2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)

Demand-Driven Refinement of Points-to Analysis

Chenguang Sun
Purdue University
West Lafayette, USA
sun47@purdue.edu

Samuel Midkiff
Purdue University
West Lafayette, USA
smidkiff@ecn.purdue.edu

***Abstract*—We present DYNASENS, a demand-driven approach to points-to analysis that uses slicing to automatically adjust the analysis' context-sensitivity. Within a points-to analysis, heap-carried data flows are composed of loads and stores, and these heap-carried dependences are difficult to resolve. Having observed the limitations of existing techniques, we propose a slicing analysis based on a demand-driven approach to resolve such dependences. Given a points-to query, a collection of relevant program elements is identified by the slicing analysis and handled context-sensitively by the points-to analysis. We compare the precision and cost of our points-to analysis against two state-of-the-art uniformly context-sensitive analyses that achieve the best trade between cost and precision to date. Evaluation results shows the points-to analysis refined by the slicing analysis achieves higher precision in most tests than the uniformly context-sensitive analyses, which are many times more costly.**

***Index Terms*—demand-driven, alias analysis, slicing analysis, points-to analysis, context-sensitivity, Java**

I. MOTIVATION

Points-to analysis is one of the most important static program analyses. For object-oriented languages like Java, context-sensitivity is a common approach to achieve precision. However, precision gains from context-sensitivity always come with a high analysis cost. More recent work [1] formulated a parametric points-to analysis framework to support tuning the trade off between precision and cost for the entire program with manually specified heuristic rules.

Instead of targeting the entire program, we propose a slicing analysis to configure the framework with a subset of program elements relevant to answering a query from certain client analysis. Given the client query, the slicing analysis resolves the corresponding data flow dependence from which relevant program elements are extracted.

Among all data flow dependences, heap-carried dependences, where data flows through the heap via loads and stores, poses a major challenge. The existence of procedural side effects (i.e. a callee's effects on a caller via heap updates) exacerbates the challenge further.

When attempting to implement the slicing analysis with existing techniques, several limitations are observed:

1) Heap-carried dependence analysis over a global heap model causes *global* data flow where valuable context information is lost [2];
2) Call-string based context-sensitivity resorts to truncation [3] to remain bounded in the existence of recursive calls. With truncated call-strings as contexts, method calls behave as `gotos`, leading to precision loss [4];
3) Bottom-up summary-based interprocedural analysis must exhaustively model all callee side effects at all call sites, which can be prohibitively expensive [5];
4) On-demand intraprocedurally backward alias analysis cannot propagate interprocedurally, which precludes discovering some interprocedurally formed alias [6].

We propose a new slicing analysis for resolving interprocedural heap-carried dependences, while overcoming the four limitations stated above. In particular, the proposed analysis (1) is based on a local heap model; (2) builds a procedural alias effect on-demand; and (3) is interprocedurally bidirectional.

II. DYNASENS SLICING ANALYSIS

The DYNASENS slicing analysis is built over the same access-path-based local heap model as FLOWDROID. Each access path α is a local variable x followed by a field path δ (written $x.\delta$), and each field path δ is a (potentially empty) sequence of field names. The *effect* of a program on certain access path α is an alias class $[\alpha]$ – the set of all access paths may alias with α. Given an access path α as a *slicing criterion*, DYNASENS generates a *slice* – a subset of the program's elements – which preserves the effect of original program, i.e. generating the same $[\alpha]$. The membership relation $\beta \in [\alpha]$ is encoded by the fact $A(\alpha, \beta)$ The following rules are used to infer aliased access paths intraprocedurally:

$$s\text{: } x = y \begin{cases} A(\alpha, y.\delta) \text{ :- } A(\alpha, x.\delta), Assign(s, x, y). \\ A(\alpha, x.\delta) \text{ :- } A(\alpha, y.\delta), Assign(s, x, y). \end{cases}$$

$$s\text{: } x = y.f \begin{cases} A(\alpha, x.\delta) \text{ :- } A(\alpha, y.f.\delta), Load(s, x, y, f). \\ A(\alpha, y.f.\delta) \text{ :- } A(\alpha, x.\delta), Load(s, x, y, f). \end{cases}$$

$$s\text{: } x.f = y \begin{cases} A(\alpha, y.\delta) \text{ :- } A(\alpha, x.f.\delta), Store(s, x, f, y). \\ A(\alpha, x.f.\delta) \text{ :- } A(\alpha, y.\delta), Store(s, x, f, y). \end{cases}$$

A rule of the form "$A(\alpha, \beta')$:- $A(\alpha, \beta), ...$" can be interpreted as: The fact that α and β may alias implies that α and β' may alias. Thus given a set of base facts modeling all statements of a method and an initial trivial alias fact $A(\alpha, \alpha)$, the derived relation $Alias$ can be interpreted as an alias class $[\alpha]$ (i.e. $\{\beta \mid \langle \alpha, \beta \rangle \in Alias\}$) – the set of alias access paths implied by the effect of all statements of the method.

At method call s: $x = p(y_0, ..., y_k)$ where the declaration of the callee p is "$t\ \ p(t_0\ \ u_0, ..., t_k\ \ u_i)\{body\}$" and "$s'$: return z" $\in body$, a bottom-up approach is needed to propagate a callee's alias effect to callers (rule (1) to (4)). The propagated facts only contains part of callee's alias effect visible to the caller (similar to *effect masking* [7]). All rules are

2574-1934/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE-Companion.2019.00106

guarded by literals of the form $A(\alpha, \_)$ to guide the building of callee's summary (similar to *magic facts* [8]). The initial query $A(\alpha, \alpha)$ is generated at each call site (rule (5) and (6)).

$$A(\alpha, y_j.\delta_j) \text{ :- } A(\alpha, y_i.\delta_i), Arg(s, i, y_i), Arg(s, j, y_j), \quad (1)$$
$$Param(p, i, u_i), Param(p, j, u_j), A(u_i.\delta_i, u_j.\delta_j)$$
$$A(\alpha, x.\delta_2) \text{ :- } A(\alpha, x.\delta_1), RetTo(s, x),$$
$$RetFrom(p, z), A(z.\delta_1, z.\delta_2). \quad (2)$$
$$A(\alpha, x.\delta) \text{ :- } A(\alpha, y_i.\delta_i), Arg(s, i, y_i), RetTo(s, x),$$
$$Param(p, i, u_i), RetFrom(p, z), A(u_i.\delta_i, z.\delta). \quad (3)$$
$$A(\alpha, y_i.\delta_i) \text{ :- } A(\alpha, x.\delta), RetTo(s, x), Arg(s, i, y_i),$$
$$RetFrom(p, z), Param(p, i, u_i), A(u_i.\delta_i, z.\delta). \quad (4)$$
$$A(u_i.\delta_i, u_i.\delta_i) \text{ :- } A(\alpha, y_i.\delta_i), Arg(s, i, y_i), Param(p, i, u_i). \quad (5)$$
$$A(z.\delta, z.\delta) \text{ :- } A(\alpha, x.\delta), RetTo(s, x), RetFrom(p, z). \quad (6)$$

Smaragdakis et al. developed a parametric points-to analysis [9] which can be configured to enable different context sensitivities, such as call-site-sensitivity and object/type-sensitivity. Later, its configurability is extended to selectively allow manual specification of different context sensitivities on different program elements [1], so that a subset of the program elements are analyzed context-sensitively while the rest are analyzed context-insensitively. Such configurability comes from a parametric redesign of the analysis inference rules and the introduction of two new input relations to configure these rules: (1) ***ObjectToRefine***(s) which is a set of allocation sites (s) where context sensitivity is enabled; (2) ***SiteToRefine***(s, p) which is a set of call edges from call sites (s) to callees (p) where context sensitivity is enabled. These configuring input relations can be populated from the analysis result with the following rules:

$$ObjectToRefine(s) \text{ :- } A(\alpha, x.\delta), Alloc(s, x).$$
$$SiteToRefine(s, p) \text{ :- } A(\alpha, y_i.\delta), Arg(s, i, y_i).$$
$$SiteToRefine(s, p) \text{ :- } A(\alpha, x.\delta), RetTo(s, x).$$

III. Empirical Evaluation

We evaluated our analysis with two clients: (1) downcast safety checking and (2) copy constant propagation. The experimental results are compared with those of *2type+1H* and *1type1obj+1H* points-to analysis, named according to the naming convention in [9].

For downcast safety checking, the DaCapo benchmark suite [10] is used to evaluate DynaSens. For each downcast site in the benchmark program, the client analysis tries to prove its safety by configuring the points-to analysis with the result of the slicing analysis carried out with respect to the downcast. The program elements from the slicing result are configured with 2obj+1H context-sensitivity. DynaSens proves more casts safe than other analyses on most benchmarks (8 out of 10) and is very close to the most precise one on others. On the other hand, DynaSens has the desirable feature of low cost: its running time for any benchmark is close to the context-insensitive one. This advantage is more salient on large benchmarks where DynaSens incurs the lowest cost among all context-sensitive analyses. This is because context sensitivity is enabled only on elements relevant to each query.

For copy constant propagation, the Android framework is used to evaluate DynaSens. We apply constant propagation to infer message types denoted by integer values of the "`what`" field of `Message` objects. The precision is measured as the number of *safe* handlers, i.e. all possible message types handled by the handler are defined for it. Variants of DynaSens configured with different context-sensitivity of depth up to 4 are tested. The results are presented in Table I. Unlike uniformly context-sensitive analyses in previous work [11], whose cost grows exponentially in context depth, all cost metrics of our analysis grow much more slowly because of its accurate selective context sensitivity. Meanwhile, the precision continues to improve as the context depth becomes deeper, and all message handling sites are proven safe at depth 4, which would be intractable using a uniformly context sensitive analysis.

Analysis	Reachables	Call-Graph (K)	Points-To (K)	Safe	Time (sec)
Insen	19886	20/106	6494	0/95	147
2type+1H	18860	117/1795	7356	35/95	516
1type1obj+1H	18841	165/2264	10620	35/95	574
DynaSens (2obj+1H)	19717	39/179	6529	61/95	58 +214
DynaSens (3obj+2H)	19716	39/191	6603	61/95	58 +228
DynaSens (4obj+3H)	19716	40/189	6594	95/95	58 +231

TABLE I: Results for demand-driven copy constant propagation. The "reachables" rows show the number of reachable methods. The "call-graph (K)" rows show the number of thousands of nodes/edges within the context-sensitive call graph built. The "points-to (K)" rows show the number of thousands of pairs of a context-sensitive local variable and an object within the points-to relation built. The "time" rows show the run-time of the (slicing and) points-to analyses in seconds. The "Safe" column shows the number of safe handlers determined by the copy constant propagation analysis.

References

[1] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, "Introspective analysis: Context-sensitivity, across the board," ser. PLDI '14.

[2] P. Liang and M. Naik, "Scaling abstraction refinement via pruning," ser. PLDI '11.

[3] O. Shivers, "Control flow analysis in scheme," ser. PLDI '88.

[4] M. Sridharan and R. Bodík, "Refinement-based context-sensitive points-to analysis for java," ser. PLDI '06.

[5] E. M. Nystrom, H.-S. Kim, and W.-m. W. Hwu, "Bottom-up and top-down context-sensitive summary-based pointer analysis," R. Giacobazzi, Ed.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," ser. PLDI '14.

[7] J. M. Lucassen and D. K. Gifford, "Polymorphic effect systems," ser. POPL '88.

[8] T. W. Reps, "Solving demand versions of interprocedural analysis problems," ser. CC '94.

[9] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: Understanding object-sensitivity," ser. POPL '11.

[10] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis."

[11] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for java," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 1, Jan. 2005.