r'' <: \tau_p'$, so

$$\tau_r'' \not<: \tau_f'' \tag{21}$$

must in fact hold. Also, from the definition of *FailEdge*, (18) 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

$$\tau_p' <: \tau_f'' \vee (\tau_p' \sqcap \tau_f'') \neq \bot$$

Taken with (20), this implies $(\tau_p' \sqcap \tau_f'') \neq \bot$. Collecting the above we have

$$\tau_p' \not<: \tau_f'' \wedge (\tau_p' \sqcap \tau_f'') \neq \bot \tag{22}$$

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 10 and Figure 11).

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 $\mathsf{fSET/JIF/STR/INT}$, except the special case where $P'_{pc'} = \mathsf{LG}\ x$. Under these cases, we can see that rules $\mathsf{pLC/INST/USE}$ 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 (22) 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 2. *fSET and $P'_{pc'} = \mathsf{LG}\ x$, i.e., u is acc and s' has the form $\langle P', pc' \rangle :: ....$*
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 acc), 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 fSET. By this rule and the fact that $\{s''\} = \mathrm{next}(s')$,

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

where υ' is defined such that $\langle s'', \{\langle s', x \rangle\} \rangle \vdash_f acc : \upsilon'$. Hence ${\tau'_f}^x <: \upsilon'$.

In this case (i.e., fLG), u is acc and therefore τ_f'' is defined such that $\langle s'', \mathcal{T}_\emptyset \rangle \vdash_f acc : \tau_f''$. By Lemma 4 we can conclude that $\upsilon' <: \tau_f''$. Therefore, by transitivity we conclude that

$${\tau'_f}^x <: \tau_f'' \tag{23}$$

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 acc where $\Sigma''(acc) : \tau_r''$.
By the μPython semantics, we conclude that ${\tau'_r}^x = \tau_r''$. Therefore, since we know from (21) that $\tau_r'' \not<: \tau_f''$, we can also conclude that ${\tau'_r}^x \not<: \tau_f''$. Also since we know from (23) that ${\tau'_f}^x <: \tau_f''$, we can now conclude that

$${\tau'_r}^x \not<: {\tau'_f}^x \tag{24}$$

From Theorem 3 we know that a state that has been executed several times using the checked semantics maintains an error-preserving simulation. This means that since $\langle \emptyset, \varepsilon \rangle \overset{*}{\dashrightarrow} \langle \Sigma', S' \rangle$, then $\langle \Sigma', S' \rangle \in \mathit{StateComp}$, i.e. ${\tau'_r}^x <: {\tau'_f}^x$. We have therefore found a contradiction with (24), as required. $\square$

By this proof we have shown that preemptive type checking is at least linearly optimal in terms of type error preemption.

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

Figure 12: Algorithm for inserting type checks in μPython programs, expressed as a function *specialise*(P, s) that returns an updated program P'.

4.2. Type check insertions

We now 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 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 12. 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 appendix for reference. These are expanded to a list of bytecode instructions. Procedure *extend*, 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.

```
0   def f():
1       return intOp(x)
2   if * :
3       x = '42'
4   else :
5       x = 42
6   f()
```

```
0   def f():
1       return intOp(x)
2   if * :
        raise
3       x = '42'
4     else :
5       x = 42
6   f()
```

Figure 13: A simple μPython example (left) and transformed using preemptive type checking (right)

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
inst	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
inst	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	
inst	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 14: Control Flow for the μPython example

5. A worked example

We now go through the μPython example in Figure 13 (left), which can raise a TypeError depending on the branch taken at line 4. This compiles to M and P^f, defined as

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

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 14. The analysis shows why the edge $\langle\langle M,4\rangle,\langle M,5\rangle\rangle$ is in *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 diverge. From the definition of *FailEdge*, we need to show that

$$\forall\Sigma'\cdot\langle\langle M,4\rangle,\langle M,5\rangle,\Sigma'\rangle\notin \mathit{StateComp} \tag{25}$$

We have derivations of the following in Figure 15,

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

Since $\mathsf{Int}\neq\top$, $\mathsf{Int}\not<:\mathsf{Str}$, and the fact that there can be no $\tau_r\neq\bot$ such that $\tau_r<:\bot$, we know that (25) holds. Similarly, we also conclude that $\langle\langle M,3\rangle,\langle M,4\rangle\rangle\in \mathit{FailEdge}$.

The edge in (25) 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 M. Since this is the program at the outermost scope, the specialisation argument is ε and $\mathit{specialise}(M,\varepsilon)$ is called. According

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle\langle M,10\rangle, \{\langle\langle P^f,2\rangle :: \langle M,9\rangle, x\rangle, \langle\langle P^f,1\rangle :: \langle M,9\rangle, x\rangle, ...\}\rangle \vdash_f x : \top}\,\textsf{fEND}}{\langle\langle P^f,2\rangle :: \langle M,9\rangle, \{\langle\langle P^f,1\rangle :: \langle M,9\rangle, x\rangle, \langle\langle P^f,0\rangle :: \langle M,9\rangle, x\rangle, ...\}\rangle \vdash_f x : \top}\,\textsf{fRET}}{\langle\langle P^f,1\rangle :: \langle M,9\rangle, \{\langle\langle P^f,0\rangle :: \langle M,9\rangle, x\rangle, \langle\langle M,9\rangle, x\rangle, ...\}\rangle \vdash_f acc : \mathsf{Int}/x : \top}\,\textsf{fINT/DEFAULT}}{\langle\langle P^f,0\rangle :: \langle M,9\rangle, \{\langle\langle M,9\rangle, x\rangle, \langle\langle M,6\rangle, x\rangle, ...\}\rangle \vdash_f x : \mathsf{Int}}\,\textsf{fLG}}{\langle\langle M,9\rangle, \{\langle\langle M,6\rangle, x\rangle, \langle\langle M,5\rangle, acc\rangle, ...\}\rangle \vdash_f x : \mathsf{Int}}\,\textsf{fCF2}}{\langle\langle M,6\rangle, \{\langle\langle M,5\rangle, acc\rangle\}\rangle \vdash_f x : \mathsf{Int}}\,\textsf{fRET/DEFAULT}}{\langle\langle M,5\rangle, \mathcal{T}_\emptyset\rangle \vdash_f acc : \mathsf{Int}}\,\textsf{fSG2}$$

Figure 15: Derivations of present and future use types at $\langle M, 4\rangle$ and $\langle M, 5\rangle$. For simplicity, the side-conditions are not shown in the rules. The rules are applied to the location at the top of the call stack.

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:

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

where the inserted code is underlined and n is the length of the instructions in $\underline{\text{failIfTrue}}$. This is equivalent to the high-level program shown in Figure 13 (right). 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 [9]. 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 [4, 10, 11]. Other static analysis approaches for dynamic languages would reject this program outright [12, 13, 14]. A gradual typing approach would not preempt the type error.

6. Implementation

We implemented the preemptive type checking tool[2] 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. 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 only once.

We have based our implementation on CPython 3.3 and we support a number of features, including lexical scoping and and global variables, the evaluation stack, control structures such as while-loops, polyadic functions, anonymous functions, simple data structures, and operators without overloading. In total, we support 40 out of around 90 bytecodes in the Python language. Most of the rest of the bytecodes should only involve engineering effort to support. Bytecode instructions that can perform metaprogramming are harder to support (we mostly support the `MAKE_FUNCTION` bytecode from these). A full "soundy" [15] implementation is within reach given enough engineering resources. We have also manually annotated some

[2]http://github.com/nevillegrech/preemptive-type-checking

```
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 16: Phases of the type checking process, outlined in the user code.

primitive standard library functions with type information and provided the user with the ability to explicitly add this information to any functions using function annotations [16].

Our type checker tool does not require the program's source code. Instead, it works directly on a live program and environment, introspecting and analysing the environment for the currently loaded program. Our type checking mechanism is called on a particular entry function, for example `main`, which is created and initialised by the standard interpreter. Before the type checker is called, the full power of the Python language can be used, and not just the features implemented above. We show this in an example in Figure 16, 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. This mechanism, similar to that used in HeapDL [17, 18], counters the unsoundness introduced due to not modelling all Python features in the analysis.

The analyser, which partly implements preemptive type checking, splits the problem into different stages. 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 in the implementation of the control flow analysis. There are several algorithms that could be used, including ones based on k-CFA [19] with call-site sensitivity. In our case, every edge in the control flow graph is a pair of execution points. 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.

Once control flow analysis has finished, type inference takes place, 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. Emitting bytecode (as in line 13) 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 emitting the bytecode, the type checking tool copies the bytecode in the original function that is being type checked or any function called from within and interleaves the type checks.

7. Evaluation

We tested our implementation on a number of Python benchmarks from the Computer Language Benchmarks Game [20], together with other examples. In order to run the benchmarks we had to manually provide type information for external functions such as `cout`. Some benchmarks in this suite have been ported from original code in statically typed languages and therefore type errors should be rare. The results of our evaluation are tabulated in Figure 17. In this table, *execution point length* refers to the number of elements in an execution point s. The column *dynamic checks* corresponds to the number of type checks that are present in the program's CFG, for the corresponding execution point length.

In these results, we have also obtained statistics about the underlying type analysis - in particular the number of types per variable in the program. We can see that the variables are *polymorphic*. Some of this effect is due to the nature of the analysis of dynamically typed program. For instance, variables that are unassigned have a type `Un`, and many variables also exist as unassigned in some scope. In addition, we can also see that as we increase the execution point length, the precision in the underlying analysis increases too.

One of the benchmarks that we analysed is `mandelbrot`, which plots the Mandelbrot set on a bitmap. This raises a type error when this is run with certain parameters due to a tuple of bytes being used instead of a byte string by function `cout`. With our tool, failure assertions are inserted at two different points, which preempt the type error. Warnings are also statically displayed, which indicate the type errors. 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 47, in main
Expected bytes or bytearray but found tuple

Failure 2 - partial Traceback:
File "mandelbrot-python3-3.py", line 37, in main
Expected bytes or bytearray but found tuple
```

These two failures correspond to the lines `cout((byte_acc,))`. Running the original benchmark in Python without preemptive type checking raises a `TypeError`, with the following output:

```
Traceback (most recent call last):
  File "mandelbrot-python3-3.py", line 47, in <module>
    main()
  File "mandelbrot-python3-3.py", line 37, 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.

An interesting benchmark that we tested is `meteor-contest`, which was ported to Python from a C++ version consisting of approximately 500 lines of code. A number of type checks were inserted, especially since some type information is lost, such as when heterogenous objects are placed into lists and subsequently retrieved. When *running* this benchmark no type errors were encountered, with or without preemptive type checking. A possible failure was however *statically* inferred by our analyser in function `findFreeCell`:

```
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
```

We can see that if no free cells are found in a board, this function will not return anything, so by default this would return `None`. In this case, a type error would occur as `None` cannot be unpacked, like a tuple. The programmer is therefore assuming an invariant that asserts that a “free cell” will always be found in the “board”. The invariant that the loop will terminate without returning is explicitly inserted by our tool. 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.

Preemptive type checking can be successfully scaled to medium sized programs. For example, the benchmark `meteor-contest` with an execution point depth of 4 yields a control flow graph with over 30k nodes. In this case, it took little over 10 minutes to analyse the program and half a second to transform it on an old laptop (Lenovo T440p). The same program however takes 4.5 seconds to analyse and transform when the execution point depth is set to 1. Optimality is still guaranteed in both cases, however more error information can be presented to the user with a larger execution point depth.

Preemptive type checking can be also particularly helpful for less experienced programmers and so we also tested our implementation on code in a question posed by a Python beginner on stackoverflow.com. [3] Our implementation statically produces warnings that corroborate the answer given to this question by Python developers.

[3] http://stackoverflow.com/questions/320827/python-type-error-issue

execution point length	analysis time (ms)	transformation time (ms)	CFG size	dynamic checks	fail edges	inserted checks	types per var
erasefile, 23 lines of code							
1	14	0	38	11	0	1	2.40
2	26	1	55	17	1	0	2.20
3	30	1	61	20	1	0	1.80
pidigits-python3-2, 40 lines of code							
1	96	1	132	62	0	0	2.27
2	97	1	132	62	0	0	2.27
3	95	1	132	62	0	0	2.27
mandelbrot-python3-3, 46 lines of code							
1	113	1	127	55	2	0	2.76
2	110	1	127	55	2	0	2.76
3	113	1	127	55	2	0	2.76
fasta, 96 lines of code							
1	140	1	165	94	0	0	2.62
2	243	3	222	129	0	0	2.62
3	243	4	222	129	0	0	2.62
meteor-contest, 206 lines of code							
1	6673	11	857	395	1	10	2.63
2	17407	18	1719	869	1	35	2.55
3	100686	83	6357	3215	1	179	2.55
4	613965	526	30945	15587	1	1043	2.55

Figure 17: Table of results.

8. Related Work

Combinations of static and dynamic typing have been proposed for most modern programming languages. These enable statically typed code to interact with dynamically typed code. Abadi et al. [10] introduced the type Dyn to model finite disjoint unions or subclassing in object-oriented languages. Since then, type systems that include Dyn have proliferated, and are primarily referred to as gradual type systems.

In gradual typing [21], type consistency $\backsim$ (a reflective, symmetric but non-transitive relation) is used to relate Dyn with static types, and Dyn is statically consistent with any type. Gradual typing has been applied to Python [5] where flow-sensitive type inference is used, but only for local variables. The implementation carries out source-to-source translation at load time [5], and therefore cannot be used unless the source is available. In our implementation, the translation is carried out on bytecode after load time, allowing for code to be evaluated, and some types to be determined prior to this translation, resulting in a more accurate type inference. Sound gradual typing suffers from performance problems in practice [22]. However, by leveraging the underlying VM data structures to tag underlying objects with type information [23], one would instead improve the JIT compilation of the executing gradually typed program. This implementation strategy can be also applied to the preemptive type checking context too. TypeScript [24] is a gradually typed extension of JavaScript that compiles down to JavaScript. Erasure is performed during compilation, thus maximising efficiency and simplicity. However, the type system is unsound primarily because it allows unchecked downcasting and runtime resolving of indexing of properties in objects. Safe TypeScript [25], made safe by adding RTTI (run time type information) at strategic places using differential subtyping. Furthermore, partial erasure prevents dynamic types without RTTI from being coerced. Facebook's Flow [26] also introduces gradual typing to JavaScript, with a type annotation syntax compatible with TypeScript, and uses flow-sensitive type inference. However, the dataflow algorithms employed are heap-insensitive. Gradual typing has also been applied to PHP [27]. Despite their advantages, these type systems do not perform any type error preemption.

As in preemptive type checking, soft typing [4] uses union types to approximate static types in an untyped language and inserts type narrowers to prevent implicit type error exceptions. However, the original work [4] did not handle assignments, so there is no notion of preemption. Soft typing was extended to support Scheme [28] and to handle assignments, but all occurrences of the assigned variable have to have the same type, which makes it impossible to successfully typecheck even the simple example from Figure 2. Soft typing has also been applied to Python [29] and Erlang [30]. In the latter case, the author also bases the type system on a data flow analysis, but does not distinguish between p and f types. Bracha introduces the notion of pluggable type systems [31]. Since preemptive type checking does not affect the semantics of μPython in runtime executions that terminate without raising type errors (Section 3.4) and no type annotations are required, our type system meets this definition.

We now look at JavaScript static type inference mechanisms, which turn dynamically typed languages into statically typed subsets. Many proposals and prototypes for JavaScript exist [32, 33, 34, 35, 36]. These systems are used for speeding up JavaScript implementations or for type checking during development. SJS [32] is a JavaScript type system that infers the structure of objects. It assumes that the object's structure is immutable but also considers subtyping with prototype-based inheritance. Immutability may seem like a subtle property, but in JavaScript, small side effects in functions may change the structure of the object in big ways. These assumptions greatly facilitate the creation of a sound type system that supports subtyping for JavaScript objects, as the distinction between prototype and class based inheritence becomes fuzzy. This technique should also facilitate the inference of object structures in dynamic Python code too. This work is extended [33] in various ways: it now includes recursive and abstract types and is evaluated more rigorously. The addition of abstract types, in particular, allows a larger number of programs to type check. A pragmatic approach adopted in the Firefox JavaScript implementation [34] 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 that are further refined by the JIT. A more similar system to our type system is TAJS [35], which contains an abstract-interpretation based type system. TAJS also incorporates recency abstractions and models many of JavaScript's pecularities in its type system lattice. TeJaS [36] is a type system that is retrofitted to JavaScript. It enforces a subset of JavaScript semantics similar to TypeScript, but includes prototype based subtyping. The type system is locally flow sensitive and globally flow insensitive.

Other static type inference mechanisms to turn languages into statically typed subsets have been proposed. Felleisen and Tobin-Hochstadt [37] propose the notion of *occurrence typing* for implementing a statically typed version of Scheme. A translation of the simple example in Figure 2 is statically rejected by this system. Similarly, statically typed subsets of Python [12] and Ruby [38] have been proposed. These however do not catch all type errors statically, and limit the expressiveness of the language by flagging false positives. Recency types [39] deal with object initialisation patterns in JavaScript, where members are assigned dynamically. The concept of a recency type is similar to the *present types* in our work. Present types are however more sophisticated as these can change throughout intraprocedural paths of control flow rather than blocks.

The use of trails in our type system to implement the coinductive reasoning is inspired by a similar approach used for model checking in the modal μ-calculus to exactly the same effect [40]. Interestingly this technique has also been used in the field of co-logic programming [41] where coinductive type systems for logic programming are introduced and the tracking of previously computed atoms plays the same role as our trails. In practice the use of trails effectively allowed us to turn an abstract semantics into a concrete demand driven analysis. Similarly, logic programming languages have a top down execution strategy and coinductive reasoning allows a concrete implementation of algorithms operating on infinate structures such as proofs or streams [42]. These techniques have been used for type inference of object oriented programs with both data and parametric polymorphism [43, 44]. Specifically, co-logic programming facilitates the support of recursive types and mutually recursive methods within the type inference logic which would be represented as infinate proof trees. Co-logic programming has also been successfully applied to LTL model checking of ActionScript [45].

Lastly, we look at control flow analysis for dynamically typed languages. In particular, k-CFA [46] are a family of algorithms to perform inter-procedural control flow analysis, originally on Scheme by abstract interpretation.

9. Conclusions and Future Work

In this paper, we have described the details of *preemptive type checking*, which is our main contribution. This is a type checking mechanism that acts *preemptively*, raising errors at earlier execution points than in dynamic or gradual typing. Preemptive type checking is guaranteed to not raise any type errors for programs that run to completion under dynamic typing, and will thus not impose restrictions on the programmer's style. We have proved correctness and optimality properties for the theoretical underpinnings of preemptive type checking. Furthermore, we have also demonstrated that this type checking mechanism can be implemented for existing and future languages, for instance as a Python library, which we have evaluated on a small number of real world benchmarks. Preemptive type checking as a mechanism was designed so that it can be bolted on top of existing dynamically-typed (or even gradually-typed) languages, without affecting their expressiveness, thus avoiding adoption barriers.

There are many future avenues of research, particularly to support larger subsets of programming languages. Heap objects and their fields can be modelled using access-path abstractions[47]. Preemptive type checking can be applied to other popular dynamically typed languages. Finally, it should be relatively straightforward to support limited metaprogramming capability while preserving the correctness properties. One way to do so would be by automatically calling the type checking implementation whenever a new part of the running program is generated. The fact that the analysis and transformation mechanism is implemented in the target language, which is imported within a library in the target program, makes this possible.

References

[1] TIOBE Programming Community Index, July 2015, Tech. rep.
URL `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

[2] B. Ray, D. Posnett, V. Filkov, P. T. Devanbu, A large scale study of programming languages and code quality in Github, in: Proceedings of FSE, 2014.

[3] J. Hoenicke, K. R. M. Leino, A. Podelski, M. Schäf, T. Wies, Doomed program points, Formal Methods in System Design 37 (2) (2010) 171–199.

[4] R. Cartwright, M. Fagan, Soft typing, in: Proceedings of PLDI, 1991, pp. 278–292. doi:10.1145/113445.113469.
URL `http://portal.acm.org/citation.cfm?doid=113445.113469`

[5] M. M. Vitousek, A. M. Kent, J. G. Siek, J. Baker, Design and evaluation of gradual typing for Python, in: Proceedings of DLS, 2015.

[6] N. Grech, J. Rathke, B. Fischer, Preemptive type checking in dynamically typed languages, in: Proceedings of the 10th International Colloquium on Theoretical Aspects of Computing, Vol. 8049 of Lecture Notes in Computer Science, Springer, 2013, pp. 195–212.

[7] L. O. Andersen, Program analysis and specialization for the C programming language, Ph.D. thesis, DIKU, University of Copenhagen (May 1994).

[8] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proceedings of POPL, 1977.
URL `http://dl.acm.org/citation.cfm?id=512973`

[9] J. Siek, M. Vachharajani, Gradual typing with unification-based inference, in: Proceedings of DLS, 2008. doi:10.1145/1408681.1408688.
URL `http://portal.acm.org/citation.cfm?doid=1408681.1408688`

[10] M. Abadi, L. Cardelli, B. Pierce, G. Plotkin, Dynamic typing in a statically typed language, ACM Transactions on Programming Languages and Systems 13 (2) (1991) 237–268. doi:10.1145/103135.103138.
URL `http://portal.acm.org/citation.cfm?doid=103135.103138`

[11] T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, J. Vitek, Integrating typed and untyped code in a scripting language, in: Proceedings of POPL, 2010, pp. 377–388.
URL `http://dl.acm.org/citation.cfm?id=1706343`

[12] D. Ancona, M. Ancona, A. Cuni, N. D. Matsakis, RPython: a step towards reconciling dynamically and statically typed OO languages, in: Proceedings of DLS, 2007, pp. 53–64.

[13] A. H. Borning, D. H. H. Ingalls, A Type Declaration and Inference System for Smalltalk, in: Proceedings of POPL, 1982, pp. 133–141.

[14] O. Agesen, J. Palsberg, M. I. Schwartzbach, Type Inference of SELF: Analysis of Objects with Dynamic and Multiple Inheritance, in: Proceedings of ECOOP, 1993, pp. 247–267.
URL `http://onlinelibrary.wiley.com/doi/10.1002/spe.4380250903/abstract`

[15] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, D. Vardoulakis, In defense of soundiness: A manifesto, Commun. ACM 58 (2) (2015) 44–46. doi:10.1145/2644805.
URL `http://doi.acm.org/10.1145/2644805`

[16] C. Winter, T. Lownds, Python Enhancement Proposal (PEP) 3107, Available online: http://www.python.org/dev/peps/pep-3107/.

[17] N. Grech, G. Fourtounis, A. Francalanza, Y. Smaragdakis, Heaps don't lie: Countering unsoundness with heap snapshots, Proceedings of the ACM on Programming Languages 1 (OOPSLA). doi:10.1145/3133926.

[18] N. Grech, G. Fourtounis, A. Francalanza, Y. Smaragdakis, Shooting from the heap: Ultra-scalable static analysis with heap snapshots, in: International Symposium on Software Testing and Analysis (ISSTA), ISSTA '18, ACM, New York, NY, USA, 2018. doi:10.1145/3213846.3213860.

[19] O. Shivers, Control-flow analysis of higher-order languages, Ph.D. thesis, Carnegie Mellon University (1991).
URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.75.2777\&amp;rep=rep1\&amp;type=pdf`

[20] The Computer Language Benchmarks Game (2013).

[21] J. Siek, W. Taha, Gradual typing for functional languages, in: Proceedings of Scheme and Functional Programming Workshop, 2006.
URL `http://ece-www.colorado.edu/~siek/pubs/pubs/2006/siek06:_gradual.pdf`

[22] A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek, M. Felleisen, Is sound gradual typing dead?, in: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16, ACM, New York, NY, USA, 2016, pp. 456–468. doi:10.1145/2837614.2837630.
URL `http://doi.acm.org/10.1145/2837614.2837630`

[23] G. Richards, E. Arteca, A. Turcotte, The VM already knew that: Leveraging compile-time knowledge to optimize gradual typing, Proc. ACM Program. Lang. 1 (OOPSLA) (2017) 55:1–55:27. doi:10.1145/3133879.
URL `http://doi.acm.org/10.1145/3133879`

[24] G. M. Bierman, M. Abadi, M. Torgersen, Understanding typescript, in: Proceedings of ECOOP, 2014.

[25] A. Rastogi, N. Swamy, C. Fournet, G. Bierman, P. Vekris, Safe & efficient gradual typing for TypeScript, in: Proceedings of POPL, 2015.

[26] A. Chaudhuri, B. Hosmer, G. Levi, Flow, a new static type checker for JavaScript (2014).

[27] J. Verlaguet, A. Menghrajani, Hack: a new programming language for HHVM (2014).

[28] A. K. Wright, R. Cartwright, A practical soft type system for scheme, in: ACM Transactions on Programming Languages and Systems, Vol. 19, 1997, pp. 87–152. doi:10.1145/239912.239917.
URL `http://portal.acm.org/citation.cfm?doid=239912.239917`

[29] M. Salib, Starkiller: A Static Type Inferencer and Compiler for Python, Masters Thesis, Department of Electrical Engineering and Computer Science, MIT., 2004.

[30] S.-O. Nyström, A soft-typing system for Erlang, in: Proceedings of Erlang Workshop, 2003, pp. 56–71.

[31] G. Bracha, Pluggable type systems, in: OOPSLA workshop on revival of dynamic languages, 2004.
URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.175.1460\&rep=rep1\&type=pdf`

[32] W. Choi, S. Chandra, G. C. Necula, K. Sen, Sjs: A type system for JavaScript with fixed object layout., in: S. Blazy, T. Jensen (Eds.), SAS, Vol. 9291 of Lecture Notes in Computer Science, Springer, 2015, pp. 181–198.
URL `http://dblp.uni-trier.de/db/conf/sas/sas2015.html#ChoiCNS15`

[33] S. Chandra, C. S. Gordon, J.-B. Jeannin, C. Schlesinger, M. Sridharan, F. Tip, Y. Choi, Type inference for static compilation of JavaScript, SIGPLAN Not. 51 (10) (2016) 410–429. doi:10.1145/3022671.2984017.
URL `http://doi.acm.org/10.1145/3022671.2984017`

[34] B. Hackett, S.-y. Guo, Fast and precise hybrid type inference for JavaScript, in: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, ACM, New York, NY, USA, 2012, pp. 239–250. doi:10.1145/2254064.2254094.
URL `http://doi.acm.org/10.1145/2254064.2254094`

[35] S. H. Jensen, A. Møller, P. Thiemann, Type analysis for JavaScript, in: Proceedings of the 16th International Symposium on Static Analysis, SAS '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 238–255.
URL `http://dx.doi.org/10.1007/978-3-642-03237-0_17`

[36] B. S. Lerner, J. G. Politz, A. Guha, S. Krishnamurthi, Tejas: Retrofitting type systems for JavaScript, SIGPLAN Not. 49 (2) (2013) 1–16. doi:10.1145/2578856.2508170.
URL `http://doi.acm.org/10.1145/2578856.2508170`

[37] S. Tobin-Hochstadt, M. Felleisen, The design and implementation of typed Scheme, in: Proceedings of POPL, 2008, pp. 395–406.
URL `http://dl.acm.org/citation.cfm?id=1328486`

[38] M. Furr, J.-h. D. An, J. S. Foster, M. Hicks, Static type inference for Ruby, in: Symposium on Applied Computing, 2009, pp. 1859–1866. doi:10.1145/1529282.1529700.
URL `http://portal.acm.org/citation.cfm?doid=1529282.1529700`

[39] P. Heidegger, P. Thiemann, Recency Types for Dynamically-Typed, Object-Based Languages, in: Proceedings of FOOL, 2009.
URL `http://www.informatik.uni-freiburg.de/~linkenhe/2009-fool-heidegger-thiemann.pdf`

[40] G. Winskel, A note on model checking the modal μ-calculus, Theoretical Computer Science 83 (1) (1991) 157 – 167. doi:https://doi.org/10.1016/0304-3975(91)90043-2.
URL `http://www.sciencedirect.com/science/article/pii/0304397591900432`

[41] L. Simon, A. Bansal, A. Mallya, G. Gupta, Co-logic programming: Extending logic programming with coinduction, in: Proceedings of International Colloquium on Automata, Languages and Programming (ICALP), Vol. 4596 of LNCS, Springer-Verlag, 2007.
URL `https://doi.org/10.1007/978-3-540-73420-8_42`

[42] D. Ancona, Regular corecursion in prolog, Computer Languages, Systems & Structures 39 (4) (2013) 142 – 162, special issue on the Programming Languages track at the 27th ACM Symposium on Applied Computing. doi:https://doi.org/10.1016/j.cl.2013.05.001.
URL `http://www.sciencedirect.com/science/article/pii/S1477842413000158`

[43] D. Ancona, G. Lagorio, E. Zucca, Type inference by coinductive logic programming, in: Types for Proofs and Programs, Vol. 5497 of LNCS, 2008, pp. 1–18.

[44] D. Ancona, G. Lagorio, Coinductive type systems for object-oriented languages, in: Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming, Genoa, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 2–26.

[45] B. W. DeVries, G. Gupta, K. W. Hamlen, S. Moore, M. Sridhar, Actionscript bytecode verification with co-logic programming, in: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security, PLAS '09, ACM, New York, NY, USA, 2009, pp. 9–15. doi:10.1145/1554339.1554342.
URL `http://doi.acm.org/10.1145/1554339.1554342`

[46] O. Shivers, Control-flow analysis in Scheme, in: Proceedings of PLDI, 1988, pp. 164–174.
URL `http://dl.acm.org/citation.cfm?id=54007`

[47] J. Lerch, J. Späth, E. Bodden, M. Mezini, Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths (T), in: 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015, 2015, pp. 619–629. doi:10.1109/ASE.2015.9.

Appendix A. Full proofs

Lemma 1 (p-Bounding)**.** *For any variables u, v, truncated execution points s, s', and trails $\mathcal{T}, \mathcal{T}'$ such that $\mathcal{T}' \subseteq \mathcal{T}$. If*

$$\begin{aligned} \langle s, \mathcal{T}\rangle &\vdash_p u : \tau \\ \langle s', \mathcal{T}'\rangle &\vdash_p v : \tau' \\ \langle s, \mathcal{T} \cup \{\langle s', v\rangle\}\rangle &\vdash_p u : \tau'' \\ \tau'' &<: \tau' \end{aligned} \tag{A.1}$$

then also

$$\tau <: \tau' \tag{A.2}$$

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., 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 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 (A.1), 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 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 size$(\mathcal{T}_U - \mathcal{T}) = n$. If

$$\begin{aligned} \langle s, \mathcal{T}\rangle &\vdash_p u : \tau \\ \langle s', \mathcal{T}'\rangle &\vdash_p v : \tau' \\ \langle s, \mathcal{T} \cup \{\langle s', v\rangle\}\rangle &\vdash_p u : \tau'' \end{aligned} \tag{A.3}$$

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

$$\tau <: \tau' \tag{A.4}$$

We then show that it also holds for $n + 1$. For this, we choose two trail variables $\mathcal{T}''$ and $\mathcal{T}'''$, where 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 (A.3). 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 size$(\mathcal{T}_U - \mathcal{T}'') = n + 1$, and

$$\begin{aligned} \langle s, \mathcal{T}''\rangle &\vdash_p u : \tau \\ \langle s', \mathcal{T}'''\rangle &\vdash_p v : \tau' \\ \langle s, \mathcal{T}'' \cup \{\langle s', v\rangle\}\rangle &\vdash_p u : \tau'' \end{aligned} \tag{A.5}$$

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

$$\tau <: \tau' \tag{A.6}$$

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 8 and Figure 9) used in the proof.
Case pINIT. Looking up a variable in the entry point of the program.
We have $s = \varepsilon$ and we rewrite some of our judgements from (A.5) 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

$$\Sigma_I(u) : \tau$$
$$\Sigma_I(u) : \tau''$$

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 (A.5) 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 2.
Case pLC. u is acc, s has the form $\langle P, pc\rangle :: ...$ and P_{pc} is LC c.
We rewrite our judgements from (A.5) to:

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

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 8 follow the same pattern as this case.
We now look at the recursive rules in Figure 9. The proofs for these cases follow the same pattern. We will only look at the case for pLG and omit the other cases.
Case pLG. In this case u is acc, s has the form $\langle P, pc\rangle :: ...$ and P_{pc} is LG x and we rewrite our judgements from (A.5) accordingly to:

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

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

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

for all $s_i \in \text{prev}(s)$.
Let $\mathcal{T}$ be $\mathcal{T}'' \cup \{\langle s, acc\rangle\}$, 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 $\mathcal{T}''' \subseteq \mathcal{T}'' \subseteq \mathcal{T}'' \cup \{\langle s, acc\rangle\} = \mathcal{T}$. The hypothesis in (A.3) all hold and 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 2. *For any variable u, execution point s and trails $\mathcal{T}$, $\mathcal{T}'$ such that $\mathcal{T}' \subseteq \mathcal{T}$, $\tau <: \tau'$ where*

$$\begin{gathered}\langle s, \mathcal{T}\rangle \vdash_p u : \tau \\ \langle s, \mathcal{T}'\rangle \vdash_p u : \tau'\end{gathered} \tag{A.7}$$

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., 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 (A.7):

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

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 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

$$\begin{gathered}\langle s, \mathcal{T}\rangle \vdash_p u : \tau \\ \langle s, \mathcal{T}'\rangle \vdash_p u : \tau'\end{gathered} \tag{A.8}$$

We now show that it also holds for $n+1$. For this we choose trail variable $\mathcal{T}''$, where 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 size$(\mathcal{T}_U - \mathcal{T}'') = n + 1$, $\tau <: \tau'$ where

$$\begin{gathered}\langle s, \mathcal{T}''\rangle \vdash_p u : \tau \\ \langle s, \mathcal{T}'''\rangle \vdash_p u : \tau'\end{gathered} \tag{A.9}$$

We proceed by analysing the last rule used in the proof of $\langle s, \mathcal{T}'''\rangle \vdash_p u : \tau'$
Case pINIT. Looking up a variable in the entry point of the program.
In this case $s = \varepsilon$, and we can therefore rewrite (A.9) as:

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

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 8 follow the same pattern. We only elaborate the case that matches rule pLC.
Case pLC. u is *acc*, s has the form $\langle P, pc\rangle :: ...$ and P_{pc} is LC c.
We rewrite our judgements from (A.9) as

$$\begin{gathered}\langle s, \mathcal{T}''\rangle \vdash_p acc : \tau \\ \langle s, \mathcal{T}'''\rangle \vdash_p acc : \tau'\end{gathered}$$

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 9, all follow the same pattern. We only elaborate the case that matches rule pLG.
Case pLG. u is *acc*, s has the form $\langle P, pc\rangle :: ...$ and P_{pc} is LG x.

We rewrite our judgements from (A.9) as

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

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

$$\begin{aligned} \langle s_i, \mathcal{T}'' \cup \{\langle s, acc\rangle\}\rangle \vdash_p x : \tau_i \\ \langle s_i, \mathcal{T}''' \cup \{\langle s, acc\rangle\}\rangle \vdash_p x : \tau_i' \end{aligned} \tag{A.10}$$

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 $\mathcal{T}'' \cup \{\langle s, acc\rangle\}$ and $\mathcal{T}'$ be $\mathcal{T}''' \cup \{\langle s, acc\rangle\}$. Then by rewriting (A.10) we have the hypothesis (A.8), 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. □

Lemma 3 (f-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*

$$\begin{aligned} \langle s, \mathcal{T}' \rangle \vdash_f v : \tau \\ \langle s', \mathcal{T} \rangle \vdash_f u : \tau' \\ \langle s', \mathcal{T} \cup \{\langle s, v\rangle\}\rangle \vdash_f u : \tau'' \end{aligned} \tag{A.11}$$

Proof. We proceed by induction on n, where as in Lemma 1, 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 $\tau' = \perp$ 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

$$\begin{aligned} \langle s, \mathcal{T}' \rangle \vdash_f v : \tau \\ \langle s', \mathcal{T} \rangle \vdash_f u : \tau' \\ \langle s', \mathcal{T} \cup \{\langle s, v\rangle\}\rangle \vdash_f u : \tau'' \end{aligned} \tag{A.12}$$

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

$$\begin{aligned} \langle s, \mathcal{T}''' \rangle \vdash_f v : \tau \\ \langle s', \mathcal{T}'' \rangle \vdash_f u : \tau' \\ \langle s', \mathcal{T}'' \cup \{\langle s, v\rangle\}\rangle \vdash_f u : \tau'' \end{aligned} \tag{A.13}$$

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 2. 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 2. 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. $s' = \langle P, pc\rangle :: \varepsilon$ 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. u is acc, s' has the form $\langle P', pc' \rangle :: ...$ and $P'_{pc'}$ is $\mathsf{JIF}\ n$.
In this case τ' is defined such that

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

and τ'' is defined such that

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

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 11. The proofs for these cases are all similar to each other, with the most intricate being the case that matches fLG.
Case fLG. u is x, s' has the form $\langle P', pc' \rangle :: ...$ and $P'_{pc'}$ is $\mathsf{LG}\ x$.
We rewrite our judgements from (A.13) into

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

By fLG, $\tau' = \bigsqcup(\upsilon'_i \sqcap \nu'_i)$ and $\tau'' = \bigsqcup(\upsilon''_i \sqcap \nu''_i)$, where υ'_i, ν'_i, υ''_i and ν''_i are defined as follows

$$\begin{gathered}\langle s'_i, \mathcal{T}'' \cup \{\langle s', x \rangle\} \rangle \vdash_f acc : \upsilon'_i \\ \langle s'_i, \mathcal{T}'' \cup \{\langle s', x \rangle\} \rangle \vdash_f x : \nu'_i \\ \langle s'_i, \mathcal{T}'' \cup \{\langle s', x \rangle\} \cup \{\langle s, v \rangle\} \rangle \vdash_f acc : \upsilon''_i \\ \langle s'_i, \mathcal{T}'' \cup \{\langle s', x \rangle\} \cup \{\langle s, v \rangle\} \rangle \vdash_f x : \nu''_i\end{gathered}$$

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

$$\begin{gathered}\langle s'_i, \mathcal{T} \rangle \vdash_f acc : \upsilon'_i \\ \langle s'_i, \mathcal{T} \rangle \vdash_f x : \nu'_i \\ \langle s'_i, \mathcal{T} \cup \{\langle s, v \rangle\} \rangle \vdash_f acc : \upsilon''_i \\ \langle s'_i, \mathcal{T} \cup \{\langle s, v \rangle\} \rangle \vdash_f x : \nu''_i\end{gathered}$$

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

$$\begin{aligned}\tau' = \bigsqcup(\upsilon'_i \sqcap \nu'_i) <: \bigsqcup(\tau \sqcup \upsilon''_i) \sqcap (\tau \sqcup \nu''_i) &= \bigsqcup \tau \sqcup (\upsilon''_i \sqcap \nu''_i) \\ &= \tau \sqcup \bigsqcup(\upsilon''_i \sqcap \nu''_i) \\ &= \tau \sqcup \tau''\end{aligned}$$

as required. $\square$