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:

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

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

\[
\begin{align*}
[a = 42]^1 \\
[b = 70]^2 \\
\text{while } [b \neq 0]^3: \\
\quad \text{if } [a > b]^4: \\
\quad \quad [a -= b]^5 \\
\quad \text{else:} \\
\quad \quad [b -= a]^6 \\
[gcd = a]^7
\end{align*}
\]
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 $l$ if there is an edge $(l', l)$
Solving Monotone Frameworks

- Fixpoint iteration
- Basic/naive algorithm:
  - Initialization:
    - $A_\circ(l) = \nu$ 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:
    \[
    \text{[def]}^{l_d} \ [\text{f(x)}]^{l_n}_{l_x} : \ldots
    \]
▶ Procedure call:
  ▶ call and return nodes $l_c$ and $l_r$
  ▶ Example:
    \[
    [\text{f(1)}]^{l_c}_{l_r}
    \]
▶ 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 $\Lambda$:
  $$\Lambda[f] = (l_n, l_x)$$
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 \( \Lambda \),
  - 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

\[ \text{def} \quad f(x) = \begin{cases} \text{return } x + 1 & \quad \text{if } x \geq 1 \\ f(1) & \quad \text{if } x = 0 \\ f(2) & \quad \text{if } x < 0 \end{cases} \]

Function table:

<table>
<thead>
<tr>
<th>x</th>
<th>f(1)</th>
<th>f(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>
Late Binding: Example

\[
\text{[def]}^1 \quad \text{[f(x)]}^2_3:
\]

\[
\text{[return } x + 1\text{]}^4
\]

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

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

Function table:

<table>
<thead>
<tr>
<th></th>
<th>(l_n)</th>
<th>(l_x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Late Binding: Example

\[
[\text{def}]^1 \quad [f(x)]^2_3 : \\
\quad [\text{return } x + 1]^4
\]

[\[f(1)]^5_6
\[f(2)]^7_8

Function table:

\[
\begin{array}{cc}
  l_n & l_x \\
  1 & 2 & 3 \\
\end{array}
\]
Late Binding: Example

\[
\begin{align*}
[\text{def}]^1 & \quad [f(x)]^2_3: \\
& \quad [\text{return } x + 1]^4 \\
[f(1)]^5_6 \\
f(2)]^7_8 \\
\text{Function table:} \\
\begin{array}{c|cc}
\hline
l_n & l_x \\
1 & 2 & 3 \\
\hline
\end{array}
\end{align*}
\]
Late Binding: Example

\[
\text{[def]}^1 \quad [f(x)]^2_3: \\
\quad \text{[return } x + 1\text{]}^4
\]

\[
[f(1)]^5_6 \quad [f(2)]^7_8
\]

Function table:

<table>
<thead>
<tr>
<th></th>
<th>(l_n)</th>
<th>(l_x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Inferring Types: Overview

Source Code
  ↓ Parsing
  ↓ Abstract Syntax Tree
  ↓ Control Flow Graph
  ↓ Monotone Framework
  ↓ Fixpoint iteration
  ↓ Results
Type Lattice for Python

Types for variables in Python programs:

\[ u \in \text{UTy} \quad \text{union types} \quad u ::= \{v\} \mid \top \]
Type Lattice for Python

Types for variables in Python programs:

\[ u \in UTy \quad \text{union types} \quad u ::= \{v\} | \top \]
\[ v \in ValTy \quad \text{value types} \quad v ::= b | f | c | i \]
Type Lattice for Python

Types for variables in Python programs:

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

  \[
  x = 1 \\
  y = "a"
  \]

\[
\{ x \mapsto \{ \text{int} \}, y \mapsto \{ \text{str} \} \}
\]
Analysis variants under evaluation

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

- Example code:
  ```python
  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:
    ```python
    list  \langle \text{int} \rangle
    ```
    for list containing values of type `int`

- Extended type lattice:
  ```
b ::= \cdots | list \langle u \rangle | set \langle u \rangle | dict \langle u; u \rangle | tuple \langle [u] \rangle
```
Parameterized Datatypes

- Example code:
  ```python
  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<int>` for list containing values of type `int`
  - Extended type lattice:
    $$ b ::= \cdots \mid list(u) \mid set(u) \mid dict(u; u) \mid tuple([u]) $$
Context-sensitive Analysis

\[
\text{def}^1 \ [\text{id}(x)]^2_3: \\
\text{return } x^4 \\
[a = [\text{id}("abc")]^5_6]^7 \\
b = [\text{id}(1)]^8_9]^{10}
\]
[def] \[1\]  [id(x)] \[2\]:
[return x] \[4\]
[a = [id("abc")]] \[5\]
[b = [id(1)]] \[8\]

Diagram:

1 \rightarrow 5
5 \rightarrow 6\rightarrow 7
6 \rightarrow 2\rightarrow 4
2 \rightarrow 3
3 \rightarrow 9
9 \rightarrow 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)
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:
  
  ```python
  math.pi : \{float\}
  math.sqrt : \{\lambda \{\text{bool, int, float}\} \to \{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

<table>
<thead>
<tr>
<th></th>
<th>euler</th>
<th>adventure</th>
<th>bitstring</th>
<th>feedparser</th>
<th>twitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>precision</td>
<td>0.45</td>
<td>0.75</td>
<td>0.91</td>
<td>0.53</td>
<td>0.75</td>
</tr>
<tr>
<td>time in ms</td>
<td>14</td>
<td>981</td>
<td>2,531</td>
<td>22,929</td>
<td>3,114</td>
</tr>
<tr>
<td>size (loc)</td>
<td>110</td>
<td>2,211</td>
<td>4,299</td>
<td>4,454</td>
<td>1,868</td>
</tr>
</tbody>
</table>

The value 0.45 is the percentage of variables for which a non-⊥ and non-⊤ type could be inferred. The higher, the better.
Experimental Evaluation: Results

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

<table>
<thead>
<tr>
<th></th>
<th>euler</th>
<th>adventure</th>
<th>bitstring</th>
<th>feedparser</th>
<th>twitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>precision</td>
<td>0.45</td>
<td>0.75</td>
<td>0.91</td>
<td>0.53</td>
<td>0.75</td>
</tr>
<tr>
<td>time in ms</td>
<td>14</td>
<td>981</td>
<td>2,531</td>
<td>22,929</td>
<td>3,114</td>
</tr>
<tr>
<td>size (loc)</td>
<td>110</td>
<td>2,211</td>
<td>4,299</td>
<td>4,454</td>
<td>1,868</td>
</tr>
</tbody>
</table>

with manually specified types:

<table>
<thead>
<tr>
<th></th>
<th>euler</th>
<th>adventure</th>
<th>bitstring</th>
<th>feedparser</th>
<th>twitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>precision</td>
<td>0.72</td>
<td>0.81</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>time in ms</td>
<td>18</td>
<td>1,075</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Experimental Evaluation: Results

With respect to everything off:

<table>
<thead>
<tr>
<th>Parameterized datatypes</th>
<th>Precision</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+2.51 %</td>
<td>+5.80 %</td>
</tr>
<tr>
<td>Context-sensitive analysis</td>
<td>+0.69 %</td>
<td>+153.95 %</td>
</tr>
<tr>
<td>Flow-insensitive analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>for module-scope variables</td>
<td>+25.13 %</td>
<td>−19.98 %</td>
</tr>
<tr>
<td>for class types</td>
<td>+15.37 %</td>
<td>+264.88 %</td>
</tr>
<tr>
<td>for instance types</td>
<td>0.00 %</td>
<td>−0.79 %</td>
</tr>
<tr>
<td>Manually specified types</td>
<td>+54.59 %</td>
<td>+4.31 %</td>
</tr>
</tbody>
</table>

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.