Improving the precision of incremental algorithms for pointer analysis

Nick Roberts
(nroberts@andrew.cmu.edu)

Vijay Ramamurthy
(vijayram@andrew.cmu.edu)

https://ncik-roberts.github.io/15745
15-745 Spring 2019

1 Introduction

Pointer analysis is a well-established technique for estimating aliasing among variables and heap objects in a program. Its results can be used in race detection, among other compilation and debugging scenarios. Incremental algorithms (or "differential algorithms") for pointer analysis allow for greater efficiency in the presence of change: As the programmer modifies source code for which pointer analysis facts have already been derived, an incremental algorithm will update the derived facts accordingly, with the same result as if a non-incremental algorithm had been run on the entirety of the modified program. The benefit of an incremental algorithm is that updating a database of facts can be orders of magnitude more efficient than rederiving the database from scratch.

Recent work by Liu and Huang on their D4 framework [1] puts this into practice, combining the insight about incremental algorithms with two other major insights: (1) that incremental algorithms can be designed for parallelism, and (2) that decoupling the client (the IDE where the code is developed) from the server (the location where the pointer analysis data structures are maintained) makes it feasible for the server to be a much higher-performance machine than would be reasonable for a client. Applying these insights together, they see significant speedup in race detection: 0.12s, on average, compared to 25s using previous approaches. But their improved incremental algorithm alone provides speedup compared to previous incremental approaches: on average, 0.72ms (versus 6.8ms) for insertions and 73ms (versus 26s) for deletions.

Liu and Huang's algorithm is based on Andersen's flow-insensitive, context-insensitive algorithm for points-to analysis. Because their algorithm is context-insensitive, when analyzing the program in Figure 1 it computes that the variables `b1` and `b2` may each point to an object allocated at allocation site 1 or allocation site 2, when in reality `b1` may only point to an object allocated at site 1 and `b2` may only point to an object allocated at site 2. This inaccuracy comes from the context-insensitive treatment of the variable `x`. As this analysis treats the arguments at call sites 1 and 2 as flowing into the same variable `x`, it is computed `x` may point to an object from allocation site 1 or 2. Context-insensitivity therefore manifests itself in D4 as a source of false positives: the algorithm thinks that variables may be manipulating data which they are not, resulting in spurious bug reporting.

Qualifying variables with the call sites they were reached from would reduce the algorithm's unsoundness. Instead of treating `x` as a single entity which allocation site 1 and allocation site 2 both flow into, such a context-sensitive algorithm would treat `x` reached through call site 1 as a separate entity from `x` reached through call site 2 and therefore correctly compute that `b1` may only point to allocation site 1 and similarly for `b2`.

```
void foo() {
  Object a1 = new Object(); // allocation site 1
  Object b1 = identity(a1); // call site 1
}
void bar() {
  Object a2 = new Object(); // allocation site 2
  Object b2 = identity(a2); // call site 2
}
Object identity(Object x) {
  return x;
}
void main() { // entry point
  foo(); // call site 3
  bar(); // call site 4
}
```

Figure 1: A program with two allocation sites

Our primary contribution is an algorithm for context-sensitive pointer analysis which makes use of Liu and Huang's incremental techniques to get similar performance gains. As Liu and Huang's analysis improved on the state-of-the-art in an incremental setting, our algorithm improving on theirs is novel. We implement this algorithm and compare the performance to the original context-sensitive algorithm. To conclude, we explore the possibility of an alternative incremental algorithm for context-sensitive pointer analysis which uses less memory than our primary one.

2 Algorithm design

2.1 Graph construction

At the core of Liu and Huang's incremental implementation of Andersen's algorithm is the construction of a *pointer assignment graph* (PAG) which encompasses how data flows between variables in a program. Figure 2 demonstrates what the PAG looks like for the program in Figure 1 under Liu and Huang's context-insensitive analysis.

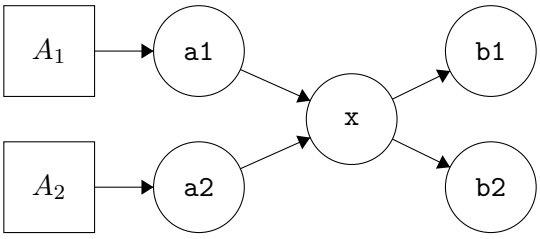

Figure 2: Context-insensitive PAG for Figure 1

There are two types of nodes: those corresponding to allocation sites (denoted as squares and labeled A_1 and A_2 to represent allocation site 1 and allocation site 2, respectively) and those corresponding to program variables (denoted as circles). An edge $u \to v$ represents an assignment v = u; that is, the data in u flows into v. A variable v may then point to an object

allocated an allocation site u if there exists a path from u to v. The algorithm computes the points-to sets $pts(v)$ for each variable node v by enforcing the relation that for any edge $u \to v$, $pts(u) \subseteq pts(v)$.

The issue caused by context-insensitivity manifests itself in the PAG as the single node `x` coincident to both `a1` and `a2`. Since `x` is mapped to just one node despite having two different meanings in two different calling contexts, there is a path from each of A_1 and A_2 to each of `b1` and `b2`. This results in the original algorithm erroneously computing the points-to set $\{A_1, A_2\}$ for each of `b1` and `b2`.

Our algorithm maintains the PAG differently; each variable node is qualified with the context through which it was reached, where the context is determined by the sequence of call-sites via which the node was reached.

In Figure 3, we construct the PAG for Figure 1 with a 1-call site–sensitive context. (That is, the context for a node is just the call site from which the node was reached; we defer a presentation of a more general framework for determining context to Section 3.) In Figure 3, call sites 1, 2, 3, and 4 are denoted by C_1, C_2, C_3, and C_4, respectively.

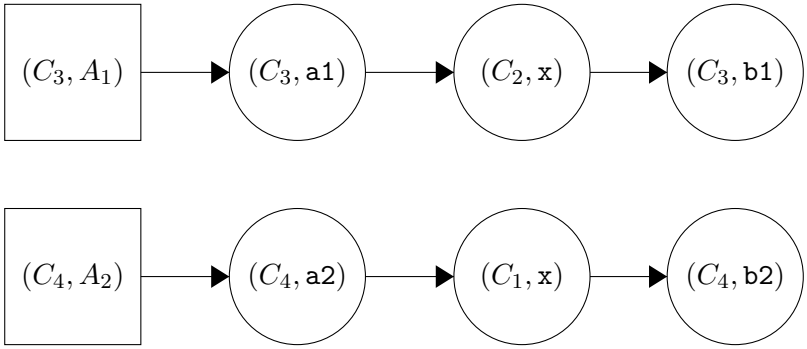

Figure 3: Context-sensitive PAG for Figure 1

Qualifying variable nodes to be context sensitive removes the path from A_1 to `b2` and the path from A_2 to `b1`, calculating precisely that $pts(C_3, \texttt{b1}) = \{(C_3, A_1)\}$ and $pts(C_4, \texttt{b2}) = \{(C_4, A_2)\}$

2.2 Incremental updates

Reconstructing the PAG upon every insertion and deletion of a program statement is expensive. The key insight to efficient incremental updates is to make small incremental changes to the graph every time a program statement is inserted or deleted. In Liu and Huang's algorithm, inserting a statement v `=` u inserts an edge $u \to v$, and deleting a statement v `=` u deletes an edge $u \to v$. Upon inserting and deleting such edges, the points-to sets for nodes affected by this change must be recomputed.

By keeping our graph construction similar to Liu and Huang's, the recomputation of points-to sets can be done locally and therefore at multiple parts of the graph in parallel for the same reasons that these are afforded by the original algorithm. The algorithm for local propagation of points-to set updates remains the same as in Liu and Huang's algorithm.

The key difference in how our algorithm does updates is therefore not in how points-to sets are resolved once edges are added/removed, but in which edges are added/removed. Rather than a statement v `=` u corresponding to an edge $u \to v$, it corresponds to all edges $(C, u) \to (C, v)$ where C ranges over all call sites to the function which contains this statement. This multiplies the amount of work associated with an update by the number of call sites. Fortunately, each

of these updates is still independent of one another and therefore easy to parallelize. Section 5 explores an alternative to this approach which does not duplicate nodes and edges yet is still context-sensitive.

3 Implementation

Liu and Huang's algorithm is implemented in the WALA static analysis framework for Java[1], and, as such, its pointer analysis must operate over the full set of Java bytecode instructions. Furthermore, the analysis must be implemented using only the extension points defined by the WALA framework. For these reasons, we found the original implementation to be difficult to instrument with call-site–based contexts.

Instead, we reimplement Liu and Huang's algorithm, making the following modifications:

- The implementation is defined with respect to a simplified subset of Java bytecode instructions.
- The implementation creates the PAG for arbitrary call-site–based contexts, and allows for incremental updates to all nodes affected by the addition and deletion of statements from the program.

3.1 Construction of PAG for simplified Java bytecode

A simplified Java bytecode statement is given by one of the following:

$s :=$ `x = y`	Simple assignment
$\mid$ `x =`$_A$ `new O`	Allocation at allocation site A
$\mid$ `x = y.f`	Field load
$\mid$ `x.f = y`	Field store
$\mid$ `x =`$_S$ `y.f(z)`	Method invocation at call site S

Liu et al. give rules for constructing the PAG for a similarly simplified language [2]; we extend and simplify these rules to create a call-site–sensitive PAG. So that the analysis may be generic in the domain of contexts $\mathbb{C}$, our rules require only that there is an initial context **empty** : $\mathbb{C}$ and an operation **merge** : $\mathbb{C} \times CS \to \mathbb{C}$, where CS is the domain of call sites; we adapt the framework for context-sensitivity given by Smaragdakis et al. [3], but develop rules specific to our simplified language.

Vertices in the PAG are given by one of the following:

$v :=$ $(C,$ `x`$)$	Variable plus context
$\mid$ $(C,$ `x.f`$)$	Variable field plus context
$\mid$ $(C, A_{\texttt{T}})$	Allocation site of type `T` plus context

We introduce a judgment $C \vdash S \rightsquigarrow E$ to indicate that, under context C, statement S generates edges E, where an edge is of the form $v_1 \to v_2$ for vertices v_1 and v_2. This judgment is given by the rules in Figure 4, all of which have no premises except Rule (V).

An edge $v_1 \to v_2$ can be read as "$pts(v_1)$ is a subset of $pts(v_2)$". Rules (I)–(IV) straightforwardly introduce these constraints based on assignment statements; rule (V) is more involved,

[1] `https://github.com/wala/WALA`

$$C \vdash \texttt{x = y} \quad \rightsquigarrow \{(C, \texttt{y}) \to (C, \texttt{x})\} \quad \text{(I)}$$

$$C \vdash \texttt{x} =_A \texttt{new T} \quad \rightsquigarrow \{(C, A_{\texttt{T}}) \to (C, \texttt{x})\} \quad \text{(II)}$$

$$C \vdash \texttt{x = y.f} \quad \rightsquigarrow \{(C, \texttt{y.f}) \to (C, \texttt{x})\} \quad \text{(III)}$$

$$C \vdash \texttt{x.f = y} \quad \rightsquigarrow \{(C, \texttt{y}) \to (C, \texttt{x.f})\} \quad \text{(IV)}$$

$$\frac{(\_, A_{\texttt{T}}) \in pts(C, \texttt{y}) \quad f_{\texttt{T}}(\texttt{this}, \texttt{z'}) = \{S; \texttt{return x';}\} \quad \textbf{merge}(C, CS) = C' \quad C' \vdash S \rightsquigarrow E}{C \vdash \texttt{x} =_{CS} \texttt{y.f(z)} \quad \rightsquigarrow \quad E \cup \{(C, \texttt{z}) \to (C', \texttt{z'}), (C, \texttt{y}) \to (C', \texttt{this}), (C', \texttt{x'}) \to (C, \texttt{x})\}} \text{ (V)}$$

If S is the entry point to a program, and $\textbf{empty} \vdash S \rightsquigarrow E$, then $E \subseteq E'$, where E' is the edges for the PAG. E' is the least set given by these rules.

Figure 4: Rules for constructing edges of the PAG

since Java is a language with dynamic dispatch. Rule (V) states that, for each type `T` that `y` could point to, edges are introduced between the argument `z` and the parameter `z'` of method $f_{\texttt{T}}$, and between the returned value `x'` and the target of the assignment `x`. Furthermore, the body S of $f_{\texttt{T}}$ is evaluated under the merged context C' to yield new edges E.

The interaction between the calculation of points-to sets and the construction of the PAG means that the incremental algorithm for calculating points-to sets must also be run as the PAG is constructed. This algorithm is adapted from Liu and Huang's, but modified to account for the contexts of each node.

3.2 Structure of implementation

Our program takes as input a Java program, which is compiled into an SSA-based bytecode IR using the WALA framework. Then, the bytecode IR is converted to the simplified instruction set used in this paper, discarding some bytecode instructions (including those related to throwing and handling exceptions, and those related to reflection). The original pointer-analysis graph is created for the simplified IR using Andersen's on-the-fly algorithm, adding context sensitivity using the client-provided **empty** and **merge** operations. To update the pointer-analysis graph incrementally, the client provides a set of changes consisting of additions and deletions of statements; the program then detects which edges are affected by the additions and deletions, and finally runs the incremental addition and deletion algorithms for those edges.

4 Evaluation

The goal of our evaluation is to observe the effect of different contexts on the efficiency of the incremental algorithm. To do this, we first observe some properties of the points-to graph for various contexts, and we then proceed to analyze the performance of the incremental algorithm under these contexts.

4.1 Effect of choice of context on points-to graph

We find that adding context to nodes does not necessarily increase the size of the PAG, but that increasing the length of the call string in the context tends to increase the number of nodes in the PAG while decreasing the average size of the points-to sets. While these effects are well

	0-context		1-context		2-context		3-context	
	Worst	Mean	Worst	Mean	Worst	Mean	Worst	Mean
Sunflow	2.01ms	0.260ms	0.231ms	0.070ms	0.278ms	0.070ms	0.412ms	0.100ms
Jython	141ms	4.64ms	70.4ms	1.87ms	1.98s	3.80ms	12.9s	16.9ms
H2	13.1ms	0.632ms	5.12ms	0.301ms	25.6ms	0.757ms	99.7ms	1.65ms
TSP	2.01ms	0.104ms	1.77ms	0.038ms	7.45ms	0.125ms	25.4ms	0.314ms
Eclipse	119ms	2.43ms	22.9ms	0.593ms	176ms	1.65ms	2.51s	12.0ms

Table 1: Addition time for incremental algorithm.

known in the literature [3], we reproduce the results to lay the basis for our discussion of the performance of the incremental algorithm.

In Figure 5, we report statistics related to the PAGs constructed for a series of benchmark programs. It may seem unintuitive that 1-call site–sensitive context has a smaller PAG than the 0-call site–sensitive context, but this effect is due to the interaction in Rule (V) between the construction of the PAG and the construction of the points-to sets. Because nodes for a function for type `T` are generated only if the target `y` of the invocation may point to an object of type `T`, increased precision can actually shrink the size of the PAG. After the initial benefit of the increased precision, a calling context of length 2 or 3 will cause the PAG to be larger (because of the combinatorial effect of the increased number of contexts).

Figure 5 also demonstrates the average size of the points-to sets for reachable nodes (i.e. nodes with a non-empty points-to set). Just looking at average size of points-to sets, it would appear that 3-call site–sensitive analysis is more imprecise, but this is only because the same heap item appears multiple times with different contexts. Shown in the third chart is the result of collapsing equivalence classes of nodes (i.e., ignoring context), whereby we can see that a more sensitive context increases precision in all cases (about 1.5 allocation sites per node).

4.2 Performance experiments

To test the performance of the incremental algorithm with different levels of context sensitivity, we first construct the original PAG for the program (reported in Table 3). Then, for each reachable statement in the program, we perform the following sequence of steps:

1. Delete the statement.
2. Calculate the edges removed by the statement deletion (which can be several, especially for method calls), and incrementally re-calculate points-to sets based on the deletion.
3. Re-add the edges removed by the statement deletion, and incrementally re-calculate points-to sets.

This experiment design allows us to measure the performance of addition deletion and to determine correctness (that addition restores the points-to sets to their pre-deletion state; and that each node's points-to set is exactly the union of its predecessors' points-to sets in the PAG). In Tables 1 and 2, we report the worst-case and mean time for updating the points-to sets following addition and deletion.

4.3 Interpretation of results

Most importantly, addition and deletion run much faster, on average, than reconstructing the PAG from scratch: in general, a few milliseconds as compared to a few seconds. In the worst

	0-context		1-context		2-context		3-context	
	Worst	Mean	Worst	Mean	Worst	Mean	Worst	Mean
Sunflow	4.18ms	0.150ms	0.772ms	0.025ms	1.00ms	0.039ms	4.20ms	0.080ms
Jython	167ms	1.29ms	58.5ms	0.467ms	43.4ms	0.115ms	90.4ms	0.298ms
H2	7.98ms	0.156ms	3.47ms	0.091ms	2.99ms	0.055ms	5.62ms	0.072ms
TSP	2.92ms	0.022ms	0.956ms	0.009ms	2.28ms	0.014ms	1.72ms	0.019ms
Eclipse	41.4ms	0.209ms	9.38ms	0.089ms	7.29ms	0.050ms	592ms	0.384ms

Table 2: Deletion time for incremental algorithm.

	0-context	1-context	2-context	3-context
Sunflow	51.0ms	9.0ms	14.0ms	18.0ms
Jython	62.1s	11.4s	7.00s	21.6s
H2	2.76s	0.437s	0.207s	0.477s
TSP	49.0ms	32.0ms	51.0ms	104ms
Eclipse	5.02s	0.642s	1.98s	5.29s

Table 3: Non-incremental construction of original PAG.

case, addition sometimes performs as badly as reconstructing the PAG (see the worst case for 3-contexts), but this is rare, as evidenced by the low mean time.

Because the increased precision of context sensitivity can actually reduce the size of the PAG for some choices of context, increased context sensitivity is sometimes seen to improve the performance of the incremental algorithm from the baseline 0-call site–sensitive performance. However, for more sensitive contexts (especially 3-call site–sensitive and greater), the increased PAG size begins to negatively affect performance. For a more sensitive context, the deletion of a single statement can affect more nodes (since the deletion will remove all edges on that variable, regardless of context).

5 Alternate algorithm: subgraph sharing

Our primary approach to context sensitivity involves duplicating each variable node in the graph once for each context through which it can be reached. Unfortunately, this results in a blowup in graph size, resulting in higher memory load. Here we present another novel algorithm which achieves incremental updates on context-sensitive points-to analysis by sharing graph nodes across multiple contexts rather than duplicating them for each context.

The algorithm we presented earlier works by altering the input to Liu and Huang's incremental update algorithm (the PAG) so that context-sensitivity may be achieved without changing the incremental update algorithm itself. The alternate algorithm we are about to present takes the opposite approach; keeping the graph construction relatively unchanged, and achieving context-sensitivity by changing how we propagate incremental updates through the graph.

The key philosophy is to recognize the boundaries between functions and to treat each function and its local variables as a self-contained subgraph which has a particular input/output behavior, rather than being characterized by its call sites. Figure 6 demonstrates what this looks like when applied to the example program in Figure 1. The difference between this and Liu and Huang's original approach is in that the algorithm recognizes `identity` as a functional unit whose outputs are based on its inputs. The red arrows represent the functional input-output behavior of `identity`, which are being independently applied to the two different sets of input

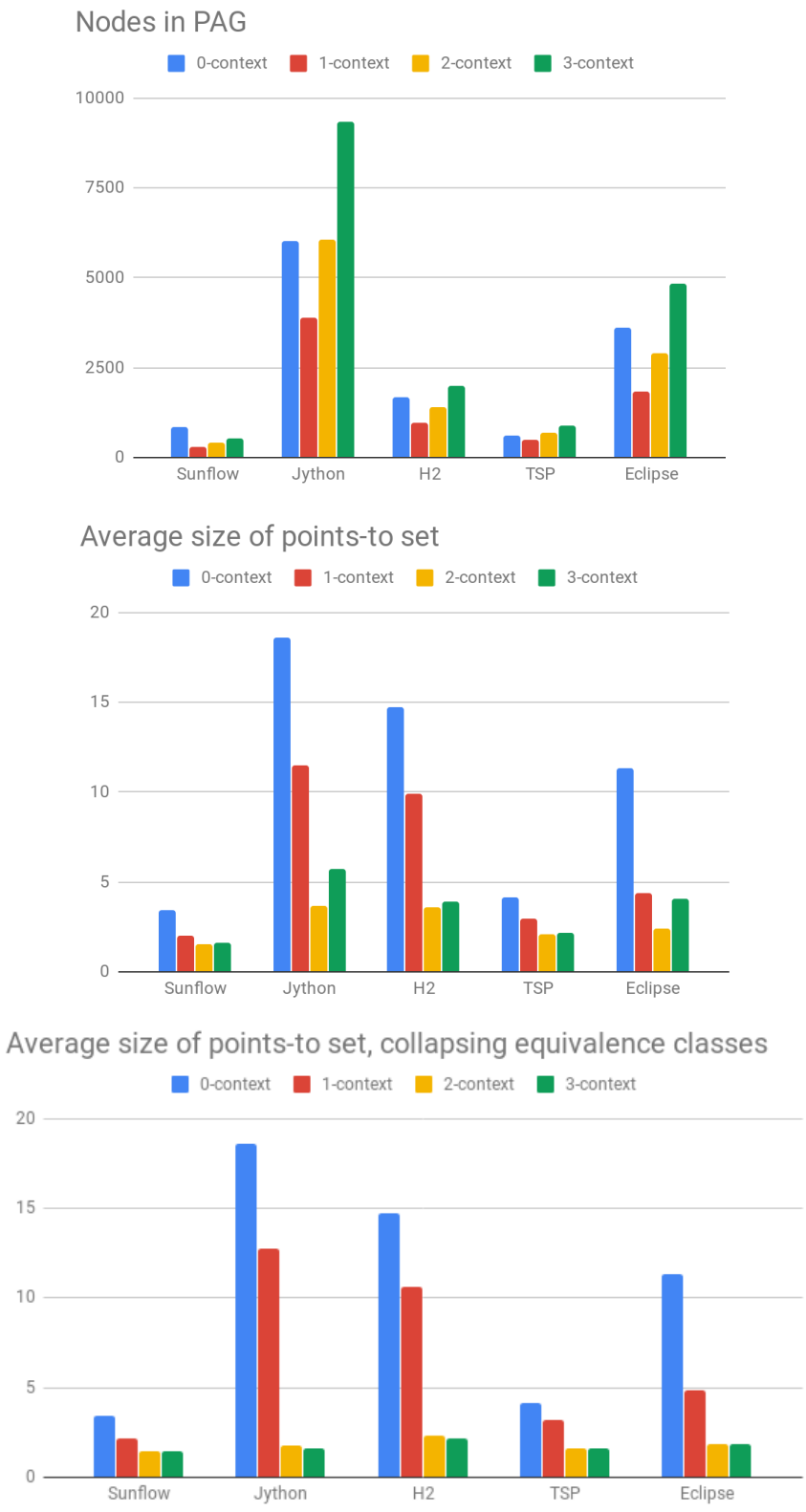

Figure 5: The number of nodes in the PAG and the average size of points-to sets under 0-, 1-, 2-, and 3-call site–sensitive contexts.

edges (in this case, each independent set of input edges contains just one edge).

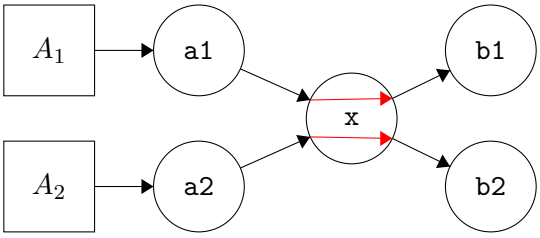

Figure 6: Context-sensitive node-sharing PAG for Figure 1

A subgraph corresponding to a function is recognized to have input edges (function arguments and global state which is read from) and output edges (function return values and global state which is written to). The algorithm infers the input-output behavior of the subgraph by recursively running the incremental points-to analysis algorithm on the function in isolation. In this isolated analysis, the points-to sets coming in along the input edges are represented with unique symbols. The points-to sets which the algorithm infers to send along the output edges will then be expressed in terms of these symbolic values. The relationship between the input symbols and the output expressions which use these symbols describes the behavior of the function.

As an example, we demonstrate how the input-output behavior of `identity` is determined. We let the symbol α represent the points-to set of the variable referred to by the argument `x`. The recursive run of the algorithm then infers that the returned output is α, which means the overall input-output behavior of `identity` is $\lambda\alpha.\alpha$. This local analysis is depicted in Figure 7. The isolated subgraph is demarcated with a dashed line; the abstract input is denoted with `[parameter]` and the abstract output is denoted with `[return]`. The abstract variable α is fed

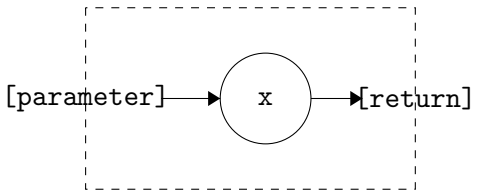

Figure 7: PAG used to identify input-output behavior of `identity`

in through `parameter`, and the analysis computes that α is output to `[return]`.

When a functional subgraph is reached while propagating incremental changes through a graph, rather than propagating changes to the nodes within the subgraph, the propagation directly continues at the output edges of the subgraph. The precise values to propagate to the output edges is determined by the inferred functional behavior of the subgraph. This invocation of the functional behavior is associated with the context of the calling site it is reached through.

When a function is updated, its functional behavior is updated and then changes are propagated to all of the outputs of its subgraphs accordingly. The first part is relatively simple; since we use the incremental algorithm for abstract behavior inference, we may update the functional behavior incrementally and efficiently. The second part is trickier; the node must store all inputs it's been invoked with.[2] Then when the functional behavior is updated, the new functional

[2]This is still more memory efficient than duplicating the subgraph for each context, as the memory footprint of the function scales only with the number of call sites and not with the number of nodes in the subgraph. This

behavior is applied to each input and the differences in outputs are propagated to each output of the subgraph.

6 Conclusions

We have utilized the techniques used in Liu and Huang's algorithm to design a novel algorithm which achieves context-sensitive pointer analysis while still supporting efficient incremental updates. We expected that this increased precision would increase the number of nodes in the PAG and result in worsened performance on incremental changes. To our surprise, when we implemented and evaluated our algorithm we found that the performance gains due to increased precision outweighed the performance losses. In most cases our algorithm is not only more precise than Liu and Huang's, but also more efficient.

We originally set out with the goal of adding specifically 1-call site sensitivity to Liu and Huang's algorithm, and we not only succeeded in this but also generalized their algorithm to arbitrary contexts. We were able to use our generic framework to evaluate the algorithm separately under 0-, 1-, 2-, and 3-call site-sensitivity. We found that while 1-call site sensitivity generally improves performance, there are diminishing performance returns to tracking more than the most recent call site and at this point the performance losses begin to outweigh. 1-call site sensitivity appears to be the "sweet spot" for performance.

We also designed another novel algorithm for context-sensitive incremental points-to analysis which has the same precision benefits as the one we implemented and has a smaller memory footprint.

7 Allocation of credit

We designate 50-50 credit between Nick and Vijay.

References

[1] B. Liu and J. Huang, "D4: Fast concurrency debugging with parallel differential analysis," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: ACM, 2018, pp. 359–373. [Online]. Available: http://doi.acm.org.proxy.library.cmu.edu/10.1145/3192366.3192390

[2] S. Zhan and J. Huang, "Echo: Instantaneous in situ race detection in the ide," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 775–786. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950332

[3] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," in *Foundations and Trends in Programming Languages*, 2015, pp. 1–69. [Online]. Available: http://dx.doi.org/10.1561/2500000014

is significant in functions with large bodies.