Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Balancing Cost and Precision of Approximate Type Inference in Python

Levin Fritz and **Jurriaan Hage**

Department of Information and Computing Sciences, Universiteit Utrecht
J.Hage@@uu.nl

January 17, 2017

Introduction

- Static typing: variables have types, checked at compile-time
  - C, Java, Haskell, ...
- Dynamic typing: values have types, checked at runtime
  - Scheme, Python, PHP, ...
- Type inference for statically typed languages:
  - determine the type of each variable
- Type inference for dynamically typed languages:
  - for each variable: which types can it refer to at runtime?
  - detecting errors
  - optimization
  - supporting tools

In this paper

- Method for approximate type inference of Python 3.2 programs
  - based on monotone frameworks
  - for interactive tools, eg, code completion in an editor
- Proof-of-concept implementation
- Experimental evaluation, balance of cost and precision
  - Focus on suitability for code completion

Python

Python:

- general-purpose, high-level programming language
- imperative, object-oriented
  - features from functional and scripting languages
- uses dynamic typing

Python Syntax

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

Indentation implies structure

Some language elements

- Mutable variables (both value and type)
- The usual (imperative) control structures
- Functions first class, anonymous functions
- Assignments, functions and class can introduce new variables
- Scope limited to enclosing block (module, function body, ...)
  - Can be overridden
- List, set and dictionary comprehensions

Object-oriented Programming

Example:

```
class C:
    x = 1
    def m(self, y):
        self.x = y
o = C()
o.m(10)
print(C.x, o.x)
```

Output:

```
1 10
```

Multiple inheritance, methods are attributes that are functions,

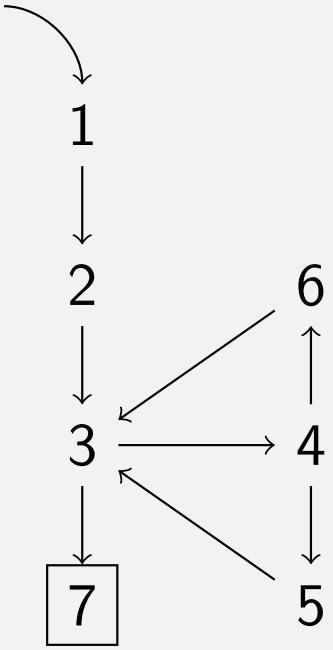
All is dynamic

- types of variables
- create classes at run-time
- add and delete attributes at run-time

Control flow graph (intraproc)

Analysis implemented on top of a program's control-flow graph:

```
[a = 42]^1
[b = 70]^2
while [b != 0]^3:
    if [a > b]^4:
        [a -= b]^5
    else:
        [b -= a]^6
[gcd = a]^7
```



Montone Framework: Overview

- Basic idea: propagate values through control flow graph.
- Two values for each program point l:
  - $A_\circ(l)$: context value
  - $A_\bullet(l)$: effect value
- Values are elements of a complete lattice L
  - join operator $\sqcup$, bottom elt. $\bot$ and top elt. $\top$
- $A_\bullet(l) = f_l(A_\circ(l))$ where f_l is the transfer function for l
- $A_\circ(l)$ is the join of effect values of l's direct predecessors
  - l' is a direct predecessor of l is there is an edge (l', l)

Solving Monotone Frameworks

- Fixpoint iteration
- Basic/naive algorithm:
  - Initialization:
    - $A_\circ(l) = \iota$ for program entry point
    - $A_\circ(l) = \bot (= \emptyset)$ for others
  - Iteration:
    - compute all $A_\bullet(l)$ using transfer functions
    - compute all $A_\circ(l)$ by propagating over CFG edges
    - repeat until there are no more changes
    - guaranteed to happen if paths from $\bot$ to $\top$ ($=$ all types) all finite (Ascending Chain Condition)

Interprocedural Data Flow Analysis

- Intraprocedural analysis: within procedures (functions)
- Interprocedural analysis:
  adds support for procedure definitions and calls
- Procedure definition:
  - entry and exit nodes l_n and l_x
  - Example:

    $[\texttt{def}]^{l_d}\ \ [\texttt{f(x)}]^{l_n}_{l_x}$:

    $\ldots$
- Procedure call:
  - call and return nodes l_c and l_r
  - Example:

    $[\texttt{f(1)}]^{l_c}_{l_r}$
- Add edges (l_c, l_n), (l_x, l_r).

Late Binding

- Late binding: which function a method call refers to is determined at runtime.
- With first-class functions and late binding, it's not obvious which function a call refers to.
- Type inference tracks functions.
- Edges for calls are added when solving the monotone framework.

Tracking Functions

- Each function is assigned a unique id.
  - included in type inferred for function
- Extend monotone framework with function table Λ: $\Lambda[f] = (l_n, l_x)$

Call Transfer Functions

- Two kinds of transfer function:
  - Simple transfer function: as before.
  - Call transfer function:
    - for l_c nodes
    - returns a set of function ids
- When solving the monotone framework, for call transfer functions:
  - look up each function id in Λ,
  - add edges for function calls to the CFG.
  - intuitively: edges in the CFG are part of the lattice, growing along with the sets of types

Late Binding: Example

$$[\texttt{def}]^1\ [\texttt{f(x)}]^2_3\texttt{:}$$
$$\quad [\texttt{return x + 1}]^4$$
$$[\texttt{f(1)}]^5_6$$
$$[\texttt{f(2)}]^7_8$$

Late Binding: Example

$[\texttt{def}]^1$ $[\texttt{f(x)}]^2_3$:

$\quad [\texttt{return x + 1}]^4$

$[\texttt{f(1)}]^5_6$

$[\texttt{f(2)}]^7_8$

Function table:

	l_n	l_x
1	2	3

Universiteit Utrecht

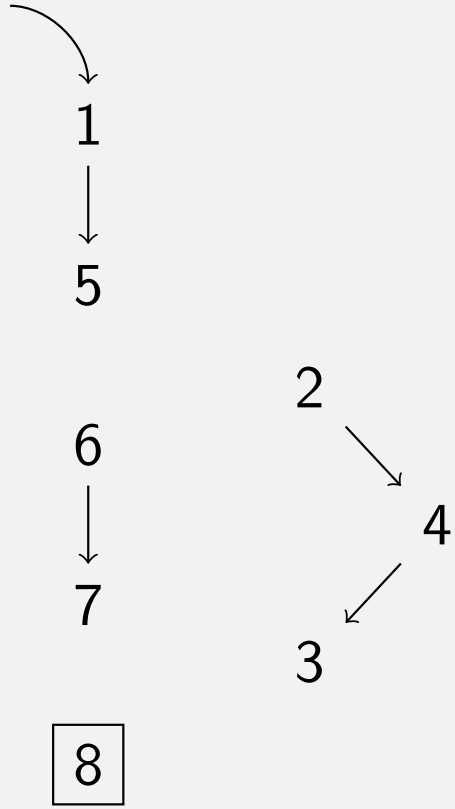
[Faculty of **Science**
Information and Computing Sciences]

Late Binding: Example

$[\texttt{def}]^1\ [\texttt{f(x)}]^2_3\texttt{:}$
$\quad [\texttt{return x + 1}]^4$
$[\texttt{f(1)}]^5_6$
$[\texttt{f(2)}]^7_8$

Function table:

	l_n	l_x
1	2	3



Late Binding: Example

$[\texttt{def}]^1$ $[\texttt{f(x)}]^2_3$:

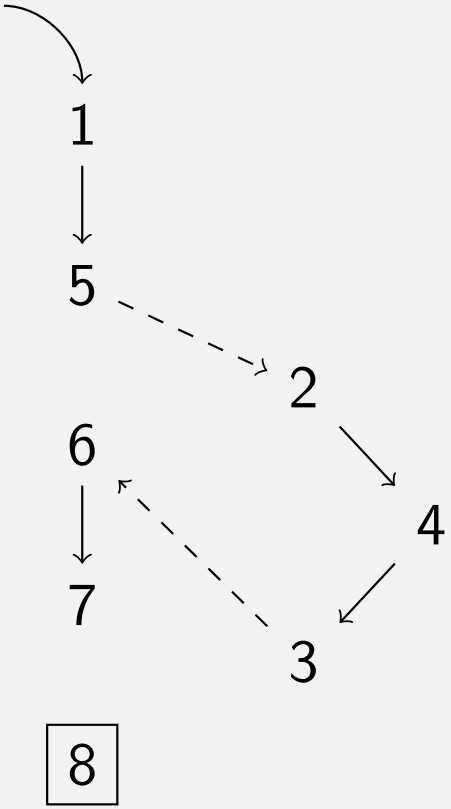
$\quad [\texttt{return x + 1}]^4$

$[\texttt{f(1)}]^5_6$

$[\texttt{f(2)}]^7_8$

Function table:

	l_n	l_x
1	2	3



Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Late Binding: Example

$[\texttt{def}]^1$ $[\texttt{f(x)}]^2_3$:

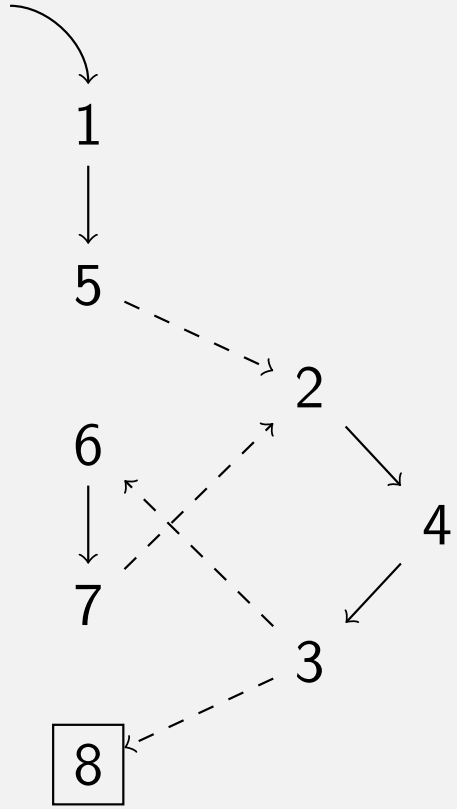
$\quad [\texttt{return x + 1}]^4$

$[\texttt{f(1)}]^5_6$

$[\texttt{f(2)}]^7_8$

Function table:

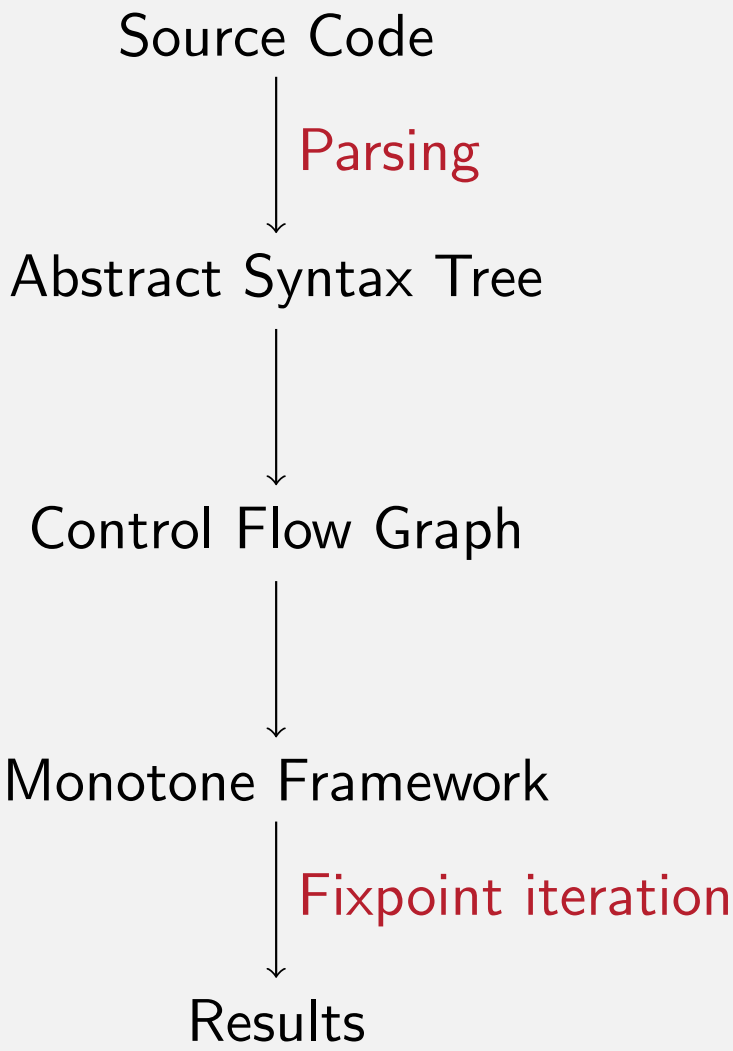
	l_n	l_x
1	2	3



Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Inferring Types: Overview



Type Lattice for Python

Types for variables in Python programs:

$u \in \mathsf{UTy}$ union types $u ::= \{v\} \mid \top$

Type Lattice for Python

Types for variables in Python programs:

$u \in \mathsf{UTy}$	union types	$u ::= \{v\} \mid \top$
$v \in \mathsf{ValTy}$	value types	$v ::= b \mid f \mid c \mid i$

Type Lattice for Python

Types for variables in Python programs:

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

Type Lattice for Python

Turning UTy into a lattice:

- $\bot = \emptyset$
- $a \sqcup b$ where neither is $\top$ is $a \cup b$, except
  - class types with the same class id are merged.
  - instance types with the same class id are merged.

Map Lattice

- Goal of the analysis: infer a type for each variable.
- Lattice used in monotone framework:
  mapping from variables to union types.
- Example:

```
x = 1
y = "a"
```

$$\{x \mapsto \{int\}, y \mapsto \{str\}\}$$

Analysis variants under evaluation

- Parameterized datatypes
- Context-sensitive analysis
- Flow-insensitive analysis

Parameterized Datatypes

- Example code:

```
list = [1, 2, 3]
sum = 0
for x in list:
    sum += x
```

- Basic analysis infers $\top$ for `sum`.

Parameterized Datatypes

- Example code:

```
list = [1, 2, 3]
sum = 0
for x in list:
    sum += x
```

- Basic analysis infers $\top$ for `sum`.
- To improve precision:
  Track types of contents of built-in collection types.
- Example: $list\langle int \rangle$ for list containing values of type int
- Extended type lattice:

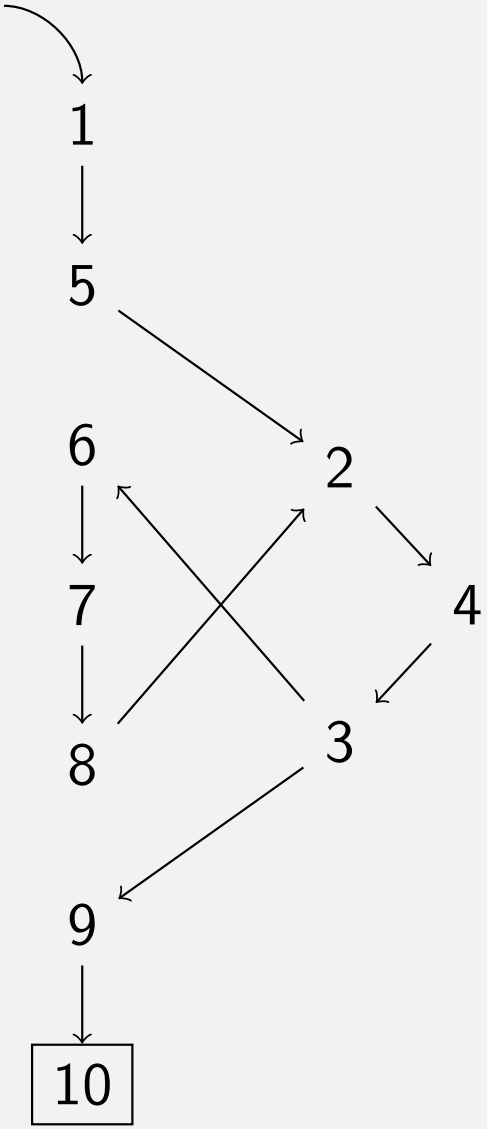
$$b ::= \cdots \mid list\langle u \rangle \mid set\langle u \rangle \mid dict\langle u; u \rangle \mid tuple\langle [u] \rangle$$

Context-sensitive Analysis

$$
\begin{array}{l}
[\texttt{def}]^{1}\ [\texttt{id(x)}]^{2}_{3}\texttt{:} \\
\qquad [\texttt{return x}]^{4} \\
[\texttt{a = }[\texttt{id("abc")}]^{5}_{6}]^{7} \\
[\texttt{b = }[\texttt{id(1)}]^{8}_{9}]^{10}
\end{array}
$$

Context-sensitive Analysis

$$[\texttt{def}]^{1}\ [\texttt{id(x)}]^{2}_{3}\texttt{:}$$
$$\quad [\texttt{return x}]^{4}$$
$$[\texttt{a = }[\texttt{id("abc")}]^{5}_{6}]^{7}$$
$$[\texttt{b = }[\texttt{id(1)}]^{8}_{9}]^{10}$$



Context-sensitive Analysis

- Problem: results of different calls are combined.
- Context-sensitive analysis: we separate different elements of the map lattice inside functions apart based on "what calls we did to get there" = call site sensitivity.
- Call-string context (up to a fixed depth)

Flow-insensitive Analysis

- Data flow analysis is flow-sensitive:
  different result (lattice value) for each program point.
- Flow-insensitive analysis:
  one global result.
- Optionally use flow-insensitive analysis for certain values:
  - types of module-scope variables
  - class types
  - instance types

Manually Specified Types

- Sometimes we have to resort to $\top$:
  - modules implemented in C
  - standard library modules
  - other libraries
- But types for a module can be specified in a text file.
- Example:

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

Manually Specified Types

- Sometimes we have to resort to $\top$:
  - modules implemented in C
  - standard library modules
  - other libraries
- But types for a module can be specified in a text file.
- Example:

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

- Polymorphic function types:

```
def id(x):
    return x

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

Experimental Evaluation

- Experiments: apply implementation to 5 Python projects
  - measure runtime
  - measure precision:
    percentage of results that are not $\top$ or $\bot$

Experimental Evaluation: Results

	euler	adventure	bitstring	feedparser	twitter
precision	0.45	0.75	0.91	0.53	0.75
time in ms	14	981	2,531	22,929	3,114
size (loc)	110	2,211	4,299	4,454	1,868

The value 0.45 is the percentage of variables for which a non-$\bot$ and non-$\top$ type could be inferred. The higher, the better.

Experimental Evaluation: Results

The best results were for flow-insensitive analysis for module-scope variables.

	euler	adventure	bitstring	feedparser	twitter
precision	0.45	0.75	0.91	0.53	0.75
time in ms	14	981	2,531	22,929	3,114
size (loc)	110	2,211	4,299	4,454	1,868
with manually specified types:					
precision	0.72	0.81			
time in ms	18	1,075			

Experimental Evaluation: Results

With respect to everything off:

	Precision	Runtime
Parameterized datatypes	+2.51 %	+5.80 %
Context-sensitive analysis	+0.69 %	+153.95 %
Flow-insensitive analysis		
for module-scope variables	+25.13 %	−19.98 %
for class types	+15.37 %	+264.88 %
for instance types	0.00 %	−0.79 %
Manually specified types	+54.59 %	+4.31 %

Some conclusions (for these examples)

- Having specified types helps much, costs little
- Context-sensitivity costs a lot, helps only little
- Flow-insensitive costs less and is more precise!!!
  - How is that possible?

Conclusions

- Method for approximate type inference for Python programs
  - based on data flow analysis
  - supporting Python's dynamic features
  - basic method + six variants
- Implementation
- Experimental evaluation

Future work

- exceptions, generators, and the with statement
- bigger programs
- a better approximation of quality

Thank you for your attention.