Cost versus Precision for Approximate Typing for Python

Levin Fritz

Jurriaan Hage

Technical Report UU-CS-2014-017
June 2014

Department of Information and Computing Sciences

Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

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

Cost versus Precision for Approximate Typing for Python

Jurriaan Hage
Universiteit Utrecht
levinfritz@gmail.com J.Hage@uu.nl

Abstract

In this paper we describe a variation of monotone frameworks that enables us to perform approximate typing of Python, in particular for dealing with some of its more dynamic features such as first-class functions and Python's dynamic class system. We additionally introduce a substantial number of variants of the basic analysis in order to experimentally discover which configurations attain the best balance of cost and precision. For example, the analysis allows us to be selectively flow-insensitive for certain classes of identifiers, and the amount of call-site context is configurable. Results of our evaluation include that adding call-site sensitivity and parameterized types has little effect on precision; in terms of speed call-site sensitivity is very costly. On the other hand, flow-insensitive treatment of module scope identifiers has a strongly positive effect, often both in terms of precision and speed.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – *soft typing, Python*

General Terms measurement

Keywords approximate typing, Python, data-flow analysis, abstract interpretation, monotone frameworks, performance, precision, call-site sensitivity

1. Introduction

Our goal in this paper is to arrive at an approximation of the types of values that variables, methods and other bindings in a Python [van Rossum and Drake, 2011] program may take. The results of our work are intended for, e.g., inclusion in an IDE for Python programming where we want to assist a programmer by providing a small enough selection of identifiers that fit the context; in this particular case there is no reason to insist on either soundness or completeness, but the results should be attained reasonably fast (at most a few seconds), and be reasonably precise.

The technique we employ to arrive at the types is abstract interpretation by means of a variant of the (well-known) monotone frameworks [Nielson et al., 2005]. The extensions we make to monotone frameworks are twofold: First, we have added facilities for dealing with the highly dynamic nature of some aspects of Python, including its dynamic class system and anonymous functions (Section 2). Second, our formulation and associated implementation are organised in a way that allows us to experiment with a substantial number of variants (Section 4): a parameterized widening operator for tuning the size of the type lattice, flow-insensitive analyses for certain kinds of identifiers (module scope identifier, class types, instance types), call-site sensitive analysis at varying levels of precision, parameterized datatypes, and the inclusion of manually specified types.

In order to establish which variants attain the best balance of cost and precision, we have applied our implementation in all its variants to five Python applications (Section 5). Some of the results we have obtained are the following (Section 6): Adding explicit type information for top-level identifiers increases precision, but hardly affects speed. Call-site sensitivity is very costly, but hardly improves precision. The use of parameterized datatypes during analysis, e.g., $list\langle int\rangle$ instead of *list*, improves precision substantially in only one case. Flow-insensitivity for module-scope identifiers (i.e., keeping only one set of types for a module scope identifier, instead of one or two sets for each statement in the program) leads to substantial improvements in speed *and* precision; flow-insensitivity for class and instance types improves precision somewhat, but at a rather high cost. Our experiments also indicate the "best" settings for the parameterized widening operator that is employed to both tune precision and guarantee termination.

To summarise, we offer the following contributions:

- We describe an extension of monotone frameworks that is particularly suited for dealing with the dynamic aspects of Python.
- We have implemented this extension in a way that allows to tune the precision of the analysis to a large extent, and the implementation (in Haskell) is publicly available from `http://www.cs.uu.nl/wiki/bin/view/Hage/Resources`.
- We applied our implementation with various levels of precision to five Python applications to, e.g., discover when additional precision in the analysis does not lead to more precise results.

Our work was implemented for Python 3.2 (Feb 2011), but should also work for 3.0 and 3.1. Python has an imperative, object-oriented core with features from functional and scripting languages, and is widely used for web development and all forms of scripting. Our implementation can deal

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

$$\begin{tikzcd}
{} \arrow[dr, bend left] & & & \\
 & 1 \arrow[r] \arrow[dd] & 2 \arrow[r] \arrow[d] & 3 \arrow[ll, bend right] \\
 & & 4 \arrow[r] \arrow[d] & \boxed{5} \\
 & \boxed{7} & 6 \arrow[uul] &
\end{tikzcd}$$

Figure 1: *While* loop with *continue* and *break* statements, and its associated control-flow graph.

with many of Python's features such as higher-order functions and classes. Our only omission is that we do not fully support exceptions and generators, and we may loose soundness in the presence of these constructs. Repairing these issues is possible, but the cost of implementation is high: in the case of exceptions we shall have to find out which exceptions may be generated by which statements, in order to add these flows to the control-flow graph. Assuming that exceptions indeed occur rarely, we decided to simply omit such edges from the graph. Generators, on the other hand, enable a stream-based way of programming, in which functions *yield* their results piece by piece to the caller. It is a feature that is rarely used, and its implementation is not trivial, which is we omitted it. An unavoidable source of unsoundness is when the programmer provides explicit types for identifiers from, e.g., Python libraries, that are not sound. Since this typically occurs for identifiers for which we have no access to the source code, there is no way that this can be discovered by our implementation.

Section 3 gives some details of the analysis itself (but given the size of the language, many details can only be found in [Fritz, 2011]), Section 7 surveys related work, and Section 8 concludes and provides pointers for future work.

2. Extending monotone frameworks

Our analysis and its variants are implemented as a data flow analysis formulated as a monotone framework with some additions to allow us to deal with some of the features of Python. Before detailing approximate typing for Python, we first explain these necessary additions.

As usual, the first step in data-flow analysis is to construct the control-flow graph of a (Python) program. This is less easy than it may seem: we have to account for control structures such as break and continue, deal with control-flow internal to an expression, short-circuiting boolean operators, loop else-clauses, list, set and dictionary comprehensions and Python's with-statement (akin to a try-finally construct). For reasons of space, we give a single example that includes a loop, break, continue and a loop else-clause in Figure 1 (see Section 4.3 of [Fritz, 2011] for the remaining cases). Here the nodes correspond to blocks in the program, the initial node is 1, and both 5 and 7 are final nodes.

Having constructed a control-flow graph, we can construct a monotone framework $(L, \mathcal{F}, F, E, \iota, \lambda l.f_l)$ where L denotes the lattice from which analysis results are taken, $\mathcal{F}$ is a set of monotone functions from which transfer functions are selected, F is the flow of the program, E contains the vertices from which information flows (the extremal vertices), ι is the extremal analysis value that is used to initialize the analysis results for extremal vertices, and the function $\lambda l.f_l$ selects for a given a label l the transfer function for the block(s) labelled with l (see [Nielson et al., 2005] for details).

In order to handle Python code properly and to support different variants of the analysis, the embellished monotone frameworks as described in [Nielson et al., 2005] are extended in two ways.

First, because Python supports first-class functions and late binding, it is less than trivial to infer which function or method is called by a particular invocation. Therefore we add a facility for adding edges dynamically to the control-flow graph to reflect these dynamics. Our algorithm assigns a unique identifier to each function in the program under analysis and these identifiers are included in the types inferred for functions, so which function a call can refer to can then become apparent during analysis. To be able to use this information, the monotone framework and worklist algorithm are extended so that edges for function calls can be added to the control flow graph during execution of the worklist algorithm.

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

Second, because we also want to experiment with partially flow-insensitive variants of the analysis, and the formulation of [Nielson et al., 2005] implies a flow-sensitive analysis (analysis results are indexed by statement/block label), we further extend the worklist algorithm employed to solve the monotone frameworks, allowing it to selectively treat identifiers flow-insensitively. Concretely, this means that these identifiers are associated with one set of types, instead of a pair of such sets for each program label. This also implies that once we find that such a set changes, the worklist algorithm must ensure this change is signalled to all parts of the program that employ the given identifier.

Extended worklist algorithm

The extended worklist algorithm is given in Algorithm 2; it is a variation of the Maximal Fixed Point algorithm given in [Nielson et al., 2005, Chapter 2].

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

```
Initialization:
    $A_\circ[l] \leftarrow \begin{cases} \iota & \text{for } l \in E \\ \bot & \text{otherwise} \end{cases}$
    $A_\bullet[l] \leftarrow \bot$
    $g \leftarrow \iota_g$
    $W \leftarrow N$
    $G \leftarrow \emptyset$
Iteration:
    while $W$ not empty do
        $i \leftarrow \text{head}(W)$
        $W \leftarrow \text{tail}(W)$
        if $i = l \in N$ then
            $(t, d, e', g', u) \leftarrow f_l(A_\circ[l])$
            if $g' \not\sqsubseteq g$ then
                $g \leftarrow g'$
                for all $l \in G$ do
                    $W \leftarrow l : W$
            if $u$ then
                $G \leftarrow \{l\} \cup G$
            if $e' \not\sqsubseteq A_\bullet[l]$ then
                $A_\bullet[l] \leftarrow e'$
                for all $l'$ with $(l, l') \in F$ do
                    $W \leftarrow (l, l') : W$
            $F' \leftarrow \bigcup\{\{(l_c, l_n), (l_x, l_r)\} \mid (l_n, l_x) \leftarrow \Lambda[f], f \leftarrow t\}$
            if $d$ then
                $F' \leftarrow F' \cup (l_c, l_r)$
            if $F' \not\subseteq F$ then
                for all $e \in F' \setminus F$ do
                    $W \leftarrow e : W$
                $F \leftarrow F' \cup F$
        else if $i = (l, l')$ then
            if $A_\bullet[l] \not\sqsubseteq A_\circ[l']$ then
                $A_\circ[l'] \leftarrow A_\circ[l'] \sqcup A_\bullet[l]$
                $W \leftarrow l' : W$
```

Algorithm 2: Extended worklist algorithm

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

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

3. Type inference for Python

Our analysis aims to infer the types for variables in Python source code. However, this formulation is not quite right, because in Python there is really no notion of "types of variables". A more precise statement of the goal of the analysis is: *for each variable in a Python program, what are the types of the values it may be bound to when the program is executed*. The term "variable" includes parameters of functions and methods.

The basic type inference method is a data flow analysis expressed as a monotone framework and solved by the worklist algorithm (see Section 2). It is flow-sensitive, context-insensitive and path-insensitive. (A path-sensitive analysis computes different results depending on predicates at nodes where the flow of control diverges. For example, a path-sensitive analysis could infer that in the body of the statement *if x is None: . . .*, variable *x* has value *None*.)

Figure 3 defines a type lattice for Python. In the notation used on the right, $\{v\}$ stands for a set of zero or more elements of the form v, and $[v]$ stands for an ordered sequence of zero or more elements of the form v.

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

Figure 3: Basic type lattice for Python.

The type assigned to a variable is called a *union type*: it is either the set of types of the values that the variable may be bound to at runtime, or $\top$, which represents the set of all types. *Value types* model types of values at runtime. The analysis distinguishes five kinds of value types. Built-in types are built into Python and rules to deal with them are built into the analysis. A function type refers to a function in the source code; these are assigned unique labels to avoid name clashes. The types for classes defined in the code under analysis and instances of these are more interesting. Classes and objects are very dynamic in Python: at runtime, new classes can be created and attributes can be added to and removed from classes and objects. Classes may also have multiple superclasses. This is captured by the definitions in Figure 3: a class type contains a list of superclasses and a mapping from names to types for class attributes; an instance type contains the instance's class and a mapping for instance attributes. Like functions, classes are also assigned unique labels.

Join operator

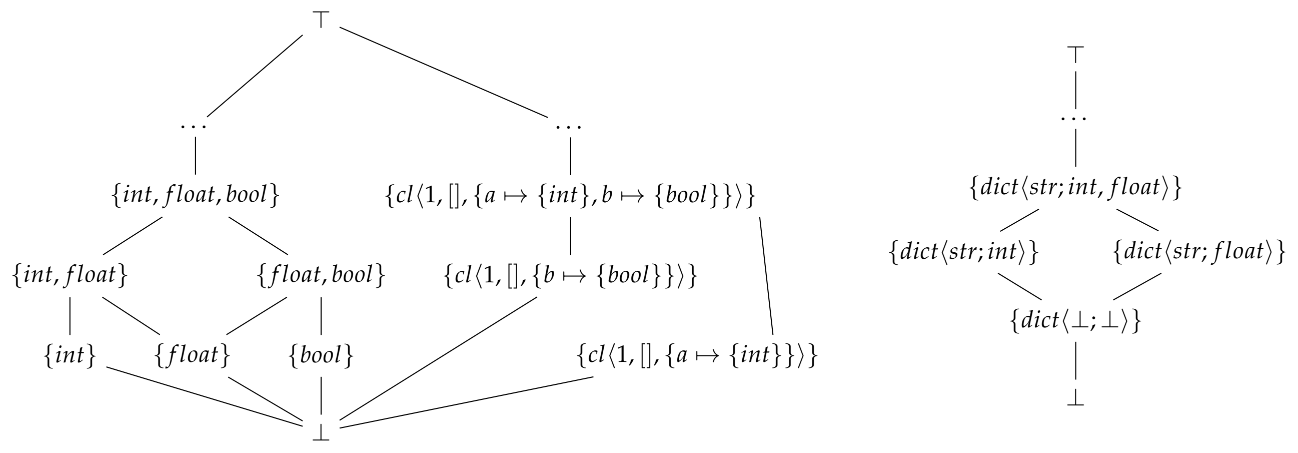
The set *UTy* becomes a join-semilattice by defining a join operator $\sqcup$ and identifying a bottom element $\bot$. For brevity, we call *UTy* a lattice in the following.

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

Figure 3(left) shows some of the elements of the lattice in the form of a diagram. In the diagram, $a \sqsubseteq b$ is expressed as an edge from a to b with a being below b.

Widening operator

To ensure termination of the fixed point computation, the basic analysis uses a widening operator $\nabla_{n,m,o}$ (see [Nielson et al., 2005, Section 4.2.1]), which is parameterized with three numbers: $n \in \mathbf{N}$ is the maximum cardinality of a set of types, $m \in \mathbf{N}$ is the maximum number of attributes of a class or instance and $o \in \mathbf{N}$ is the maximum nesting depth. Intuitively, the nesting depth of a type is the depth of the abstract syntax tree that is implicitly defined by the definitions in Figure 3, and extended for parameterized types in Section 4.1.



$\top$
$\dots$
$\{int, float, bool\}$
$\{int, float\}$
$\{float, bool\}$
$\{int\}$
$\{float\}$
$\{bool\}$
$\{cl\langle 1, [], \{a \mapsto \{int\}, b \mapsto \{bool\}\}\rangle\}$
$\{cl\langle 1, [], \{b \mapsto \{bool\}\}\rangle\}$
$\{cl\langle 1, [], \{a \mapsto \{int\}\}\rangle\}$
$\bot$
$\{dict\langle str; int, float\rangle\}$
$\{dict\langle str; int\rangle\}$
$\{dict\langle str; float\rangle\}$
$\{dict\langle \bot; \bot\rangle\}$

Figure 4: Part of the basic type lattice (left) and an part that includes parameterized datatypes (right).

If a union type exceeds one of the limits, it is replaced by $\top$. The following example illustrates the effect of the n parameter:

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

The next example shows the effect of the m parameter. Computing

$$\{cl\langle 1, [], a \mapsto \{int\}\rangle\} \nabla_{2,2,2} \{cl\langle 1, [], b \mapsto \{str\}\rangle\}$$

gives

$$\{cl\langle 1, [], a \mapsto \{int\}, b \mapsto \{str\}\rangle\},$$

but

$$\{cl\langle 1, [], a \mapsto \{int\}\rangle\} \nabla_{2,2,2} \{cl\langle 1, [], b \mapsto \{str\}, c \mapsto \{bool\}\rangle\}$$

results in $\top$. To show the effect of the o parameter:

$$\{cl\langle 1, [], a \mapsto \{float, int\}\rangle\} \nabla_{2,2,3} \bot$$

equals $\{cl\langle 1, [], a \mapsto \{float, int\}\rangle\}$, and

$$\{cl\langle 1, [], a \mapsto \{float, int\}\rangle\} \nabla_{2,2,2} \bot$$

gives $\{cl\langle 1, [], a \mapsto \top\rangle\}$. The types $float$ and int are at nesting depth 3, so for $\nabla_{2,2,2}$ they are too deeply nested, and the union type is replaced with $\top$.

The value computed for each program point is a mapping from variables to *UTy*, called a *map lattice*. The join operator for this lattice is defined as follows: $a \sqcup b$ contains all mappings present in either a or b; if a mapping is present in both, its values are combined by the join operator of the *UTy* lattice.

The analysis

Analysis intuitively proceeds by pushing sets of types over the control-flow edges. The control-flow graph itself is built by gluing together the control-flow graphs of the modules that make up the application or library. The entry point for a single module is that module's first statement. These together form the entry points of the application/library as a whole. This also allows us to analyze libraries that typically do not have a single point of entry.

An important source of type information comes from various kinds of assignments in Python. As expected the type of an expression is computed based on the types of the variables in the expression. The rules for these can be quite involved (see [van Rossum and Drake, 2011, Chapter 5]). Python also supports assignments such as

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

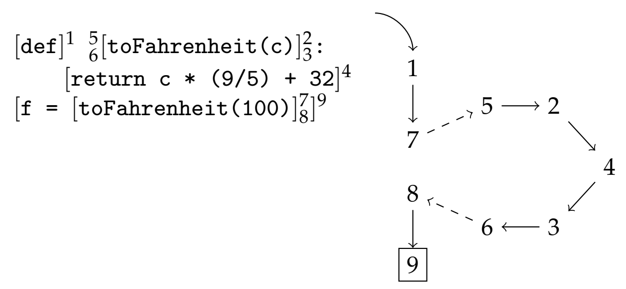
This assigns the first element of the list to x, the second to y and all further elements to z. For a star target, such as **z* in the example, the analysis deletes all non-sequence types, and assigns these to the variable. The other targets, x and y, are assigned $\top$ by the basic analysis.

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

For a subscription, such as *x[1]*, the basic analysis removes the sequence types (strings, tuples, lists) from the types of x. Because sequence types are not yet parameterized, the type of *x[1]* will be $\top$. Better support for these is part of the analysis variant for parameterized datatypes (Section 4.1).

Functions are handled as usual, except that because of the dynamic nature of Python, the edges from the call to the function entry points, and the edges from the functions exit points to the return point in the program are added during analysis, whenever the analysis finds that a call can target a given function (and generally, there may be more than one such function for a given call). Parameter passing is essentially the same as performing assignments. Special attention must be given at this point to local variables that come into scope, and variables that leave the scope during the call. This is why for function names we have two additional labels, s_n and s_x. In the former we introduce the variables that come into scope in the map lattice, and at the latter, variables that leave the scope are removed.

To illustrate the analysis, consider the code in Figure 5, for which the analysis correctly infers the type *float* for *f*. During analysis, the effect of the various transfer functions, in the order implied by the control flow graph, is as follows:


$[\texttt{def}]^1\ {}^5_6[\texttt{toFahrenheit(c)}]^2_3\texttt{:}$

$\quad[\texttt{return c * (9/5) + 32}]^4$

$[\texttt{f = }[\texttt{toFahrenheit(100)}]^7_8]^9$

$$\begin{tikzcd}
 & 1 \arrow[d] & & & \\
 & 7 \arrow[r, dashed] & 5 \arrow[r] & 2 \arrow[dr] & \\
 & & & & 4 \arrow[dl] \\
 & 8 \arrow[d] & 6 \arrow[l, dashed] & 3 \arrow[l] & \\
 & \boxed{9} & & &
\end{tikzcd}$$

Figure 5: Control flow graph for function definition and function call.

- For node 1: the type of variable *toFahrenheit* is set to $\{f_1\}$, 1 being the identifier assigned to the function.
- For node 7 (l_c): the argument variable α_0 is set to $\{int\}$, which is the type of the expression 100.
- For node 5 (s_n): variable c is set to $\perp$.
- For node 2 (l_n): the type for variable c is set to that of variable α_0, then α_0 is removed from the lattice.
- For node 4: using the current type lattice, the type of the expression *c * (9/5) + 32* is determined to be $\{float\}$ (the division *9/5* yields a floating-point value); this is assigned as the type of variable ρ that is allocated to store the types of the return value.
- For node 3 (l_x): the transfer function checks if ρ is set, and since it is, does not change anything.
- For node 6 (s_x): the transfer function removes variable c.
- For node 8 (l_r): the type ρ is copied to the generated variable ι_1 (which represents the types of the call expression), and ρ is removed from the lattice.
- For node 9: this is a simple assignment; it sets the type of f to those in ι_1.

4. Analysis variants

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

4.1 Parameterized datatypes

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

The solution is to introduce parameterized datatypes such as $list\langle int\rangle$ meaning "list containing values of type *int*", or, more concisely, "list of int".

Extended type lattice

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

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

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

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

To avoid visual clutter we write $list\langle int, float\rangle$ instead of $list\langle\{int, float\}\rangle$. Where a type has multiple parameters, they are separated by semicolons, while the elements of a union type are separated by commas, so that no ambiguity arises.

The join operator treats parameterized types specially, similar to the way it treats class and instance types. For example, if union types a and b contain types $list\langle u_a\rangle$ and $list\langle u_b\rangle$, respectively, then $a \sqcup b$ contains only one parameterized list type $list\langle u_a \sqcup u_b\rangle$. Set and dictionary types are handled analogously. For parameterized tuple types, only those of the same length are combined. Figure 3(right) shows some of the elements of the lattice with parameterized datatypes.

The widening operator $\nabla_{n,m,o}$ described in Section 3 can be used unchanged for parameterized datatypes. Its parameter n was defined to limit the size of sets of types, so that it also applies to type parameters, as in the following examples:

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

Use of parameterized datatypes

Parameterized datatypes are used in a number of circumstances to improve analysis results. One, list, set, dictionary and tuple literals are assigned parameterized types. For example, the expression *(1, 1.5)* is assigned type $tuple\langle int; float\rangle$. Two, expressions that involve a subscription or slicing make use of parameterized types. For example, assuming type $\{dict\langle int; str\rangle\}$ has been inferred for variable a, type $\{str\}$ is inferred for the expression *a[1]*. Three, when the target of an assignment is a subscription or slicing, the parameters of parameterized types are modified correctly. Four, parameterized types are assigned to the results of list, set and dictionary comprehensions. Finally, in *for* loops, parameterized types are used to assign the most precise type possible to the loop variable. The following code illustrates several of these uses:

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

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

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

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

4.2 Context-sensitive analysis

The second analysis variant implements a context-sensitive analysis as described in [Nielson et al., 2005, Section 2.5]. The type inference method uses call strings containing the labels of function calls as the context Δ. In other words, we analyze call-site sensitively.

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

The worklist algorithm tags the edges in the control flow graph according to their function: edges in the monotone framework are tagged as "regular", while those added for function calls are tagged with "call" and "return". Each kind is then treated accordingly.

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

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

To ensure termination, the worklist algorithm takes a parameter k, which is the maximum length of a call string. This parameter can also be used to trade off precision for speed: depending on the patterns of function calls in the program code, a small value of k can lead to imprecise results, but it also reduces the number of separate results that are maintained by the analysis, which should make it faster (a thorough discussion of this aspect is outside the scope of this paper, but some researchers have observed the opposite to happen, e.g., [Smaragdakis et al., 2011]). In the modified worklist algorithm, a context-insensitive analysis is treated as a context-sensitive analysis with $k = 0$.

4.3 Flow-insensitive analysis

Data flow analysis is basically flow-sensitive, but in some cases flow-insensitive analysis may be more logical. The analysis therefore includes three variants that use the extension for flow-insensitive analysis described in Section 2, which adds a "global value" with flow-insensitive results to the worklist algorithm's state. In each of these variants, the types a for certain class of variables are stored globally, while for other types it still uses context and effect values.

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

Flow-insensitive analysis for module-scope variables

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

Flow-insensitive analysis for class types

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

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

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

Writing class reference types as $cl\langle l\rangle$, the type lattice defined in Figure 3 can be adapted for this variant by adding an alternative to the definition of *ClsTy*:

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

Flow-insensitive analysis for instance types

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

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

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

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

4.4 Manually specified types

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

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

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

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

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

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

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

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

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

Polymorphic function types

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

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

can be provided with a suitable type by the following specification:

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

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

5. Evaluation method

We have implemented all the variants discussed in Section 4 in Haskell (obtainable from `http://www.cs.uu.nl/wiki/bin/view/Hage/Resources`) and applied them to a number of real-world Python programs. The main focus of our study is to discover the right balance of cost and precision. In particular, we wanted to discover when increased precision stops paying off in terms of results, and how increased precision affects run-time performance. We first describe the method used to measure precision and speed and the projects the analysis was applied to before presenting the results of the experiments. The raw data of the experiments can be found in [Fritz, 2011].

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

Measuring precision

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

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

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

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

In the next step, the types in this list are classified in two groups: $\bot$ and $\top$ types are classified as "not useful", all others are classified as "useful". The score is calculated as the ratio of "useful" types to all types in the list. Note that in an IDE that lists all the possible identifiers that may fit a particular context, variables that were assigned type $\top$ will be suggested by the IDE for any context. If the precision of an analysis is so low that many variables are assigned to $\top$ by the analysis (while another variant may find that a smaller set will suffice), then together these will drown out those identifiers that actually fit the context. Therefore, for our aims, our score is a reasonable approximation that avoids a detailed manual study of the Python applications themselves.

This method also gives appropriate weights to types of identifiers for which flow-insensitive analysis is used: because the analysis infers only one type for each of these, but one type per program point for others, they might have a disproportionally small influence on the results of a simpler algorithm.

Measuring speed

To measure speed, the implementation records the time just before and just after running the analysis, and prints the difference in microseconds (μs). The time measured is CPU time (the amount of time that the program has run on the

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

Table 6: Projects used as input for the evaluation.

CPU), which makes it less likely that the results are influenced by other factors such as the operating system's activities.

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

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

Example projects

In order to have a variety of Python source code represented in the experiments, five project were used, which are briefly presented here. These projects in particular were selected because they are compatible with Python 3.2 and they do not use modules written in C, which the analysis would not be able to process. The projects are ordered here from most to least self-contained, so the later ones are likely to be more problematic for type inference:

euler This codebase consists of solutions to five mathematical problems from the Project Euler website (`http://projecteuler.net/`) written by the first author. This is very straightforward code.

adventure The Adventure (`https://bitbucket.org/brandon/adventure/overview`) project is a fairly self-contained interactive program; having a text-based interface means it does not depend on a large GUI library.

bitstring The bitstring library (`http://code.google.com/p/python-bitstring/`) provides an interface in Python for the creation and manipulation of binary data.

feedparser The Universal Feed Parser (`http://feedparser.org/`) is a library for parsing RSS and Atom feeds implemented in Python. It consists of only two modules, but there is quite a bit of nested code to handle all the details of various versions of the two standards.

twitter The Python Twitter Tools (`http://mike.verdone.ca/twitter/`) contains a library, a command-line program, and an IRC bot to access the Twitter web site's public API.

Table 6 lists the number of modules and lines of code for each of the projects. In tables following Table 6, we write *adven* for *adventure*, *bits* for *bitstring*, and *feedp* for *feedparser*, in order to be able to format the result tables in a single column.

	euler	adven	bits	feedp	twitter	*mean*
precision	11.54	0.00	0.00	1.01	0.00	2.51
time	-9.63	0.64	31.58	3.88	2.55	5.80

Table 7: Effects of parameterized datatypes on experiment results.

6. Evaluation results

The results of the experiments, in the form printed by the implementation program, can be found in the Appendix of [Fritz, 2011]. This section presents various aspects of the results and uses them to evaluate the analysis variants, the influence of parameters and the general suitability of the method.

6.1 Variants

Tables 7–10 show the effect that the analysis variants have on precision and time. Results of each variant are compared here to the analysis with default parameters; the numbers shown are differences in percent. Thus, in the *precision* rows, positive numbers indicate better results; in the *time* rows, positive numbers indicate slower operation. The *mean* column contains the arithmetic mean of the values.

As can be seen in Table 7 and Table 8, parameterized datatypes and context-sensitive analysis did not improve the results by much. Parameterized datatypes did improve results significantly for the *euler* example code, but not for the larger projects. Context-sensitive analysis, which was tested for different values of the parameter k (maximum length of call strings) did not significantly improve the results in any of the cases.

The results of flow-insensitive analysis, shown in Table 9, are more encouraging. The first column of the table indicates what flow-insensitive analysis was used for. It contains the parameters passed to the implementation's command-line interface (see Appendix A.3 of [Fritz, 2011]): f means flow-insensitive analysis is used for module-scope variables, g means it is used for class types, and h means it is used for instance types. All seven possible combinations were used.

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

For the last set of experiments, types were specified manually for identifiers not in the code under analysis (from the standard library or third-party libraries). Because of time constraints, this was only done for the two smallest projects. Table 10 shows the results; not surprisingly, they indicate that specifying types manually improves precision at a moderate cost in runtimes.

6.2 Parameters for widening operator

Table 11 shows the results of different values for the parameters of the widening operator $\nabla_{n,m,o}$ (see Section 3), compared to the default settings $n = 3$, $m = 3$, $o = 20$. As may be expected, very small values for any of the parameters lead to poor precision and very large values lead to long runtimes. Other than that, the results indicate that the default values are actually good choices.

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

Table 8: Effects of context-sensitive analysis on experiment results.

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

Table 9: Effects of flow-insensitive analysis on experiment results.

	euler	adven
precision	70.16	39.01
time	1.81	6.81

Table 10: Effects of manually specified types on experiment results.

6.3 Evaluation

Table 12 shows the results in absolute numbers for the configuration that, according to the experiments, works best: using flow-insensitive analysis for module-scope variables, but none of the other variants. The analysis inferred a useful type for variables in the source code in between 45 and 91 percent of cases. When it was supplied with types for identifiers in external libraries, this number increased further (by 6 and 27 percent for the two projects used).

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

7. Related Work

For reasons of space we restrict our discussion here to work that involves a dynamically typed language. A more extensive discussion, and more references can be found in [Fritz, 2011].

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

Table 11: Effects of parameters of widening operator on experiment results.

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

Table 12: Precision and runtime (in *ms*) using flow-insensitive analysis for module-scope variables.

Cartwright and Fagan introduce the concept of *soft typing* as a way to combine the advantages of static and dynamic typing [Cartwright and Fagan, 1991]. A soft type system accepts all programs in a dynamically typed language and inserts dynamic checks in places where it cannot statically infer provably correct types. The programmer can then inspect the places where the checker failed and decide if the code should be changed. Flanagan introduces *hybrid type checking*, which is a synthesis of static typing and dynamic contract checking [Flanagan, 2006]. Dynamic contract checking can support more precise specifications than type checking, for example range checks and aliasing restrictions, but the propositions are not checked until runtime. With hybrid type checking, very precise interface specifications are possible. As in soft tpying, these are checked at compile time where possible and at runtime where necessary. *Gradual typing*, see, e.g., [Siek and Taha, 2007, Ina and Igarashi, 2011] allows mixing static and dynamic typing within one program: program elements with type annotations are checked statically, others are checked dynamically.

Agesen introduced the *Cartesian Product Algorithm* (CPA) in his PhD thesis [Agesen, 1996]. The algorithm infers types for programs written in the Self language, and is based on an algorithm introduced by Palsberg et al. that uses constraint-based analysis to determine types in object-oriented programs [Palsberg and Schwartzbach, 1991]. The algorithm assigns a *type variable* to each variable and expression and relates these by a set of constraints, that is solved by fixpoint iteration. Agesen duplicates the subgraph of variables and constraints for each monomorphic use of a polymorphic method.

Aycock developed a method for Python called "aggressive type inference" [Aycock, 2000]. It is flow-insensitive and does not use union types, assuming that most Python code does not make use of the dynamic features of Python's type system. Salib's master thesis describes a type inference method called "Starkiller" which is part of a Python-to-C++ compiler [Salib, 2004], loosely based on CPA. It can handle first-class functions and classes and objects, and supports parametric polymorphism as well as data polymorphism. Exceptions and generators are not supported, and the analysis is flow-insensitive. Cannon's master thesis presents a method to improve performance of Python programs which uses type inference and optional type annotations [Cannon, 2005]. The method for type inference is rather limited. Gorbovitski et al. describe a method for type inference for Python programs (called "precise type analysis") which they employ in their alias analysis when they construct the program's control flow graph [Gorbovitski et al., 2010]. Their algorithm is based on "abstract interpretation over a domain of precise types", in which types such as "bool true" and an "int between 1 and 10" can be represented. The analysis is flow-sensitive and context-sensitive. A downside is that it seem to be rather slow.

Pluquet et al. present a method for type inference for Smalltalk, which they describe as "extremely fast [...] and reasonably precise" [Pluquet et al., 2009]. Fast execution is achieved in two ways: the analysis is local in the sense that, to infer the type of a variable, it uses only information found in the methods of the class it is defined in and it uses a number of heuristics rather than a theoretical model of program execution. The authors validate the method by applying it to three Smalltalk applications. To evaluate precision, they monitored the execution of the programs, and recorded the types stored in each variable. The (incomplete) type information retrieved this way was then compared to the inferred types. Furr et al. present a system called Diamondback Ruby (DRuby), which includes union types, intersection types and parametric polymorphism [Furr et al., 2009]. Types are inferred by a method based on constraint-based analysis; however, not all code can be typed with this method. Intersection types cannot be inferred automatically, and type annotations must be used.

8. Conclusion

We have presented an extension to monotone frameworks to perform soft typing for Python, so that edges for function calls can be added to the control flow graph during the analysis. The analysis can deal with first-class functions and Python's dynamic class system. The method was implemented in a proof-of-concept implementation, which includes six variants that extend or modify the basic method. Our experimental evaluation of these variants show that the method can infer types with reasonable precision fairly quickly.

As future work, we may extend the monotone framework to support an object-sensitive analysis, instead of plain call-site sensitivity. For OO languages like Java, Object creation points seem to be a better choice for context than call-sites [Smaragdakis et al., 2011], but does this also hold for languages like Python? A limitation of our work is that we do not treat exceptions and generators as well as is possible. Finally, it may be useful to include more, and larger, applications in our case study.

References

O. Agesen. Concrete type inference: Delivering object-oriented applications. *Dissertation*, Jan 1996.

J. Aycock. Aggressive type inference. *8th International Python Conference*, Jan 2000.

B. Cannon. Localized type inference of atomic types in Python. *Master Thesis*, Jun 2005.

R. Cartwright and M. Fagan. Soft typing. *PLDI '91: 1991 Conference on Programming Language Design and Implementation*, pages 278–292, Jun 1991.

C. Flanagan. Hybrid type checking. In *POPL '06: Conference record of the 33rd ACM symposium on Principles of programming languages*, pages 245–256, New York, USA, 2006. ACM.

L. Fritz. Balancing cost and precision of approximate type inference in python, 2011. `http://www.cs.uu.nl/wiki/pub/Hage/MasterStudents/thesis.pdf`.

M. Furr, J. An, J. S. Foster, and M. Hicks. Static type inference for Ruby. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1859–1866, New York, USA, 2009. ACM.

M. Gorbovitski, Y. A. Liu, S. D. Stoller, T. Rothamel, and K. T. Tekle. Alias analysis for optimization of dynamic languages. *DLS '10: 6th Symposium on Dynamic Languages*, pages 27–42, Oct 2010.

L. Ina and A. Igarashi. Gradual typing for generics. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 609–624, New York, NY, USA, 2011. ACM.

F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Verlag, 2005.

J. Palsberg and M. Schwartzbach. Object-oriented type inference. *OOPSLA '91: 6th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 146–161, 1991.

F. Pluquet, A. Marot, and R. Wuyts. Fast type reconstruction for dynamically typed programming languages. *DLS '09: 2009 Symposium on Dynamic Languages*, pages 69–78, Oct 2009.

M. Salib. Starkiller: A static type inferencer and compiler for Python. *Master Thesis*, May 2004.

J. Siek and W. Taha. Gradual typing for objects. *ECOOP '07: 21st European Conference on Object-Oriented Programming*, Jul 2007.

Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM Symposium on Principles of programming languages*, POPL '11, pages 17–30, New York, USA, 2011. ACM.

G. van Rossum and F. L. Drake. The Python language reference, release 3.2, Mar 2011.