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

The Title

Alias Analysis for Optimization of Dynamic Languages

The Title

Alias Analysis for **Optimization** of Dynamic Languages

The Title

Alias Analysis for Optimization of **Dynamic Languages**

The Title

Alias Analysis for Optimization of Dynamic Languages

The Title

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

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>

Incrementalization

What is incrementalization?

[1] Source: Efficiency by Incrementalization: an Introduction, Liu 2000

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 $f(x)$."[1]

[1] Source: Efficiency by Incrementalization: an Introduction, Liu 2000

Incrementalization

Example:

- f = sort
- $\oplus$ = cons
- f' = insort

$$sort(cons(y, x)) = insort(y, sort(x))$$
$$\Theta(n \lg n) \supset \Theta(n)$$

< /Interlude>

The Big Idea

- *InvTS*: incrementalization optimization,
  (source, alias-analysis result) $\rightarrow$ target
- *Psyco*: specializing JIT, modified to accept statically computed alias and type information

The Big Idea

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

The Big Idea

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

Better alias analysis $\Longrightarrow$ **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

Very easy to do with Python's stdlib:

```
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.

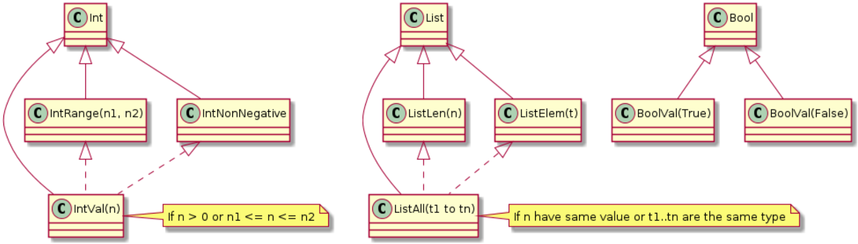
Basic types:

- *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:



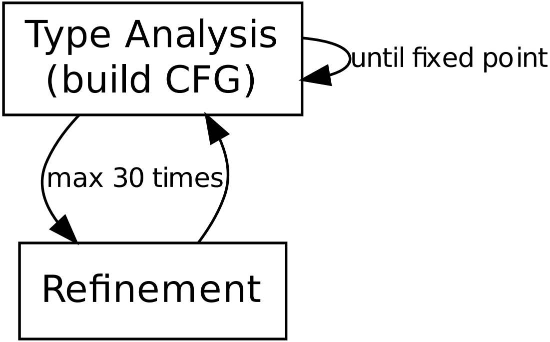
- Any set $\{t_1, ..., t_n\}$ has a minimal super type: $\top$ if any t_i is $\top$, otherwise the maximal type of the union of all t_i.
- Limit for the size of type description: no more than 60 type names.
- Generalization: going from a type to a supertype of smaller size when the size of a type exceeds a constant.

  Example:

$$union(int_{val}(2), int_{val}(4), int_{val}(8)) \implies$$
$$union(int_{range}(2, 4), int_{val}(8))$$

The Analysis

Constructing the CFG



The Analysis

Constructing the CFG

Analysis: start at program entry and visit and interpret each program node. Types of variables and expressions start at $\perp$ 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 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 $\top$.

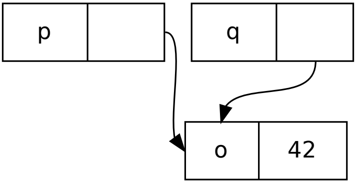
The Analysis
Question #3

What is alias analysis?

The Analysis

Alias analysis: compute *pairs* of variables and fields that *refer to the same object.*[3]

```
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

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

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

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.

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."

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)

The Analysis

Trace sensitivity

"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

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

			lxml - Valid Parent 97 alias checks			lxml - No Shared Child 81 alias checks			lxml - Indexing 1451 alias checks			nftp 31 alias checks		
flow sensitive	context sensitivity	type sensitivity	runtime overhead	checks removed	analysis time	runtime overhead	checks removed	analysis time	runtime overhead	checks removed	analysis time	runtime overhead	checks removed	analysis time
no	no	no	92%	12	36	95%	12	39	440%	35	49	119%	7	19
		basic	93%	12	36	95%	13	38	429%	35	50	119%	7	19
		precise	91%	14	36	95%	13	39	381%	41	49	112%	9	19
no	yes	no	88%	16	60	94%	15	62	364%	55	97	110%	9	83
		basic	88%	17	64	93%	17	62	350%	61	97	96%	11	82
		precise	74%	26	61	90%	23	61	323%	89	99	91%	13	84
yes	no	no	87%	17	42	93%	19	42	340%	79	62	93%	12	30
		basic	86%	17	43	91%	20	43	331%	81	61	89%	13	30
		precise	73%	28	43	90%	28	46	219%	122	61	89%	13	30
yes	yes	no	83%	18	59	93%	20	57	310%	103	98	91%	13	80
		basic	82%	18	61	90%	23	63	303%	112	95	86%	14	82
		precise	73%	30	61	89%	29	61	192%	199	98	81%	14	81
yes	trace	no	82%	20	81	91%	19	85	160%	246	103	90%	12	63
		basic	75%	28	82	88%	28	85	133%	344	109	77%	14	62
		precise	14%	68	82	85%	40	86	85%	836	104	73%	16	63
yes	trace extra	no	67%	37	308	85%	37	312	124%	455	783	78%	14	119
		basic	19%	61	308	85%	38	310	99%	603	780	74%	15	119
		precise	14%	72	310	83%	41	311	83%	892	791	70%	17	118

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

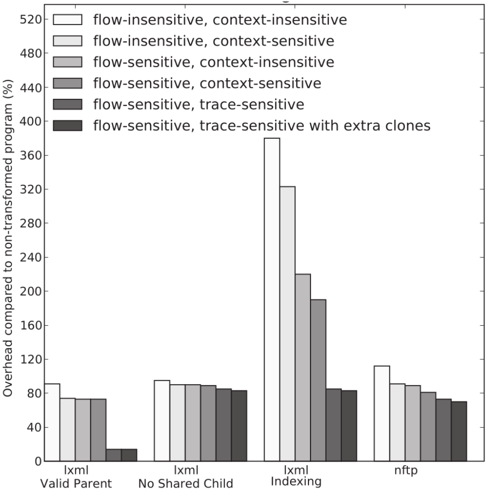


Figure 1. Runtime overhead of transformed programs, using precise-type-sensitive alias analysis, varying flow and context sensitivity.

The Experiments

Effect on specialization

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

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

The Experiments

Alias set size

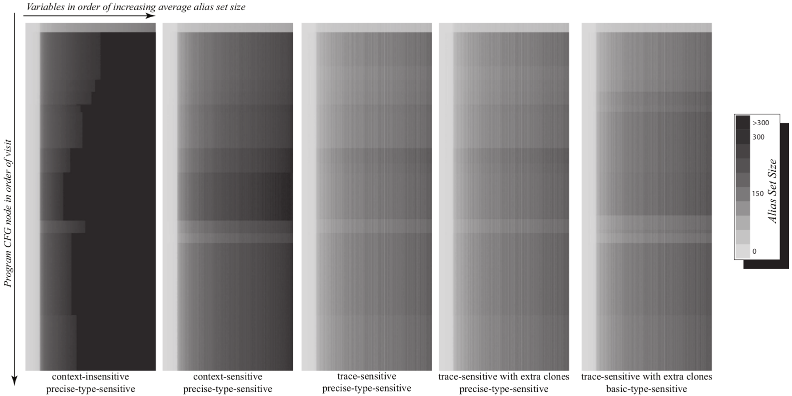


Figure 2. Alias set size for each variable (shown horizontally) for each CFG node (shown vertically) for flow-sensitive analysis variants for `tarfile`. Variables are ordered by increasing average alias set size in the context-insensitive precise-type-sensitive analysis.

The Experiments

Memory usage

Program	LOC	AST Nodes	context-insensitive unoptimized time	context-insensitive unoptimized memory	context-insensitive uncompressed time	context-insensitive uncompressed memory	context-insensitive compressed time	context-insensitive compressed memory	context-sensitive unoptimized time	context-sensitive unoptimized memory	context-sensitive uncompressed time	context-sensitive uncompressed memory	context-sensitive compressed time	context-sensitive compressed memory
chunk	172	493	not applicable		1.01	31.06	1.28	31.04	not applicable		2.58	39.07	3.10	39.07
bdb	609	2026			1.20	33.25	1.48	32.03			4.52	41.71	5.07	40.85
pickle	1392	4239			1.65	76.20	1.98	36.51			10.04	121.43	10.11	49.48
tarfile	1796	7877			3.23	1964.09	4.16	267.70			20.69	2384.95	23.11	341.45
Fortran	6503	15955			11.94	928.16	12.77	157.25			77.71	1142.45	80.97	188.16
bitTorrent	22423	102930			63.01	8134.75	90.01	1198.93			298.86	11555.96	330.44	1574.81
std. lib.	51654	420654			out of memory		317.44	2434.01			out of memory		1519.68	3726.77

Program	LOC	AST Nodes	trace-sensitive unoptimized time	trace-sensitive unoptimized memory	trace-sensitive uncompressed time	trace-sensitive uncompressed memory	trace-sensitive compressed time	trace-sensitive compressed memory	trace-sensitive with extra clones unoptimized time	trace-sensitive with extra clones unoptimized memory	trace-sensitive with extra clones uncompressed time	trace-sensitive with extra clones uncompressed memory	trace-sensitive with extra clones compressed time	trace-sensitive with extra clones compressed memory
chunk	172	493	4.09	41.74	4.97	39.16	5.65	39.13	7.10	42.26	8.89	39.26	10.37	39.15
bdb	609	2026	7.60	43.76	7.61	41.40	8.76	40.18	12.90	49.46	13.91	46.15	16.08	40.85
pickle	1392	4239	11.12	291.61	13.94	88.60	15.97	59.74	21.11	812.11	34.69	294.06	43.13	162.91
tarfile	1796	7877	31.36	4203.29	45.90	1751.84	52.38	688.53	out of memory		236.76	8631.85	283.45	2570.28
Fortran	6503	15955	123.65	3018.57	262.93	1202.04	298.23	627.41	out of memory		2687.26	8645.29	3389.17	3602.21
bitTorrent	22423	102930	out of memory		1068.36	10618.39	1211.87	2909.11	out of memory		out of time		out of time	
std. lib.	51654	420654	out of memory		out of memory		3401.69	13124.52	out of memory		out of time		out of time	

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.

</presentation>

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!