Preemptive Type Checking in Dynamically Typed Languages

Neville Grech*, Julian Rathke, and Bernd Fischer

Electronics and Computer Science, University of Southampton
{n.grech,jr2,b.fischer}@ecs.soton.ac.uk

Abstract. We describe a type system that identifies potential type errors in dynamically typed languages like Python. The system uses a flow-sensitive static analysis on bytecodes to compute, for every variable and program point, over-approximations of the variable's present and future use types. If the future use types are not subsumed by the present types, the further program execution may raise a type error, and a narrowing assertion is inserted; if future use and present types are disjoint, it will raise a type error, and a type error assertion is inserted. We prove that the assertions are inserted in optimal locations and thus preempt type errors earlier than dynamic, soft, and gradual typing. We describe the details of our type inference and assertion insertion, and demonstrate the results of an implementation of the system with a number of examples.

1 Introduction

Dynamically typed languages such as Python are among the most widely used languages [25]. In these languages, the principle type of any variable in a program is determined through runtime computations and can change throughout the execution and between different runs. Type checking is typically carried out as the program is executing and type errors manifest themselves as runtime errors or exceptions rather than being detected before execution. However, type errors are an indication that the code has latent computation errors and is therefore potentially dangerous. For example, the Mars climate orbiter crashed into the atmosphere due to *metric mixup* [1]. The earlier type errors are detected, the earlier the code can be fixed.

Fig. 1 shows a small example program with type errors. In a dynamically typed language, the program will fail at either line 15 or line 17, depending on whether arguments are passed to the program. Using the standard Python interpreter we can get for example the following trace:

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

* The research work disclosed in this publication is partially funded by a Strategic Educational Pathways Scholarship (Malta). The scholarship is part-financed by the European Union – European Social Fund (ESF).

Z. Liu, J. Woodcock, and H. Zhu (Eds.): ICTAC 2013, LNCS 8049, pp. 195–212, 2013.

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

Fig. 1. Dynamically typed program with type errors in lines 15 and 17

We can see that the program only raises a type error when it executes line 15, after the user input has already been taken. We cannot see from the error trace, however, that the program actually contains another type error, i.e., at line 17, which in cases when the modulus of the entered number with 5 is less than 3. To discover this error, we are reliant on sufficient testing.

Our goal here is the development of a *preemptive type checking* system that statically analyses the program, and inserts type checking assertions that preempt (i.e., force the termination of) the program execution as soon as a type error becomes inevitable. In contrast, under the existing dynamic, gradual [18,19] or soft [7] typing systems, these errors are only caught at the point that a value of an incorrect type is used. In the example, preemptive type checking finds both errors and presents the same error traces as shown above. Moreover, it inserts a type error assertion at the beginning of the `main` function that prevents the program from executing at all, since *all* program executions will lead to a type error:

```
def main():
    raise TypeError('Type mismatch at lines 15, 17: expected Number, found str')
    ...
```

Now we assume that the user "fixes" this bug and manually inserts explicit type casts into the `main` function:

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

However, when this program is run without preemptive type checking, the program will, depending on the input, either raise a type error or work as expected, for example:

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

$ python foo.py
enter initial value: 2
enter final value: 3
...
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

As we can see, the manual debugging process is time consuming, and relies on the right combination of inputs to find the type errors. With preemptive type checking we can minimise this effort and find and correct type errors much quicker. Our analysis *statically* infers that `x1` and `x2` are either of type `NoneType` or integers, depending on the control flow taken by the program. It also concludes that `x1` and `x2` need to be integers for the program not to raise type errors:

```
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
```

Note that it is difficult and expensive to determine the possible types of `x1` and `x2`. For example, using data flow analysis techniques, the fact that `x1` can be an integer is only discovered on a path that inlines function `compute` three times. We thus introduce an effective technique that uses *trails* to perform a flow sensitive type inference.

Preemptive type checking also transforms the `compute` function so that the type errors are preempted (see Fig. 2). The inserted assertions contain all details to identify the source of the type error, in particular the variable causing the type error, the location where the type error would be raised and the present type there. Hence, the user can correct the program with minimal debugging. Note that the assertions cannot be inserted any earlier (i.e., before the if-statement) because there are possible control flow paths that do not raise type errors.

Preemptive type checking identifies potential type errors in advance through a flow-sensitive static analysis. It computes, for every variable and every program point, an over-approximation of the types of the values that have last been assigned to a variable (its "present types") as well as the types with which it is next used in any reachable program point (its "future use types"). If the future use types are not subsumed by the present types, the further program execution *may* raise a type error, and a corresponding narrowing assertion is inserted; if future use types and present types are disjoint, the

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

Fig. 2. Transformed version of the `compute` function

further program execution *will* raise a type error, and a corresponding type error assertion is inserted. We prove that the assertions are inserted in optimal locations and thus preempt type errors earlier than dynamic typing, gradual typing [18,19], and soft typing [7]. We further show that these assertions do not change the semantics of programs that do not raise type errors. We proceed by formalising the type system and corresponding bytecode level type inference. Although the theory is presented for μPython, a dynamically typed Python-like core language, the techniques presented are applicable for any similar dynamically typed language such as Ruby or JavaScript, or indeed larger subsets of Python as in our implementation. Finally, we describe an implementation of preemptive type checking, including assertion insertion, for a subset of Python bytecodes, and evaluate it on a some benchmarks.

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 static type analysis.

High-Level Syntax. We present the high-level syntax of μPython in Fig. 3 for illustrative purposes only, as our type analysis is exclusively performed at the bytecode level. The base types of the language are standard except perhaps for the types Un of uninitialised variables 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 prime integer and string operations, which implicitly raise a type error if their argument is of the wrong type. Note that conditional statements and function calls will also implicitly raise a type error when their guard or function expressions do not evaluate to boolean or function types respectively. This contrasts with the raise operation that will immediately raise an *explicit* exception error to terminate execution.

Statements:

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

Expressions:

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

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

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

Fig. 3. Syntax of the μPython language

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, where functions are first class citizens.

Bytecode. Our type analysis is defined on the μPython bytecode. This is based on a simplified machine model consisting of a store (for mapping variables to constants), an integer-valued program counter and a single accumulator acc. The full Python VM is a stack based machine and for presentation purposes we replace the evaluation stack with an accumulator acc. Our implementation of preemptive type checking supports full 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 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.

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

Fig. 4. The μPython bytecodes

The actual bytecodes we use are given in Fig. 4. Loading places constant values in the accumulator, storing moves a constant to store from the accumulator. We assume well-formed bytecode where jumps only refer to actual program locations and every program has a RET-instruction at its final location. Note that JIF consumes the accumulator value as part of its test. The instructions intOp, strOp and raise echo the

$$
\begin{array}{rll}
\langle \Sigma, \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 \\
\quad \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{Exn} & \text{if } P_{pc} = \mathsf{raise}
\end{array}
$$

Fig. 5. Semantics of the μPython Bytecode

corresponding high-level expressions of the same names and isInst writes a boolean into the accumulator depending on whether it contains a value of the given type. The CF f instruction is of interest: to execute this the machine finds the sequence of instructions P' mapped from f in the store and pushes this program on to the call stack with program counter 0.

Reduction Semantics. We formalise this semantics by the rules for single execution steps of the abstract machine shown in Fig. 5. The states of the machine, $State^{\rightarrow}$, 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, Exn, or End. We assume that the machine begins in state $\langle \Sigma, \varepsilon \rangle$. The step applicable at this point loads $\langle M, 0 \rangle :: \varepsilon$ onto the call stack, where M is the initial, or 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 principal type τ.

3 Type Inference for μPython

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

We call these the *present* and *future use* types. To establish the former we perform a traditional forwards analysis over the execution points of the program; the present type of a variable depends on the instructions that have previously been executed. Obviously the precise present runtime type of a variable cannot be statically determined so our analysis uses an over-approximation of this. 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

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

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

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.

3.1 Execution Points and Trails

A naive idea of a program execution point might be a simple code location but because variables can change type during execution, the entire call stack is important in determining their current types. In principle, program execution points must therefore be full call stacks. The control flow graph (CFG) of a μPython program is then a relation $S \to S'$ between call stacks. This is unfortunate because, even for finite programs, the CFG of all possible program execution points could then be infinite. This has drastic consequences for a static analysis.

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

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

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

At the heart of our analysis is the forwards/backwards traversal of the truncated CFG using the $\mathrm{prev}(s)$ and $\mathrm{next}(s)$ functions in order to find the present and future use types of variables. Of course, these CFGs may contain cycles so we must take care to terminate our analysis in cases where we have reached a point that we have previously visited. This motivates the following: the type inferencer is expressed using two independent inductively defined relations written

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

where s is a truncated execution point and $\mathcal{T}$ is a *trail*. A trail is a set of pairs $\langle s, u \rangle$ of truncated execution points and variables. They represent the previously visited execution points (together with the variables that triggered the visit) and are used to ensure termination of the inferencer, as explained in the next section. 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 onwards without raising a TypeError.

3.2 Type Inference Rules

The type inferencer is expressed as inference rules, given in Fig. 6 and Fig. 7. The *leaf rules* in Fig. 6 for inferring $\vdash_p$ account for situations in which the present type is fully determined by the current instruction. For example, after loading a constant (Rule pLC) the accumulator is known to have the type of the constant that has just been loaded. The non-leaf rules all follow a shared pattern: the types of *relevant* variables in each previous state are calculated and the present type of a specific variable is the union across the types from each previous state. The relevant variables are instruction dependent. For example, in Rule pSG1 for the instruction SG x the type of x depends on the type of acc in the previous states.

Again, for the rules for $\vdash_f$ in Fig. 7 we have leaf rules and non-leaf rules. Many of the leaf rules assign an f-type of $\top$ to a variable. This follows in cases where that variable is just about to be overwritten (Rules fSET/SG1). Otherwise, the immediate uses are recorded in the type (Rules fJIF/STR/INT). Two interesting rules are fLG1 and fCF1. In these a variable is used but its contents remain intact so there may be future uses also. We define a *meet* operation on types, written as $\sqcap$, in the following rules (applied in top-down order):

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

It is worth noting that the trail sets $\mathcal{T}$ are finitely bounded. This is due to the fact that call stacks are truncated to a fixed depth and that, for a given program, there are finitely many code locations and finitely many variables. For a given program, we write $\mathcal{T}_U$ to denote the maximum trail containing all truncated execution point/variable pairs. In fact, by virtue of the fact that trail sizes strictly decrease in non-leaf rules, that all rules have finitely many hypotheses, and by König's Lemma, it is guaranteed that the

Leaf rules:

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

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

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

Non-leaf rules:

$$\frac{\langle s, x\rangle \notin \mathcal{T} \quad P_{pc} = \textsf{SG}\ x \quad \langle s_i, \mathcal{T} \cup \{\langle s, x\rangle\}\rangle \vdash_p acc : \tau_i}{\langle s, \mathcal{T}\rangle \vdash_p x : \bigsqcup \tau_i}\textsf{pSG1} \qquad \frac{\langle s, acc\rangle \notin \mathcal{T} \quad P_{pc} = \textsf{LG}\ x \quad \langle s_i, \mathcal{T} \cup \{\langle s, acc\rangle\}\rangle \vdash_p x : \tau_i}{\langle s, \mathcal{T}\rangle \vdash_p acc : \bigsqcup \tau_i}\textsf{pLG1}$$

$$\frac{\langle s, y\rangle \notin \mathcal{T} \quad P_{pc} = \textsf{SG}\ x \quad x \neq y \quad \langle s_i, \mathcal{T} \cup \{\langle s, y\rangle\}\rangle \vdash_p y : \tau_i}{\langle s, \mathcal{T}\rangle \vdash_p y : \bigsqcup \tau_i}\textsf{pSG2} \qquad \frac{\langle s, y\rangle \notin \mathcal{T} \quad P_{pc} = \textsf{LG}\ x \quad \langle s_i, \mathcal{T} \cup \{\langle s, y\rangle\}\rangle \vdash_p y : \tau_i}{\langle s, \mathcal{T}\rangle \vdash_p y : \bigsqcup \tau_i}\textsf{pLG2}$$

$$\frac{\langle s, u\rangle \notin \mathcal{T} \quad P_{pc} \in \{\textsf{RET}, \textsf{JP}\ pc', \textsf{CF}\ f\} \quad \langle s_i, \mathcal{T} \cup \{\langle s, u\rangle\}\rangle \vdash_p u : \tau_i}{\langle s, \mathcal{T}\rangle \vdash_p u : \bigsqcup \tau_i}\textsf{pRET/JP/CF} \qquad \frac{\langle s, x\rangle \notin \mathcal{T} \quad P_{pc} \in \{\textsf{LC}\ c, \textsf{JIF}\ pc', \textsf{strOp}, \textsf{intOp}, \textsf{isInst}\ \tau\} \quad \langle s_i, \mathcal{T} \cup \{\langle s, x\rangle\}\rangle \vdash_p x : \tau_i}{\langle s, \mathcal{T}\rangle \vdash_p x : \bigsqcup \tau_i}\textsf{p}^\star$$

Fig. 6. Inference rules for the $\vdash_p$ judgement. Unless stated otherwise, s is assumed to be of the form $\langle P, pc\rangle :: \ldots$ and s_i ranges over $\mathrm{prev}(s)$.

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'.

4 Correctness

We now show that the type inference rules are correct. We give proof sketches for the main results. Full proofs can be found in [11]. The notion of soundness for p-types should be clear. Given a derivation $\langle s, \mathcal{T}_\emptyset\rangle \vdash_p u : \tau$ we expect that the actual runtime type of the constant stored at u in the Σ store after the current instruction in s has been executed to be a subtype of τ. This is formally expressed in the next theorem.

Theorem 1 (Soundness of p-types). *Suppose*

$$\langle \Sigma, \varepsilon\rangle \rightarrow^* \langle \Sigma, S\rangle \rightarrow \langle \Sigma', S'\rangle \quad \textit{and} \quad \langle \lfloor S \rfloor, \mathcal{T}_\emptyset\rangle \vdash_p x : \tau_p$$

and suppose τ_r is such that $\Sigma'(x) : \tau_r$. Then $\tau_r <: \tau_p$.

Proof. (Sketch) The proof proceeds by induction on the number of reduction steps taken to reach $\langle \Sigma, S\rangle$. For the base case we know that $\langle \Sigma, S\rangle$ is of the form $\langle \Sigma, \varepsilon\rangle$

Leaf rules:

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

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

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

Non-leaf rules:

$$\frac{\langle s, x\rangle \notin \mathcal{T} \quad P_{pc} = \mathsf{LG}\ x \quad \langle s_i, \mathcal{T} \cup \{\langle s, x\rangle\}\rangle \vdash_f acc : v_i \quad \langle s_i, \mathcal{T} \cup \{\langle s, x\rangle\}\rangle \vdash_f x : \nu_i}{\langle s, \mathcal{T}\rangle \vdash_f x : \bigsqcup(v_i \sqcap \nu_i)}\,\mathsf{fLG1}$$

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

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

$$\frac{\langle s, y\rangle \notin \mathcal{T} \quad P_{pc} \in \{\mathsf{LG}\ x, \mathsf{SG}\ x\} \quad x \neq y \quad \langle s_i, \mathcal{T} \cup \{\langle s, y\rangle\}\rangle \vdash_f y : \tau_i}{\langle s, \mathcal{T}\rangle \vdash_f y : \bigsqcup \tau_i}\,\mathsf{fLG2/SG3}$$

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

$$\frac{\langle s, u\rangle \notin \mathcal{T} \quad P_{pc} = \mathsf{CF}\ f \quad u \neq f \quad \langle s_i, \mathcal{T} \cup \{\langle s, u\rangle\}\rangle \vdash_f u : \tau_i}{\langle s, \mathcal{T}\rangle \vdash_f u : \bigsqcup \tau_i}\,\mathsf{fCF2}$$

$$\frac{\langle s, u\rangle \notin \mathcal{T} \quad P_{pc} = \mathsf{JP}\ n \quad \langle s_i, \mathcal{T} \cup \{\langle s, u\rangle\}\rangle \vdash_f u : \tau_i}{\langle s, \mathcal{T}\rangle \vdash_f u : \bigsqcup \tau_i}\,\mathsf{fJP}$$

$$\frac{P_{pc} \in \{\mathsf{LC}\ c, \mathsf{JIF}\ n, \mathsf{intOp}, \mathsf{strOp}, \mathsf{isInst}\ \tau\} \quad \langle s, x\rangle \notin \mathcal{T} \quad \langle s_i, \mathcal{T} \cup \{\langle s, x\rangle\}\rangle \vdash_f x : \tau_i}{\langle s, \mathcal{T}\rangle \vdash_f x : \bigsqcup \tau_i}\,\mathsf{f^*}$$

Fig. 7. Inference rules for the $\vdash_f$ judgement. Unless stated otherwise, s is assumed to be of the form $\langle P, pc\rangle :: \ldots$ and s_i ranges over $\mathrm{prev}(s)$.

and that there is a unique reduction step from this state whose target has store Σ_I (cf. Fig. 5). The type rule pINIT then guarantees the desired result. The inductive case requires a case analysis on the last type rule used to derive type τ_p. The leaf rules all follow from the definition of reduction but the non-leaf rules require use of the inductive hypothesis along with the following lemma that relates the types of variables as the trail sets are increased. □

Lemma 1 (Bounding). *For all u, v, s, s' and all $\mathcal{T}' \subseteq \mathcal{T}$ such that $\langle s, \mathcal{T}'\rangle \vdash_p u : \tau'$ and $\langle s', \mathcal{T} \cup \{\langle s, u\rangle\}\rangle \vdash_p v : \tau''$ then $\tau <: \tau' \sqcup \tau''$ whenever $\langle s', \mathcal{T}\rangle \vdash_p v : \tau$.*

Intuitively, the lemma states that the most type information that can be gained for u in the absence of the trail assumption $\langle s, v\rangle$ is what can be established for u, with the assumption in place, along with any possible contribution to the type from v itself.

The correctness criteria for f-types are more subtle. The f-types describe constraints on 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 must mean that, supposing we execute the program in a preemptive type checked semantics, if we raise a type error exception then the same program running in the non-preemptive 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 will need to define the preemptive 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 $State^{\rightarrow}$ is called a diverge-error relation if whenever $\langle \Sigma, S\rangle \in R^{\Uparrow}$ then*

$$\langle \Sigma, S\rangle \rightarrow \langle \Sigma', S'\rangle \wedge \langle \Sigma', S'\rangle \in R^{\Uparrow} \quad or \quad \langle \Sigma, S\rangle \rightarrow \mathsf{TypeError}.$$

Let $\Uparrow$ be the largest diverge-error relation.

It follows that a state in a diverge-error relation cannot reach the state End or Exn.

Definition 2. *The state compatibility predicate SC holds at $\langle \Sigma, S\rangle$ if for all variables u such that $\langle s, \mathcal{T}_\emptyset\rangle \vdash_f u : \tau_f$ and $\Sigma(u) : \tau_r$ then $\tau_r <: \tau_f$, where $s = \lfloor S \rfloor$.*

The next proposition demonstrates that this simple predicate would already by sufficient for preemptive type checking. However, we will see in the next section that SC may be refined to make better use of static type information.

Proposition 1 (Soundness of f-types). *If $\langle \Sigma, S\rangle \notin SC$ then $\langle \Sigma, S\rangle \in \Uparrow$.*

Proof. (Sketch) We use coinduction here by proving that the complement of SC is itself a diverge-error relation. To do this we suppose that $\langle \Sigma, S\rangle \notin SC$ to see that there is some u for which $\Sigma(u) : \tau_r$, and $\langle s, \mathcal{T}_\emptyset\rangle \vdash_f u : \tau_f$ and $\tau_r \not<: \tau_f$. We perform a case analysis on the last rule used to derive the type τ_f and see that, for all applicable leaf rules, then state $\langle \Sigma, S\rangle$ reduces to TypeError. For all non-leaf rules, a lemma analogous to Lemma 1 is used to show that where $\langle \Sigma, S\rangle$ reduces to some $\langle \Sigma', S'\rangle$ then $\langle \Sigma', S'\rangle \notin SC$ as required. □

4.1 Checked μPython Semantics

The naive runtime type check SC above 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 EC holds at $\langle s, s', \Sigma'\rangle$ if for all variables u, such that*

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

then

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

Essentially this says that, if the program moves from a state s to a state s' then the f-types report no error if there is no change in the future use constraints, if the statically approximated runtime type is a subtype of future uses, or if the actual new runtime type of a variable is within the future use set (modulated by the present type). Clearly only the latter of these requires the inspection of the runtime types and even then, where the meet $\tau_p \sqcap \tau'_f$ is $\bot$, we know statically that the predicate must fail as there are no constants of type $\bot$. The predicate EC 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 FE holds at $\langle s, s' \rangle$ if $s \in \mathrm{prev}(s')$ and either $\forall \Sigma' \cdot \langle s, s', \Sigma' \rangle \notin EC'$ or $\{\langle s', s'' \rangle \mid s'' \in \mathrm{next}(s')\} \subseteq FE$.*

Definition 5. *The checked semantics is defined as a binary relation $\rightsquigarrow$ on the set of states, $State^{\rightsquigarrow}$ comprised of $\langle \Sigma, S \rangle$ states,* End*, and* Exn *such that:*

$$\begin{array}{ll}
\langle \Sigma, S \rangle \rightsquigarrow \mathsf{End} & \text{if} \langle \Sigma, S \rangle \to \mathsf{End} \\
\langle \Sigma, S \rangle \rightsquigarrow \mathsf{Exn} & \text{if} \langle \Sigma, S \rangle \to \mathsf{Exn} \\
\langle \Sigma, S \rangle \rightsquigarrow \mathsf{Exn} & \text{if} \langle \Sigma, S \rangle \to \langle \Sigma', S' \rangle \wedge \langle s, s', \Sigma' \rangle \notin EC \\
\langle \Sigma, S \rangle \rightsquigarrow \mathsf{Exn} & \text{if} \langle \Sigma, S \rangle \to \langle \Sigma', S' \rangle \wedge \langle s, s' \rangle \in FE \\
\langle \Sigma, S \rangle \rightsquigarrow \langle \Sigma', S' \rangle & \text{if} \langle \Sigma, S \rangle \to \langle \Sigma', S' \rangle \textit{ otherwise}
\end{array}$$

Definition 6. *A relation $R^{\leq}$ on $State^{\to} \times State^{\rightsquigarrow}$, which relates only identical non-terminating states (i.e., if $\langle \Sigma, S \rangle R^{\leq} \langle \Sigma_1, S_1 \rangle$ then $\Sigma = \Sigma_1$ and $S = S_1$) is called an error-preserving simulation if the following holds:*

- $\langle \Sigma, S \rangle \not\to \mathsf{TypeError}$
- *If* $\langle \Sigma, S \rangle \to \mathsf{End}$ *then* $\langle \Sigma, S \rangle \rightsquigarrow \mathsf{End}$.
- *If* $\langle \Sigma, S \rangle \to \mathsf{Exn}$ *then* $\langle \Sigma, S \rangle \rightsquigarrow \mathsf{Exn}$.
- *If* $\langle \Sigma, S \rangle \to \langle \Sigma', S' \rangle$ *then either*
  - $\langle \Sigma, S \rangle \rightsquigarrow \langle \Sigma', S' \rangle \wedge \langle \Sigma', S' \rangle \in R^{\leq}$ *or*
  - $\langle \Sigma, S \rangle \rightsquigarrow \mathsf{Exn} \wedge \langle \Sigma', S' \rangle \in \Uparrow$

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

Theorem 2. *Let R^{SC} be defined as*

$$\{\langle \Sigma, S \rangle, \langle \Sigma, S \rangle \mid \langle \Sigma_I, \langle M, 0 \rangle :: \varepsilon \rangle \to^* \langle \Sigma, S \rangle \wedge \langle \Sigma, S \rangle \in SC\}$$

Then R^{SC} is an error-preserving simulation and hence $R^{SC} \subseteq \lesssim$.

Proof. (Sketch) It is easy to see that states $\langle \Sigma, S \rangle$ in R^{SC} preserve termination steps. To show that $\langle \Sigma, S \rangle \not\to \mathsf{TypeError}$ we use proof by contradiction by assuming a type error and analyse all possible reduction steps that could cause this. In each case the inferred types must contradict the hypothesis that $\langle \Sigma, S \rangle \in SC$. To show that transitions are preserved by matching checked transitions or exceptions that guarantee future

divergence we consider the possible derivations of the checked semantics. In case that $\langle s, s', \Sigma' \rangle \notin EC$ we note $\tau'_r \not<: \tau_p \sqcap \tau'_f$ and thus, using Theorem 1 and Proposition 1, we have $\langle \Sigma', S' \rangle \in \Uparrow$. In case $\langle s, s', \Sigma' \rangle \in EC$ we analyse each type rule to show that $\langle \Sigma', S' \rangle \in SC$. □

Corollary 1. *Suppose* $\langle \Sigma_I, \langle M, 0 \rangle :: \varepsilon \rangle \rightsquigarrow^* N \not\rightsquigarrow$. *Then* N *is either* End *or* Exn.

Proof. We note immediately that $\langle \Sigma, \varepsilon \rangle \in SC$ holds by virtue of rule fINIT of Fig. 7. Therefore we have $\langle \Sigma, \varepsilon \rangle R^{SC} \langle \Sigma, \varepsilon \rangle$ and hence by the above theorem we have $\langle \Sigma, \varepsilon \rangle \lesssim \langle \Sigma, \varepsilon \rangle$. Now, suppose for contradiction that N is neither End or Exn. 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 $\langle \Sigma, S \rangle \not\rightarrow$ TypeError and, by the definition of $\rightarrow$ we must have $\langle \Sigma, S \rangle \rightarrow \langle \Sigma', S' \rangle$ for some $\langle \Sigma', S' \rangle$. This means that $N \rightsquigarrow N'$ for some N' also, contradicting maximality. □

4.2 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 Exn as soon as the control flow reaches a point where all possible further execution steps in the unchecked semantics lead to state TypeError. 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.

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

Theorem 3 (Linear optimality). *Suppose* $\langle \Sigma, \varepsilon \rangle \rightsquigarrow^* \langle \Sigma, S \rangle$ *such that there is a linear reduction sequence* $\langle \Sigma, S \rangle \rightarrow^*$ TypeError. *Then* $\langle \Sigma, S \rangle \rightsquigarrow$ Exn.

Proof. (Sketch) This is proved by assuming for contradiction that $\langle \Sigma, S \rangle \rightsquigarrow \langle \Sigma', S' \rangle$ and $\langle \Sigma', S' \rangle \rightsquigarrow$ Exn. We consider the cases that derive the latter step and can quickly rule out $\langle \Sigma', S' \rangle$ being the source of a fail edge because the assumption of linearity guarantees that $\langle s, s', \Sigma' \rangle \in FE$ in this case, which contradicts our assumption. Therefore we must have $\langle s', s'', \Sigma'' \rangle \notin EC$. We then consider the type rules used to derive the f-type in state $\langle \Sigma', S' \rangle$ and use these to derive a contradiction. □

Linear optimality is not as restrictive as it might seem: the fail edge predicate FE propagates guaranteed type errors backwards even over control flow splits.

4.3 Type Checks Insertion

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

$P' \longleftarrow \varepsilon$
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 $P_{pc} = \mathsf{JIF}\ \ pc' \wedge s' = \langle P, pc' \rangle :: \ldots \wedge \langle s, s' \rangle \in FE$: $\mathrm{extend}(P', \underline{\mathrm{failIfFalse}})$
    if $P_{pc} = \mathsf{JIF}\ \ pc' \wedge s' = \langle P, pc + 1 \rangle :: \ldots \wedge \langle s, s' \rangle \in FE$: $\mathrm{extend}(P', \underline{\mathrm{failIfTrue}})$
    if $\langle \varepsilon, s \rangle \in FE$: $\mathrm{extend}(P', \mathsf{raise})$
    if $P_{pc} \notin \{\mathsf{JIF}\ \ pc', \mathsf{CF}\ \ f, \mathsf{JP}\ \ pc'\}$: $\mathrm{extend}(P', P_{pc})$
    for $x \in \mathbb{V}$:
      let $\langle s, \mathcal{T}_\emptyset \rangle \vdash_p x : \tau_p \quad \langle s, \mathcal{T}_\emptyset \rangle \vdash_f x : \tau_f \quad \langle s', \mathcal{T}_\emptyset \rangle \vdash_f x : \tau'_f$
      if $\neg(\tau_f = \tau'_f \vee \tau_p <: \tau'_f)$:
        if $P_{pc} = \mathsf{JIF}\ \ pc' \wedge s' = \langle P, pc' \rangle :: \ldots$: $\mathrm{extend}(P', \underline{\mathrm{checkIfFalse}}(x, \tau_p \sqcap \tau'_f))$
        if $P_{pc} = \mathsf{JIF}\ \ pc' \wedge s' = \langle P, pc + 1 \rangle :: \ldots$: $\mathrm{extend}(P', \underline{\mathrm{checkIfTrue}}(x, \tau_p \sqcap \tau'_f))$
        if $P_{pc} \neq \mathsf{JIF}\ \ pc'$: $\mathrm{extend}(P', \underline{\mathrm{check}}(x, \tau_p \sqcap \tau'_f))$
    if $P_{pc} = \mathsf{CF}\ \ f$:
      $\langle Q, 0 \rangle :: \ldots \longleftarrow s'$
      $\mathrm{extend}(P', \underline{\mathrm{call}}(\mathrm{specialise}(Q, s)))$
  if $P_{pc} = \mathsf{JIF}\ \ pc' \vee P_{pc} = \mathsf{JP}\ \ pc'$: $\mathrm{extend}(P', P_{pc})$

Fig. 8. Algorithm for inserting type checks in μPython programs, expressed as a function $\mathrm{specialise}(P, s)$ that returns an updated program P'

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. Where the same program is reached with a different call stack, 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 (see Fig. 8) 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 FE and EC predicates. The algorithm uses bytecode macros that are underlined in the algorithm and implemented as a sequence of μPython bytecode instructions. Procedure extend takes a program and a list of instructions and appends the instructions to the end of the given program.

5 Implementation

We have implemented the system as a Python library for a subset of Python 3.3. We suppport both local and global variables, which helps make the system scalable as most variables in typical programs are local. Our implementation handles 40 bytecodes in total. In particular, we support extra bytecodes for arithmetic, more control structures such as while-loops, local variables, some built-in data structures and polyadic functions.

Architecture. In Python the load path for individual modules can sometimes only be resolved at runtime, and the bytecode for a module that requires type checking may not be available statically. We therefore postpone our analysis until the program has stabilised. Hence, during initialisation, the full power of Python can be used, including metaclasses, `eval` and dynamic code loading. The entry point to the type checking mechanism is the analysis in the class `Analyser`. This takes a `callable` object

such as the main function, and an integer truncation level N. By starting the analysis once the environment has stabilised, we obtain a more accurate type analysis.

Analysis. As our system is built upon a static control flow analysis, we need an implementation for this. Our type inference and its correctness are independent of the particular choice, as long as the analysis returns an over-approximation of the actual control flow at runtime. Similarly, our implementation is parametric in the implementation of the control flow analysis. As a proof of concept, we use a simplified version of $\mathrm{next}(.)$ and $\mathrm{prev}(.)$ in which we assume that all function definitions are declared once.

The `Analyser` first constructs the truncated CFG and then iterates over all nodes in order to calculate the p and f-types for the accumulator. This triggers recursive computations for other variables. All runtime type errors arise due to an ill-typed accumulator value and therefore, to just *identify* the type errors, the type of the accumulator is sufficient. However, in order to *preempt* type errors, the types of all other variables that feed into the computation are necessary as well. All type calculations are cached during the iteration across the CFG so that p and f types for all necessary variables in all states are established. An implementation of $\mathrm{specialise}$ as in Fig. 8 is then used to transform the program so that it implements preemptive type checking. This is carried out on the given callable object, such as `main`, and all other functions it may call. Then, calling the specialised `main` activates the preemption to catch any runtime type errors.

The `Analyser` class can also statically issue messages explaining the potential type failures in the given function. This includes partial stack traces with the expected type errors. Along with the expected and actual types. This information is derived from our internal representation of execution points and types.

Full Evaluation Stacks. The Python virtual machine is a stack based machine. The *evaluation stack* serves as working memory and is read and manipulated by a large portion of bytecode instructions. For example, load operations push a single element on to the stack while store operations pop a single element from it. In general, bytecode instructions may displace stack elements by a number of positions, which can be determined statically. Although the theory outlined above uses a single element evaluation stack (i.e. the accumulator), the implementation *already* supports the full stack model. We adjust the inference rules above to cater for a full stack machine simply by statically calculating how much the stack is shifted for every instruction and factoring that in to identify the particular variables that we need to analyse in the type inference rules.

6 Evaluation

We tested our implementation on a number of Python benchmarks and examples from the Computer Language Benchmarks Game [2]. 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. 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.

The largest benchmark that we tested is `meteor-contest`, where the C++ version of this is 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 witout 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 never 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 a maximum execution point depth of 4 yields a control flow graph with over 30k nodes. In this case, it took under half an hour to analyse the program and 15 seconds to transform it on a standard workstation. The same program however takes under ten seconds to analyse and transform when the maximum 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.[1] Our implementation statically produces warnings that corroborate the answer given to this question by Python developers.

7 Related Work

Combinations of static and dynamic typing have been proposed, which enable statically typed code to interact with dynamically typed code. The initial work focused on increasing the degree of dynamic typing in statically typed languages. Abadi et al. [3] introduced the type Dyn to model finite disjoint unions or subclassing in object-oriented languages. The use of dynamic types is constrained (values of type Dyn can only be used in a typecase-construct) and therefore casting is made explicit. In Gradual typing [18], 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 can support type inferance [19,24] and can be applied to object oriented languages

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

[24]. Intermediaries between Dyn and static types are introduced by Flanagan [9] (*Hybrid types*) and by Wrigstad [23] (*like* types). The type systems discussed here do not perform any type error preemption.

As in preemptive type checking, soft typing [7] 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 [7] did not handle assignments, so there is no notion of preemption. Soft typing was extended to support Scheme [22] 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 Fig. 1. Soft typing has also been applied to Python [15] and Erlang [13]. In the latter, 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 [5]. Since preemptive type checking does not affect the semantics of μPython in runtime executions that terminate without raising type errors (Section 4) and no type annotations are required, our type system meets this definition.

We now look at static type inference mechanisms, which turn dynamically typed languages into statically typed subsets. Strongtalk [6] is a subset of Smalltalk with features such as polymorphic signatures, protocol based inheritance, generics and parametric polymorphism. This language also supports the typecase construct. This work however does not define a formal type system or describe how omitted type annotations are treated. Felleisen and Tobin-Hochstadt [21] propose the notion of *occurrence typing* for implementing a statically typed version of Scheme. A translation of the simple example in Fig. 1 is statically rejected by this system. Similarly, statically typed subsets of Python [4] and Ruby [10] 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 [20] 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.

Lastly, we look at control flow analysis for dynamically typed languages. k-CFA [16] is an algorithm to perform inter-procedural control flow analysis on Scheme by abstract interpretation. Unfortunately, some variants of k-CFA are intractable [17].

8 Conclusions and Future Work

In this paper, we introduce a new method for type checking dynamically typed programs that combines elements of both static and dynamic type checking. It is described as *preemptive type checking* since the type checking happens much earlier than in dynamic typing. Preemptive type checking tries to detect type errors as early as possible and guarantees that any program that can run to completion under dynamic typing without raising a type error will also work with preemptive type checking. We also evaluate an implementation for a subset of Python.

In the future, we plan to add features such as classes and objects, possibly using structural types [14]. This can be complemented with a control flow analysis algorithm such as k-CFA [16]. We also intend to investigate how preemptive type checking can

be applied to other popular dynamically typed languages such as JavaScript. Finally, we intend to support metaprogramming by going back and forth between type checking and runtime whenever a new part of the running program is generated.

References

1. Mars Climate Orbiter Mishap Investigation Board Phase I Report. NASA (1999)
2. The Computer Language Benchmarks Game, `http://shootout.alioth.debian.org/`
3. Abadi, M., Cardelli, L., Pierce, B., Plotkin, G.: Dynamic typing in a statically typed language. ACM Transactions on Programming Languages and Systems 13(2), 237–268 (1991)
4. Ancona, D., Ancona, M., Cuni, A., Matsakis, N.: RPython: A step towards reconciling dynamically and statically typed OO languages. In: DLS, pp. 53–64 (2007)
5. Bracha, G.: Pluggable type systems. Revival of Dynamic Languages (2004)
6. Bracha, G., Griswold, D.: Strongtalk: Typechecking Smalltalk in a production environment. In: OOPSLA, pp. 215–230 (1993)
7. Cartwright, R., Fagan, M.: Soft typing. In: PLDI, pp. 278–292 (1991)
8. Findler, R., Felleisen, M.: Contracts for higher-order functions. In: ICFP, pp. 48–59 (2002)
9. Flanagan, C.: Hybrid type checking. In: POPL, pp. 245–256 (2006)
10. Furr, M., An, J., Foster, J., Hicks, M.: Static type inference for Ruby. In: Symposium on Applied Computing, pp. 1859–1866 (2009)
11. Grech, N.: Preemptive Type Checking in Dynamically Typed Languages. PhD thesis, University of Southampton (submitted)
12. Might, M., Smaragdakis, Y., Horn, D.: Resolving and Exploiting the k-CFA Paradox. In: PLDI, pp. 305–315 (2010)
13. Nyström, S.: A soft-typing system for Erlang. In: Erlang Workshop, pp. 56–71 (2003)
14. Pierce, B.: Nominal and Structural Type Systems. In: Types and Programming Languages, ch. 19, pp. 247–264 (2002)
15. Salib, M.: Starkiller: A Static Type Inferencer and Compiler for Python. Thesis, MIT (2004)
16. Shivers, O.: Control-flow analysis in Scheme. In: PLDI, pp. 164–174 (1988)
17. Shivers, O.: Higher-order control-flow analysis in retrospect: lessons learned, lessons abandoned. ACM SIGPLAN Notices 39(4), 257–269 (2004)
18. Siek, J., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming Workshop (2006)
19. Siek, J., Vachharajani, M.: Gradual typing with unification-based inference. In: DLS (2008)
20. Heidegger, P., Thiemann, P.: Recency Types for Dynamically-Typed, Object-Based Languages. In: FOOL (2009)
21. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed scheme. In: POPL, pp. 395–406 (2008)
22. Wright, A., Cartwright, R.: A practical soft type system for scheme. ACM Transactions on Programming Languages and Systems 19, 87–152 (1997)
23. Wrigstad, T., Nardelli, F., Lebresne, S., Östlund, J., Vitek, J.: Integrating typed and untyped code in a scripting language. In: POPL, pp. 377–388 (2010)
24. Rastogi, A., Chaudhuri, A., Hosmer, B.: The Ins and Outs of gradual type inference. In: POPL, pp. 481–494 (2012)
25. TIOBE Programming Community Index (2013), `www.tiobe.com`