Alias Analysis for Optimization of Dynamic Languages

DLS 2010

Michael Gorbovitski Yanhong A. Liu Scott D. Stoller
Tom Rothamel K. Tuncay Tekle

Vincent Foley-Bourgon
COMP-763 - Fall 2013
McGill University

November 2013
Plan

1. The Title
2. The Big Idea
3. The Analysis
4. The Experiments
Plan

1. The Title
2. The Big Idea
3. The Analysis
4. The Experiments
5. The Offer
The Title
Alias Analysis for Optimization of Dynamic Languages
Alias Analysis for **Optimization** of Dynamic Languages
Alias Analysis for Optimization of Dynamic Languages
Alias Analysis for Optimization of Dynamic Languages
The title says it all!

- What do we want? To optimize programs!
- Which programs? Those written in dynamic languages!
- What tool will we use? Alias analysis!
The Big Idea
“Dynamic languages such as Python allow programs to be written more easily using high-level constructs such as comprehensions for queries and using generic code.”
The Big Idea

Efficient execution of these programs requires powerful optimizations:

- Incrementalization of queries
- Specialization of generic code

Both require precise and scalable *alias analysis*
<Interlude>
What is incrementalization?

---

What is incrementalization?

“Given a program $f$ and an operation $\oplus$, a program $f'$ is called an incremental version of $f$ under $\oplus$ if $f'$ computes $f(x \oplus y)$ efficiently by making use of $f(x)$.”

\footnote{Source: Efficiency by Incrementalization: an Introduction, Liu 2000}
Incrementalization

Example:

- $f = \text{sort}$
- $\oplus = \text{cons}$
- $f' = \text{insort}$

$$sort(\text{cons}(y, x)) = \text{insort}(y, \text{sort}(x))$$

$\Theta(n \lg n) \supset \Theta(n)$
< /Interlude>
The Big Idea

- *InvTS*: incrementalization optimization, (source, alias-analysis result) → target

- *Psyco*: specializing JIT, modified to accept statically computed alias and type information
The Big Idea

- Perform alias analysis on a program
- Run \textit{InvTS} \parallel \textit{Psyco}
- Get a faster program
The Big Idea

- Perform alias analysis on a program
- Run InvTS || Psyco
- Get a faster program

Better alias analysis $\Rightarrow$ Faster programs
Question #1

Why would a better alias analysis yield faster programs?
Answer #1

Improved precision allows an optimizer more opportunities to perform more transformations.
The Analysis
The Analysis

1. Parse Python program into an AST
2. Analyze types and construct CFG \(^2\)
3. Construct a sparse evaluation graph (SEG) from the CFG by removing CFG nodes that do not affect aliases
4. Do the described alias analysis

\(^2\)Apply steps 1 and 2 recursively for import statements
The Analysis

Parsing Python

Very easy to do with Python’s stdlib:

```python
text
import ast

with open("main.py") as f:
    root = ast.parse(f.read())
    next_step(root)

Done.
```
The Analysis

Precise type analysis

Inference algorithm that computes not only the basic types, but also values, ranges of values, number of elements in a collection, etc.

Basic types:

- none
- primitive types: int, float, bool
- collection types: string, list, tuple, set, dict
- module
- class
- instance
- function
- method
- union: combine different types together
- \( \top \) and \( \bot \)
The Analysis
Precise type analysis

Precise types:

- Int
- IntRange(n1, n2)
- IntNonNegative
- IntVal(n)

if $n > 0$ or $n1 \leq n \leq n2$

- List
- ListLen(n)
- ListElem(t)
- ListAll(t1 to tn)

if $n$ have same value or $t1..tn$ are the same type

- Bool
- BoolVal(True)
- BoolVal(False)
The Analysis
Precise type analysis

- Any set \( \{t_1, ..., t_n\} \) has a minimal super type: \( \top \) if any \( t_i \) is \( \top \), otherwise the maximal type of the union of all \( t_i \).

- Limit for the size of type description: no more than 60 type names.

- Generalization: going from a type to a supertype of smaller size when the size of a type exceeds a constant.

Example:

\[
\text{union}(\text{int}_{\text{val}}(2), \text{int}_{\text{val}}(4), \text{int}_{\text{val}}(8)) \implies \text{union}(\text{int}_{\text{range}}(2, 4), \text{int}_{\text{val}}(8))
\]
The Analysis

Constructing the CFG

Type Analysis
(build CFG)

until fixed point

max 30 times

Refinement
The Analysis
Constructing the CFG

**Analysis:** start at program entry and visit and interpret each program node. Types of variables and expressions start at ⊥ and go up until fixed-point.

**Refinement:**

- Clone functions so that there is one clone for each different combination of basic types of arguments
- Eliminate code that is dead for the argument types, and fix the call sites
- Inline function calls when that doesn’t increase the number of program nodes.
Authors claim that handling of most nodes is obvious, but give some details on how to handle the dynamic features of Python. I won’t go into all of them, but let’s look at our friend `eval`. 
Question #2

There are two cases to handle for `eval`:

- What do you think are the two cases?  
  (Hint: remember Ismail’s first presentation)

- How do you handle each case?
The Analysis

Eval

The analysis distinguishes two cases:

▶ If the argument type is a union of *constant* strings, inner function nodes are created and edges are added appropriately. Return type is the minimum super type of the newly-created functions.

▶ Otherwise, the return type of *eval* is $\top$. 

What is alias analysis?
Alias analysis: compute pairs of variables and fields that refer to the same object.\(^3\)

```plaintext
int o = 42;
int *p, *q;
p = &o;
q = &o;
```

\(^3\)Undecidable in general.
The Analysis

Their proposed alias analysis:

- “May” analysis (over-approximation)
- Inter-procedural
- Is flow-sensitive
- Is context-sensitive (trace sensitivity)
- Uses precise type analysis
- Uses a compressed representation
The Analysis
Flow sensitivity

```python
#removes all instances of 0 from collection C
def removeObject(C, O):
    if isinstance(C, set):
        if 0 in C:
            C.remove(0)

    if isinstance(C, list):
        for n in range(C.count(0)):
            C.remove(0)
```

Incrementalization is going to add guards before the `remove` method calls; with flow sensitivity, the alias set of C can be different at the two different call sites, and if the alias set has only one member, the guard can be removed.
Only allow alias pairs that have compatible types.

“Our experiments show that using precise types significantly increases alias analysis precision compared to using basic types.”
The Analysis
Types to improve precision

Only allow alias pairs that have compatible types.

“Our experiments show that using precise types significantly increases alias analysis precision compared to using basic types.”

Mais pourquoi!? 
The Analysis
Trace sensitivity

Context sensitivity necessary for precise alias analysis. Traditional $n$-CFA not great with dynamic languages:

- If $n$ is small, precision suffers
- If $n$ is larger, memory usage becomes unacceptably high
The Analysis
Trace sensitivity

- Inline non-recursive calls
- Inline recursive calls once
- Merge alias pairs from the inlined procedures into the corresponding SEG node
- Remove inlined nodes (save memory)
“Our trace-sensitive analysis is always at least as precise as, and in our experiments always more precise than, context-insensitive analyses. The increased precision is because our algorithm distinguishes aliasing information in different contexts during analysis, even though it subsequently merges information for different contexts.”
To reduce memory usage, they introduce a “simple, but important optimization”.

If a node as a single predecessor, the alias pairs are not stored directly, but as a diff of the predecessor node.

Reduces memory consumption by 10x
The Experiments
The Experiments

Disclaimer

All tables and figures are taken from the article.
The Experiments

The setup

18 variants of the analysis:

- Flow-insensitive + Context insensitive
- Flow-insensitive + Context sensitive
- Flow-sensitive + Context insensitive
- Flow-sensitive + Context sensitive
- Flow-sensitive + Trace sensitive
- Flow-sensitive + Trace sensitive + extra clones \(^4\)

Each is combined with no type checking, basic type checking and precise type checking.

\(^4\)Recursive functions are inlined twice
The Experiments
Effect on incrementalization

<table>
<thead>
<tr>
<th>flow sensitive</th>
<th>context sensitivity</th>
<th>type sensitivity</th>
<th>lxml - Valid Parent</th>
<th>lxml - No Shared Child</th>
<th>lxml - Indexing</th>
<th>nftp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>runtime checks</td>
<td>analysis</td>
<td>runtime checks</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>overhead removed</td>
<td>analysis</td>
<td>overhead removed</td>
<td>time</td>
</tr>
<tr>
<td>no</td>
<td>no</td>
<td>basic</td>
<td>92%</td>
<td>12</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>precise</td>
<td>91%</td>
<td>14</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>basic</td>
<td>88%</td>
<td>16</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>precise</td>
<td>74%</td>
<td>26</td>
<td>61</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>no</td>
<td>basic</td>
<td>87%</td>
<td>17</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>precise</td>
<td>73%</td>
<td>28</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>basic</td>
<td>83%</td>
<td>18</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>precise</td>
<td>73%</td>
<td>30</td>
<td>61</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>trace</td>
<td>basic</td>
<td>82%</td>
<td>20</td>
<td>81</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>precise</td>
<td>75%</td>
<td>28</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>yes</td>
<td>trace extra</td>
<td>basic</td>
<td>67%</td>
<td>37</td>
<td>308</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>precise</td>
<td>19%</td>
<td>61</td>
<td>308</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Runtime overhead, number of alias checks removed, and analysis time (in seconds) in InvTS experiments. Runtime overhead is $\frac{time_t - time_o}{time_o}$, where $time_t$ and $time_o$ are running times of the transformed and original programs, respectively.
The Experiments
Effect on incrementalization

Figure 1. Runtime overhead of transformed programs, using precise-type-sensitive alias analysis, varying flow and context sensitivity.
The Experiments
Effect on specialization

<table>
<thead>
<tr>
<th>flow sensitive</th>
<th>context sensitivity</th>
<th>type sensitivity</th>
<th>program speedup</th>
<th>uncompiled procedures</th>
<th>analysis time</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td>no</td>
<td>basic</td>
<td>3.8%</td>
<td>27</td>
<td>1.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>precise</td>
<td>4.8%</td>
<td>26</td>
<td>1.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6.7%</td>
<td>23</td>
<td>2.2</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>basic</td>
<td>7.2%</td>
<td>24</td>
<td>26.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>precise</td>
<td>7.7%</td>
<td>23</td>
<td>26.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10.9%</td>
<td>21</td>
<td>27.0</td>
</tr>
<tr>
<td>yes</td>
<td>no</td>
<td>basic</td>
<td>7.2%</td>
<td>25</td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>precise</td>
<td>7.2%</td>
<td>23</td>
<td>4.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11.3%</td>
<td>20</td>
<td>4.2</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>basic</td>
<td>6.7%</td>
<td>24</td>
<td>23.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>precise</td>
<td>7.7%</td>
<td>23</td>
<td>24.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>13.4%</td>
<td>18</td>
<td>23.8</td>
</tr>
<tr>
<td>yes</td>
<td>trace</td>
<td>basic</td>
<td>8.2%</td>
<td>24</td>
<td>51.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>precise</td>
<td>10.0%</td>
<td>22</td>
<td>51.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15.5%</td>
<td>16</td>
<td>52.6</td>
</tr>
<tr>
<td>yes</td>
<td>trace extra</td>
<td>basic</td>
<td>9.9%</td>
<td>22</td>
<td>331.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>precise</td>
<td>11.3%</td>
<td>20</td>
<td>335.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15.9%</td>
<td>15</td>
<td>339.3</td>
</tr>
</tbody>
</table>

**Table 2.** Program speedup, number of procedures left uncompiled at compile-time, and analysis time (in seconds) in Psyco experiments. Program speedup is $\frac{\text{time}_a - \text{time}_o}{\text{time}_o}$, where $\text{time}_a$ is the running time using Psyco with alias information, and $\text{time}_o$ is the time using the original Psyco, which leaves 30 procedures uncompiled.
The Experiments

Alias set size

Figure 2. Alias set size for each variable (shown horizontally) for each CFG node (shown vertically) for flow-sensitive analysis variants for tarfile. Variables are ordered by increasing average alias set size in the context-insensitive precise-type-sensitive analysis.
### The Experiments

#### Memory usage

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Nodes</th>
<th>AST Nodes</th>
<th>unoptimized</th>
<th>compressed</th>
<th>trace-sensitive</th>
<th>compressed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>time memory</td>
<td></td>
<td>time memory</td>
<td></td>
</tr>
<tr>
<td>chunk</td>
<td>172</td>
<td>493</td>
<td></td>
<td>4.09 41.74</td>
<td>4.97 39.16</td>
<td>5.65 39.13</td>
<td>7.10 42.26</td>
</tr>
<tr>
<td>bdb</td>
<td>609</td>
<td>2026</td>
<td></td>
<td>7.60 43.76</td>
<td>7.61 41.40</td>
<td>8.76 40.18</td>
<td>12.90 49.46</td>
</tr>
<tr>
<td>pickle</td>
<td>1392</td>
<td>4239</td>
<td></td>
<td>11.12 291.61</td>
<td>13.94 88.60</td>
<td>15.97 59.74</td>
<td>21.11 812.11</td>
</tr>
<tr>
<td>tarfile</td>
<td>1796</td>
<td>7877</td>
<td></td>
<td>31.36 4203.29</td>
<td>45.90 1751.84</td>
<td>52.38 688.53</td>
<td>out of memory</td>
</tr>
<tr>
<td>Fortran</td>
<td>6503</td>
<td>15955</td>
<td></td>
<td>123.65 3018.57</td>
<td>262.93 1202.04</td>
<td>298.23 627.41</td>
<td>out of memory</td>
</tr>
<tr>
<td>bitTorrent</td>
<td>22423</td>
<td>102930</td>
<td></td>
<td>out of memory</td>
<td>out of memory</td>
<td>1068.36 10618.39</td>
<td>1211.87 2909.11</td>
</tr>
<tr>
<td>std. lib.</td>
<td>51654</td>
<td>420654</td>
<td></td>
<td>out of memory</td>
<td>out of memory</td>
<td>3401.69 13124.52</td>
<td>out of memory</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Nodes</th>
<th>AST Nodes</th>
<th>unoptimized</th>
<th>compressed</th>
<th>trace-sensitive</th>
<th>compressed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>time memory</td>
<td></td>
<td>time memory</td>
<td></td>
</tr>
<tr>
<td>chunk</td>
<td>172</td>
<td>493</td>
<td></td>
<td>1.01 31.06</td>
<td>1.28 31.04</td>
<td>2.58 39.07</td>
<td>3.10 39.07</td>
</tr>
<tr>
<td>bdb</td>
<td>609</td>
<td>2026</td>
<td></td>
<td>1.20 33.25</td>
<td>1.48 32.03</td>
<td>4.52 41.71</td>
<td>5.07 40.85</td>
</tr>
<tr>
<td>pickle</td>
<td>1392</td>
<td>4239</td>
<td></td>
<td>1.65 76.20</td>
<td>1.98 36.51</td>
<td>10.04 121.43</td>
<td>10.11 49.48</td>
</tr>
<tr>
<td>tarfile</td>
<td>1796</td>
<td>7877</td>
<td></td>
<td>3.23 1964.09</td>
<td>4.16 267.70</td>
<td>not applicable</td>
<td>20.69 2384.95</td>
</tr>
<tr>
<td>Fortran</td>
<td>6503</td>
<td>15955</td>
<td></td>
<td>11.94 928.16</td>
<td>12.77 157.25</td>
<td>77.71 1142.45</td>
<td>80.97 188.16</td>
</tr>
<tr>
<td>bitTorrent</td>
<td>22423</td>
<td>102930</td>
<td></td>
<td>63.01 8134.75</td>
<td>90.01 1198.93</td>
<td>298.86 11555.96</td>
<td>330.44 1574.81</td>
</tr>
<tr>
<td>std. lib.</td>
<td>51654</td>
<td>420654</td>
<td></td>
<td>out of memory</td>
<td>out of memory</td>
<td>317.44 2434.01</td>
<td>out of memory</td>
</tr>
</tbody>
</table>

Table 3. Running time (in seconds) and maximum memory usage (in MBytes) for flow- and precise-type-sensitive alias analysis variants. “unoptimized” means that trace optimization and compression are both disabled; trace optimization is enabled for all other trace-sensitive variants; “not applicable” means that trace optimization is not applicable to trace-insensitive variants; “out of memory” means that the memory usage of the analysis exceeded 16 GB; “out of time” means that its running time exceeded 4 hours.
The Offer
We’ve seen in presentations this semester that dynamically-typed languages present hard challenges:
We’ve seen in presentations this semester that dynamically-typed languages present hard challenges:

- Optimizations are harder
We’ve seen in presentations this semester that dynamically-typed languages present hard challenges:

- Optimizations are harder
- Static analyses are less precise
We’ve seen in presentations this semester that dynamically-typed languages present hard challenges:

- Optimizations are harder
- Static analyses are less precise
- Development tools are more rudimentary
We’ve seen in presentations this semester that dynamically-typed languages present hard challenges:

- Optimizations are harder
- Static analyses are less precise
- Development tools are more rudimentary
- No machine-checked form of “documentation”
We’ve seen in presentations this semester that dynamically-typed languages present hard challenges:

- Optimizations are harder
- Static analyses are less precise
- Development tools are more rudimentary
- No machine-checked form of “documentation”
- No safety net for maintenance and refactorings
We’ve seen in presentations this semester that dynamically-typed languages present hard challenges:

- Optimizations are harder
- Static analyses are less precise
- Development tools are more rudimentary
- No machine-checked form of “documentation”
- No safety net for maintenance and refactorings
- No ability to encode compiler-checked invariants
Cost of dynamic typing

These represent the *price* of using dynamically-typed languages.

1. What does it buy us?

2. Is it worth the price?

You have answers or opinions? Come see me and let’s discuss this over a beer!