Context Transformations for Pointer Analysis

Rei Thiessen Ondřej Lhoták
University of Waterloo
{rthiesse,olhotak}@uwaterloo.ca

Abstract

Points-to analysis for Java benefits greatly from context sensitivity. CFL-reachability and k-limited context strings are two approaches to obtaining context sensitivity with different advantages: CFL-reachability allows local reasoning about data-value flow and thus is suitable for demand-driven analyses, whereas k-limited analyses allow object sensitivity which is a superior calling context abstraction for object-oriented languages. We combine the advantages of both approaches to obtain a context-sensitive analysis that is as precise as k-limited context strings, but is more efficient to compute. Our key insight is based on a novel abstraction of contexts adapted from CFL-reachability that represents a relation between two calling contexts as a composition of transformations over contexts.

We formulate pointer analysis in an algebraic structure of *context transformations*, which is a set of functions over calling contexts closed under function composition. We show that the context representation of context-string-based analyses is an explicit enumeration of all input and output values of context transformations. CFL-reachability-based pointer analysis is formulated to use call-strings as contexts, but the context transformations concept can be applied to any context abstraction used in k-limited analyses, including object- and type-sensitive analysis. The result is a more efficient algorithm for computing context-sensitive results for a wide variety of context configurations.

CCS Concepts • **Theory of computation → Program analysis**

Keywords Pointer analysis, context-sensitive analysis, static analysis

1. Introduction

Pointer analysis is a fundamental static analysis that determines the objects that pointers may point to. Precise pointer information is essential for program verification, refactoring tools, and other downstream static analyses. In order to compute precise pointer information, an analysis must account for different calling contexts of methods.

A method may have different run-time behaviour in each invocation. A context-insensitive analysis produces a single conservative result that models all invocations. A more precise analysis result can be obtained if methods are analyzed multiple times to model different calling contexts. A *method context* represents a set of run-time invocations of a method in some static and finite partitioning of all invocations of the method during an execution of a program.

Although context-insensitive pointer analysis scales to the largest of Java programs, context-sensitive pointer analyses fare less well. Various techniques have been proposed to improve the scalability of context-sensitive pointer analysis, including but not limited to using Binary Decision Diagrams to compress analysis data [8, 21, 25], merging of redundant pointer information [22], combining different flavours of context sensitivity [6], demand driven algorithms that progressively refine the precision of the analysis based on the needs of a client [17], and methods of varying the level of context sensitivity of program elements [9, 10, 16, 23].

We have developed a new context abstraction based on insights from the *context-free language reachability* (CFL-reachability) formulation of pointer analysis [13, 17, 18, 24]. In the CFL-reachability formulation of pointer analysis, points-to relationships are identified by paths in a graph representation of Java programs. The paths are filtered by strings formed by labels of traversed edges, which are required to be in the intersection of two context-free languages that model data flow across heap accesses and method calls. We identify these paths as *transformations* over contexts, and show that the traditional representation of context information is the explicit enumeration of input-output mapping pairs of these transformations. We introduce an alternative representation of context transformations that more efficiently represents data. Unlike the CFL-reachability approach which is formulated only for call-site sensitivity, our abstraction works under call-site [14], object [11], and type sensitivity [15].

The traditional representation of context information as context strings explicitly enumerates the method contexts under which points-to and call-graph relationships hold. For example, given the statement "`p = new T();`" labelled `h`,

PLDI'17, June 18–23, 2017, Barcelona, Spain
ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062359

a traditional context-string-based pointer analysis concludes that variable p *points-to* (an object allocated at) h, and this fact is enumerated for every method context of the method that contains the statement. Our new abstraction represents this information by a single fact: p *points-to* h *under the identity transformation over method contexts*, meaning that p points to an object allocated at h in the same method context in which the object was allocated. Context-sensitive call-graph edges, which express a relationship between caller and callee method contexts, can also be expressed as transformations over method contexts. Points-to facts arising from interprocedural data flow are computed through function composition of context transformations of points-to facts with that of call-graph edges. Thus, the new representation of context information is an algebraic structure of transformations over method contexts, closed under function composition.

We present a *parameterized* set of deduction rules for pointer analysis that can be instantiated with either the traditional representation of contexts as pairs of fixed-length strings (*context string abstraction*), or with our new abstraction. Our new abstraction is strictly more precise than context strings in theory, but obtains exactly the same precision in practice under call-site and object sensitivity. When the parameterized rules are *instantiated* to a particular context sensitivity and representation, they become plain Datalog [4] rules, and analysis results can be computed by fast Datalog engines.

Contributions. This paper makes the following contributions:

- We formulate a pointer analysis using a new algebraic structure of *context transformations*. We show that one representation of context transformations is in the form of k-limited context strings. We propose an alternative representation of context transformations as a composition of elemental transformations called *transformer strings*.
- We present a parameterized set of deduction rules for pointer analysis which can be instantiated to use either the traditional context string representation or our new representation. Similar to the DOOP framework [3], the rules can be instantiated as a call-site-, object-, or type-sensitive analysis.
- We present a systematic method of transforming the parameterized deduction rules into plain Datalog. We evaluate the efficiency and performance of our representation and technique on large programs in the DaCapo benchmark suite. The most precise context sensitivity configuration evaluated is 2-limited object-sensitive analysis with 1-limited heap abstraction, and our new transformer string abstraction obtains a geometric mean 29% reduction in the number of facts and 27% reduction in analysis time compared to the traditional context string representation.

```
class T {
  Object f;
  Object id(Object p) { return p; }
  Object id2(Object q) {
    Object t = id(q); // c1
    return t;
  }
  Object m() { return new T(); // m1 }
  public static void main(String[] args) {
    Object x = new Object(); // h1
    Object y = new Object(); // h2
    Object r = new T(); // h3
    Object x1 = r.id(x); // c2
    Object y1 = r.id(y); // c3
    Object s = new T(); // h4
    Object t = new T(); // h5
    Object x2 = s.id2(x); // c4
    Object y2 = t.id2(y); // c5
    T a = s.m(); // c6
    T b = t.m(); // c7
    a.f = x;
    Object z = b.f;
  }
}
```

Figure 1. Context sensitivity code example.

2. Background

There are two primary variations on context sensitivity for an object-oriented language: A *call-site-sensitive* analysis defines contexts by static call sites of invocations, while an *object-sensitive* analysis uses the heap allocation site of a receiver object to differentiate contexts [11, 14, 15]. *Type sensitivity* can be considered a subclass of object sensitivity, where heap allocation sites are replaced by class types that contain the methods containing the allocation sites [15].

The example Java code in Figure 1 illustrates the differences between the two types of context sensitivity. The example contains two identity methods id and id2, where id returns its parameter directly, and id2 indirectly by calling id. Heap objects are abstracted by their allocation sites: we say that variable x *points to* h1 to mean that the value of x, at run-time, may be the address of an object allocated at h1.

In a context-insensitive analysis, only one points-to set is maintained for the parameter p of the method id, and thus the analysis concludes that p points to objects allocated at h1 and h2. Therefore, x1 and y1 also point to h1 and h2. In a *1-call-site-sensitive* analysis, method id is analyzed in three different method contexts: the call sites corresponding to expressions id(x), id(y), and id(q). The analysis precisely deduces that x1 only points to h1 and y1 only points to h2. However, the call site of id(q) merges information from two different call sites of id2, and thus a 1-call-site analysis has imprecise points-to sets for x2 and y2. A 2-call-site analysis is required to obtain the most precise result for x2 and y2.

In a *1-object-sensitive* analysis, invocations of id using a receiver object allocated at h3 are analyzed in a single

context: that of heap allocation site h3. Thus, the points-to sets of x1 and y1 are imprecise: the analysis concludes that both variables could point to either h1 or h2. However, the invocations of id2 and id2's nested invocation of id are analyzed in two independent contexts: that of allocation sites h4 and h5. Thus, the points-to sets of x2 and y2 are precise: x2 points only to h1 and y2 points only to h2.

Heap contexts are required to differentiate objects returned by calls to m. Without heap contexts, an analysis concludes that a and b may point to heap objects allocated at m1 in any context of m. Thus, the analysis would imprecisely conclude that the heap accesses a.f and b.f are aliased, and that z may point to h1. Using one level of heap context, heap objects allocated at m1 are differentiated by the method contexts of m, which are c6 and c7 under call-site sensitivity and h1 and h2 under object sensitivity. Either flavour concludes that a and b do not point to a common object at run-time.

2.1 CFL-Reachability Formulation of Pointer Analysis

A *Pointer Assignment Graph (PAG)* is a graph representation of a program where nodes represent variables and heap allocation sites, and edges represent data flow through assignments. Figure 2 presents a simplified representation of a Java program containing only assignments, stores to and loads from fields of objects, heap allocations, and static invocations of methods. The PAG contains a node for each variable and for each heap allocation site in a program. Each statement in the program induces an edge in the PAG labelled as shown in the right column in the table. Interprocedural assignments are additionally labelled below the arrow by the call sites where the assignments occur. The label $\widehat{c}$ denotes that the assignment occurs at the start of an invocation from call site c, and $\check{c}$ denotes that the assignment occurs when returning from an invocation from c. Edges corresponding to a store or a load of a field have a label that includes the field that is accessed. Every edge and every label has a corresponding *backwards* equivalent: For every edge from x to y in the graph labelled l, let there be an edge from y to x in the graph labelled $\bar{l}$. Call site labels $\widehat{c}$ become $\check{c}$, and vice-versa. For example, in a program with edge $\mathtt{a}_1 \xrightarrow[\widehat{c}]{\mathsf{assign}} \mathtt{f}_1$, an implicit edge $\mathtt{f}_1 \xrightarrow[\check{c}]{\overline{\mathsf{assign}}} \mathtt{a}_1$ is present.

The *realized string* of a path is formed by concatenating the labels of traversed edges. Given a context-free language

Statement	Edge
`x = y;`	$\mathtt{y} \xrightarrow{\mathsf{assign}} \mathtt{x}$
`x.f = y;`	$\mathtt{y} \xrightarrow{\mathsf{store}[\mathtt{f}]} \mathtt{x}$
`x = y.f;`	$\mathtt{y} \xrightarrow{\mathsf{load}[\mathtt{f}]} \mathtt{x}$
`x = new T(); // h`	$\mathtt{h} \xrightarrow{\mathsf{new}} \mathtt{x}$

`x = T.m(`$\mathtt{a}_1,\ldots,\mathtt{a}_n$`); // c`
`static U m(`$\mathtt{f}_1,\ldots,\mathtt{f}_n$`)`
`{... return u; }`
	$\mathtt{a}_k \xrightarrow[\widehat{c}]{\mathsf{assign}} \mathtt{f}_k$
	$\mathtt{u} \xrightarrow[\check{c}]{\mathsf{assign}} \mathtt{x}$

Figure 2. Statements and their graph representations.

L, a path P is an *L-path* iff the realized string of P is in L. We use two distinct alphabets in our formulation, one for the labels above edges, and one for the call site labels below them. When we say P is an L-path, edge labels not in the alphabet of L are ignored when forming the string realized by P. An *all-pairs L-path problem* asks whether there exists an L-path from u to v for each pair of vertices u, v in a graph.

2.1.1 Intraprocedural Field-Sensitive Analysis

If a program consisted only of assignments, then pointer analysis would be a simple problem of computing the transitive closure over assign edges to establish data-flow paths from a heap allocation site to variables that may point to objects allocated at the site. Handling of heap accesses has been formulated as a CFL-reachability problem over a *balanced parentheses* language [17, 18]. An *indirect* data flow occurs between two variables y and z when the value of y is stored to a field of an object (e.g. "w`.f = `y`;`"), and the value of z is the result of loading the same field of the same object (e.g. "z `= `x`.f`", where w and x point to a common object). Thus, the store and the load form a conceptual pair of balanced parentheses. Variables w and x must point to the same object, which means there must be value-flow paths from a common allocation site to w and x. These paths in turn may involve indirect data flows, and thus a CFL is required to precisely handle heap accesses.

Let $\mathbf{\Sigma_F}$ be an alphabet used to define a language that models field loads and stores:

$$\mathbf{\Sigma_F} \equiv \{\mathsf{new}, \overline{\mathsf{new}}, \mathsf{assign}, \overline{\mathsf{assign}}, \mathsf{store}[f], \overline{\mathsf{store}}[f], \mathsf{load}[f], \overline{\mathsf{load}}[f] \mid f \in \mathbf{Field}\}.$$

$\mathbf{Field}$ is the set of all field signatures in a program. Let L_F be a language over $\mathbf{\Sigma_F}$ generated by the non-terminal flowsto defined by the following productions:

$$\begin{aligned}
\mathsf{flowsto} &\rightarrow \mathsf{new}\ \mathsf{flows}^*. \\
\overline{\mathsf{flowsto}} &\rightarrow \overline{\mathsf{flows}}^*\ \overline{\mathsf{new}}. \\
\mathsf{alias} &\rightarrow \overline{\mathsf{flowsto}}\ \mathsf{flowsto}. \\
\mathsf{flows} &\rightarrow \mathsf{assign} \mid \mathsf{store}[f]\ \mathsf{alias}\ \mathsf{load}[f]. \\
\overline{\mathsf{flows}} &\rightarrow \overline{\mathsf{assign}} \mid \overline{\mathsf{load}}[f]\ \mathsf{alias}\ \overline{\mathsf{store}}[f].
\end{aligned}$$

The flowsto non-terminal models the flow of values from heap allocation sites to variables.

In a context-insensitive points-to analysis, *x points-to h* iff there exists a L_F-path from h to x [18]. An *exhaustive* computation of context-insensitive points-to information is an all-pairs L_F-path problem from all heap allocation sites to all variables.

2.1.2 Context-Sensitive Analysis

Let $\mathbf{\Sigma_C}$ be an alphabet consisting of letters $\widehat{c}$ and $\check{c}$, where c ranges over $\mathbf{Inv}$, the set of all call sites of a program. Let L_C

be a language over $\mathbf{\Sigma}_{\mathcal{C}}$ generated by the non-terminal feasible defined by the following productions.

$$\begin{aligned}
\mathsf{balanced} &\rightarrow \widehat{c}\,\mathsf{balanced}\,\breve{c}.\\
\mathsf{balanced} &\rightarrow \mathsf{balanced}\ \mathsf{balanced} \mid \epsilon.\\
\mathsf{unbal_exits} &\rightarrow \breve{c}\,\mathsf{unbal_exits} \mid \epsilon.\\
\mathsf{unbal_entries} &\rightarrow \widehat{c}\,\mathsf{unbal_exits} \mid \epsilon.\\
\mathsf{feasible} &\rightarrow \mathsf{unbal_exits}\ \mathsf{balanced}\ \mathsf{unbal_entries}.
\end{aligned}$$

A path P is said to be *feasible* iff it is an L_C-path. An infeasible path characterizes data flow which cannot occur in practice: for example, data flow that enters a method from one call site and exits the method from a different call site. In a precise context-sensitive points-to analysis, x *points-to* h iff there exists a path from h to x that is both an L_F-path and an L_C-path. Computing this relation is an undecidable problem [13]. A computable analysis can be obtained by approximating one of the languages. One approach is to collapse all methods in a recursive call cycle into a single method [17]. Then L_C becomes a regular language, and thus $L_F \cap L_C$ is a context-free language.

2.2 Context-String-Based Analysis

Non-demand-driven algorithms for context-sensitive pointer analysis predominantly use a k-*limited* representation of method contexts, which are method contexts truncated to a finite length. Although the CFL-reachability approach uses only call sites as elemental pieces of context, context string formulations exist for a wide variety of contexts, such as call sites, heap allocation sites, class types, and combinations thereof [6].

We use the name $\mathbf{Ctxt}$ for the set of elemental contexts of a particular flavour of context sensitivity. For call-site-sensitive analysis, $\mathbf{Ctxt}$ is the set of call sites. For object-sensitive analysis, $\mathbf{Ctxt}$ is the set of heap allocation sites. For a type sensitivity analysis, $\mathbf{Ctxt}$ is the set of class types.

Context strings are representations of contexts as strings over $\mathbf{Ctxt}$, truncated to a finite length. The "top-most" context appears first: for example, in Figure 1, method `id` is invoked from call site `c1` in `id2`, and `id2` is in turn invoked from site `c4`. The context for `id` in a call-site-sensitive analysis for this sequence of invocations is the string $[\texttt{c1}, \texttt{c4}, \texttt{entry}]$, where `entry` is a special context for entry points in a program.

In an object-sensitive analysis, the method context for a non-static invocation is the heap context of the receiver object of the invocation prefixed with the allocation site of the object [11]. The heap context of an object is the method context in which the heap allocation occurred. For example, the receiver object of the invocation of `id2` at `c4` is a heap object allocated at `h4` in method context $[\texttt{entry}]$ (the special method context for entry points), and thus $[\texttt{h4}, \texttt{entry}]$ is the method context for the invocation of `id2`. The receiver object for the subsequent invocation of `id` inside `id2` stays the same, and thus `id` is invoked with the same method context of $[\texttt{h4}, \texttt{entry}]$. This is a variant of object sensitivity called *full object sensitivity* which contrasts with *plain object sensitivity* [15]. Under plain object sensitivity, the heap allocation site of a receiver object is prefixed to the method context of the invocation: in this example, `id` is invoked with the method context of $[\texttt{h4}, \texttt{h4}, \texttt{entry}]$ under plain object sensitivity. Full object sensitivity is the variant of object sensitivity used throughout this paper, because full object sensitivity has superior precision and analysis performance compared to plain object sensitivity [3, 15]. The method context of a static invocation is the same context as the method context in which the invocation occurred.

2.3 Notation

Function symbols are italicized and predicate symbols are in sans-serif font. The following string manipulation functions are used throughout this document: Let $\mathit{prefix}_i(s)$ be the prefix of string s of length $\mathit{min}(\|s\|, i)$. Let $\mathit{drop}_i(s)$ be the suffix of s of length $\|s\| - \mathit{min}(\|s\|, i)$.

3. Context Transformations

An interpretation of elements of $\mathbf{\Sigma}_{\mathcal{C}}$ is that they are *transformations* over contexts. For a given path, its realized string relates the context at the source of the path to the context at its target. For example, let P be an $L_F \cap L_C$-path from `h4` in `main` to `p` in `id` in Figure 1:

$$P \equiv \texttt{h1} \xrightarrow{\mathsf{new}} \texttt{x} \xrightarrow[\widehat{\texttt{c4}}]{\mathsf{assign}} \texttt{q} \xrightarrow[\widehat{\texttt{c1}}]{\mathsf{assign}} \texttt{p}.$$

The realized string of P over $\mathbf{\Sigma}_{\mathcal{C}}$ is $[\widehat{\texttt{c4}}, \widehat{\texttt{c1}}]$. The string can be interpreted as a function over method contexts that prefixes `c4`, then prefixes `c1` to its input. When the function is applied to context $[\texttt{entry}]$ of `main`, we obtain a context $[\texttt{c1}, \texttt{c4}, \texttt{entry}]$ for `id`.

Let P' be an $L_F \cap L_C$-path from `p` in `id` to `x2` in `main`:

$$P' \equiv \texttt{p} \xrightarrow[\breve{\texttt{c1}}]{\mathsf{assign}} \texttt{t} \xrightarrow[\breve{\texttt{c4}}]{\mathsf{assign}} \texttt{x2}.$$

Its realized string over $\mathbf{\Sigma}_{\mathcal{C}}$ is $[\breve{\texttt{c1}}, \breve{\texttt{c4}}]$. The string can be interpreted as a function over contexts that drops `c1` then drops `c4` from the front of its input. Applying it to context $[\texttt{c1}, \texttt{c4}, \texttt{entry}]$ of `id` yields $[\texttt{entry}]$ for `main`. We conclude that the path $P \cdot P'$ can be interpreted as an identity function.

We can deduce that if an object is allocated at site `h4` in a method context M, then `x2` will point to that object in M, and that this is true for all contexts M of `main`. In a traditional context-string-based analysis, the fact that `x2` in context M points-to `h4` allocated in context M is enumerated for all reachable contexts M of `main`. By representing context information as functions over contexts, this fact is compactly represented: that the context in which `x2` points to an object, and the context in which the object was allocated at `h4`, are related by the identity function over contexts.

We interpret any L_F-path as a transformation over method contexts, which may include feasible paths (L_C-paths) and infeasible paths (non L_C-paths). The following is an example of an infeasible L_F-path:

$$P'' \equiv \texttt{h1} \xrightarrow{\mathsf{new}} \texttt{x} \xrightarrow[\widehat{\texttt{c4}}]{\mathsf{assign}} \texttt{q} \xrightarrow[\widehat{\texttt{c1}}]{\mathsf{assign}} \texttt{p} \xrightarrow[\breve{\texttt{c1}}]{\mathsf{assign}} \texttt{t} \xrightarrow[\widehat{\texttt{c5}}]{\mathsf{assign}} \texttt{y2}.$$

We interpret P'' as a transformation that maps any method context to a special "error context" (denoted *err*) that indicates an infeasible path.

We will define a set of transformations over contexts as an algebraic structure. Let $\mathbf{T} \equiv \{\widehat{a}, \breve{a} \mid a \in \mathbf{Ctxt}\}$ be the set of *primitive context transformations*. Let the domain of method contexts be $\mathbf{Ctxts} \equiv \mathbf{Ctxt}^* \cup \{err\}$. The primitive transformations over $\mathbf{Ctxts}$ are defined as follows:

$$\widehat{a}(M) \equiv \begin{cases} a \cdot M & \text{if } M \neq err \\ err & \text{otherwise.} \end{cases}$$

$$\breve{a}(M) \equiv \begin{cases} M' & \text{if } M = a \cdot M' \\ err & \text{otherwise.} \end{cases}$$

Let the set of *context transformations*, denoted $\mathbf{CtxtT}$, be formed by the closure of $\mathbf{T}$ under function composition. Let ϵ be the identity transformation. We use a *postfix notation* for function composition: $f\,;g \equiv g \circ f$ means first apply f then g.

The set of context transformations is an *inverse semigroup*, which is a semigroup with unique inverses. Given a context transformation f, f^{-1} is its inverse in the semigroup sense, meaning that $f^{-1}\,;f\,;f^{-1} = f^{-1}$ and $f\,;f^{-1}\,;f = f$.

3.1 Pointer Analysis using Context Transformations

CFL-reachability problems correspond to *chain programs*, which are a restricted class of Datalog programs [12]. In this section, derivations of L_F-paths from heap allocation sites to variables are encoded as Datalog rules. This section assumes familiarity with basic logic programming and terminology.

Each deduction rule models a particular language construct, such as a field load or a method invocation, and derives points-to facts arising from the construct. Rules are motivated by exemplary dynamic executions of program constructs described by untruncated method contexts in which the executions occur. From the example executions, deduction rules are inferred that express context transformations appearing in the conclusion of rules as a function composition of context transformations that appear in the premises of rules.

We use the following notation to specify a set comprising transformations that map an untruncated method context to another:

$$[A \to B] \equiv \{T \in \mathbf{CtxtT} \mid T(A) = B\}.$$

Heap allocation sites and assignments. The following rule models heap allocations:

$$\frac{\mathsf{assign\_new}(H, Y, P), \mathsf{reach}(P, \_)}{\mathsf{pts}(Y, H, \epsilon)}$$

Predicate assign_new is an *input predicate*, which is a predicate that describes a program under analysis. The literal $\mathsf{assign\_new}(H, Y, P)$ indicates that in the program under analysis, the addresses of objects allocated at heap allocation site H are assigned to variable Y inside method P. reach is a *derived predicate*, which is a predicate representing a relation defined by rules. The literal $\mathsf{reach}(P, N)$ indicates that method P is reachable under some method context that has prefix N. (An anonymous variable "_" is used in place of N.) The literal $\mathsf{pts}(Y, H, A)$ indicates that Y points to H *under* the context transformation A, meaning if an object is allocated in method context M, then Y points to the object in context $A(M)$. Each predicate with a context transformation argument denotes flow from some source method to some destination method. The context transformation is interpreted as transforming the context of the source method to the context of the destination method. For the pts predicate, its context transformation's source method is the method containing allocation site H (denoted $parent(H)$), and its destination is the method declaring local variable Y ($parent(Y)$). The derived fact $\mathsf{pts}(Y, H, \epsilon)$ indicates that Y points to an object allocated at H in the same context as the context in which the object was allocated.

Intraprocedural assignments do not alter the contexts under which the points-to relationships hold:

$$\frac{\mathsf{pts}(Z, H, A) \quad \mathsf{assign}(Z, Y) \quad \texttt{// Y = Z;}}{\mathsf{pts}(Y, H, A)}$$

Field accesses of heap objects. Points-to relationships arising from loads of fields can be described as a sequence of events during an execution of a program: a *pointee* object is allocated at an allocation site H in context C_H; a *base* object is allocated at allocation site G in context C_G; variable W points to the pointee object in context C_{WX}, and variable X points to the base object in the same context C_{WX}; through W and X, the pointee object is stored to field F of the base object; variable Y points to the base object in context C_{YZ}; through Y, field F of the base object is loaded into a variable Z. Then, variable Z in context C_{YZ} points to the pointee object that was allocated at H in context C_H. Expressed in terms of pts facts, we must have $\mathsf{pts}(W, H, B)$, $\mathsf{pts}(X, G, C)$, and $\mathsf{pts}(Y, G, D)$ such that $B \in [C_H \to C_{WX}]$, $C \in [C_G \to C_{WX}]$, and $D \in [C_G \to C_{YZ}]$. Thus, $B; C^{-1} \in [C_H \to C_G]$, and thus $B; C^{-1}; D \in [C_H \to C_{YZ}]$. Thus, we derive the fact $\mathsf{pts}(Z, H, B; C^{-1}; D)$. From the above deductions, we infer the following rule:

$$\frac{\begin{array}{ll}\mathsf{pts}(W, H, B), \mathsf{pts}(X, G, C) & \\ \mathsf{store}(W, F, X) & \texttt{// X.F = W;} \\ \mathsf{pts}(Y, G, D) & \\ \mathsf{load}(Y, F, Z) & \texttt{// Z = Y.F;}\end{array}}{\mathsf{pts}(Z, H, B\,;C^{-1}\,;D)}$$

Parameter passing and return values. Constructing a PAG representation with interprocedural *assign* edges, as in Figure 2, requires a call-graph constructed ahead of time. On-the-fly construction of call-graphs is essential for precise points-to analysis in a language with predominantly dynamic dispatch of function calls [8]. Thus, parameter passing is performed with a derived predicate call, and rules that derive

it will be presented in following paragraphs. Let $\mathsf{call}(I, P, B)$ indicate that the (static or virtual) invocation I calls method P under context transformation B. The source and destination of B is $parent(I)$ and P, respectively. B transforms the context of a caller to the context of the callee. Points-to relationships arising from parameter passing and return values are handled by the following rules:

$$\frac{\mathsf{pts}(Z, H, B) \quad \mathsf{actual}(Z, I, O) \quad \mathsf{call}(I, P, C) \quad \mathsf{formal}(Y, P, O)}{\mathsf{pts}(Y, H, B\,;\,C)} \qquad \frac{\mathsf{pts}(Z, H, B) \quad \mathsf{return}(Z, P) \quad \mathsf{call}(I, P, C) \quad \mathsf{assign\_return}(I, Y)}{\mathsf{pts}(Y, H, B\,;\,C^{-1})}$$

Literal $\mathsf{actual}(Z, I, O)$ indicates that Z is the O^{th} actual of invocation I, and $\mathsf{formal}(Y, P, O)$ indicates that Y is the O^{th} formal of method P. Literal $\mathsf{return}(Z, P)$ indicates that variable Z is the return value of method P and $\mathsf{assign\_return}(I, Y)$ indicates that the return value of invocation I is assigned to Y.

Call-edge derivation under call-site sensitivity. Under call-site sensitivity, context transformations for static invocations are easily inferred: for each reachable method context M of a method P that contains a static invocation I of a method Q, the invocation invokes Q with context $I \cdot M$. Thus, we derive $\mathsf{call}(I, Q, \widehat{I})$:

$$\frac{\mathsf{static\_invoke}(I, Q, P), \mathsf{reach}(P, \_)}{\mathsf{call}(I, Q, \widehat{I})}$$

Handling virtual invocations is more difficult because the methods that they invoke depend on the receiver points-to relationship, and the points-to relationship is context-dependent. Suppose that a receiver object is allocated at site H in context C_H, and variable Z in method P points to the object in context C_Z. Then, we have $\mathsf{pts}(Z, H, B)$ such that $B \in [C_H \rightarrow C_Z]$. Suppose that a virtual invocation I invokes method Q using Z as its receiver variable. If we derive $\mathsf{call}(I, Q, \widehat{I})$, then an actual variable of I pointing to an object in a context outside the image of B would result in a formal of Q pointing to the object: an imprecise derivation because $\mathsf{pts}(Z, H, B)$ only indicates that Z points-to H in some method context in the image of B. Using the insight that $B^{-1}(X) = \mathit{err}$ if X is not in the image of B, we derive $\mathsf{call}(I, Q, B^{-1}; B; \widehat{I})$, which has the desired effect of passing actuals of I to formals of Q from the context C_Z, but not from method contexts outside the image of B. Thus, the following rule is inferred for deriving call-graph edges and their context transformations for virtual invocations:

$$\frac{\mathsf{virtual\_invoke}(I, Z, S) \quad \mathsf{pts}(Z, H, B) \quad \mathsf{heap\_type}(H, T), \mathsf{implements}(Q, T, S) \quad \mathsf{this\_var}(Y, Q)}{\mathsf{call}(I, Q, B^{-1}\,;\,B\,;\,\widehat{I})}$$

Literal $\mathsf{heap\_type}(H, T)$ indicates that T is the type of the objects allocated at H, and $\mathsf{implements}(Q, T, S)$ indicates that an invocation of a method signature S on a receiver object of type T dispatches to method Q.

Call-edge derivation under object sensitivity. The derivation of context transformations for call-graph edges under object-sensitive analysis is less intuitive than under call-site-sensitive analysis. In an object-sensitive analysis, if a static invocation I in method P invokes Q in some context, then Q is invoked with the same context [11]. Deriving $\mathsf{call}(I, Q, \epsilon)$ in this scenario is tempting, but is imprecise. Although I in context M invokes Q with context M for each reachable context M of P, the reverse is not true: reachable contexts of Q are not necessarily reachable contexts of P. The rule for handling return values would then derive points-to relationships through infeasible paths. If M is a reachable method context of P, then $\mathsf{reach}(P, N)$ must be true for some prefix N of M. Given a string $N \equiv n_1 \cdot \ldots \cdot n_k$, let $\breve{N} \equiv \widetilde{n_1}\,; \ldots ; \widetilde{n_k}$ and $\widehat{N} \equiv \widehat{n_k}\,; \ldots ; \widehat{n_1}$. The context transformation $\breve{N} \cdot \widehat{N}$ has the property that $(\breve{N} \cdot \widehat{N})(M) = M$ if N is a prefix of M and $(\breve{N} \cdot \widehat{N})(M) = \mathit{err}$ otherwise. Thus, the following rule is inferred for handling static invocations under object sensitivity:

$$\frac{\mathsf{static\_invoke}(I, Q, P), \mathsf{reach}(P, N)}{\mathsf{call}(I, Q, \breve{N} \cdot \widehat{N})}$$

Suppose that a receiver object is allocated at site H in context C_H, and variable Z in method P points to the object in context C_Z. Then, we have $\mathsf{pts}(Z, H, B)$ such that $B \in [C_H \rightarrow C_Z]$. Suppose that a virtual invocation I invokes method Q using Z as its receiver variable. Then the context of the invoked method is $H \cdot C_H$ under object sensitivity. A context transformation A is desired such that $A(C_Z) = H \cdot C_H$ and $A(X) = \mathit{err}$ for all X not in the image of B, but expressed in terms of B. The transformation $A = B^{-1}\,; \widehat{H}$ satisfies these requirements. Thus, we infer the following rule for handling virtual invocations under object sensitivity:

$$\frac{\mathsf{virtual\_invoke}(I, Z, S) \quad \mathsf{pts}(Z, H, B) \quad \mathsf{heap\_type}(H, T), \mathsf{implements}(Q, T, S) \quad \mathsf{this\_var}(Y, Q)}{\mathsf{call}(I, Q, B^{-1}\,;\,\widehat{H})}$$

4. Abstraction

Recursive call cycles in a program result in method contexts of unbounded length. A finite abstraction of context transformations requires some form of approximation. Due to approximation, abstractions of context transformations may map a single method context to multiple contexts, and thus the abstractions are defined as transformations over *sets* of method contexts.

4.1 Context Strings

Pairs of context strings used in traditional points-to analysis can be interpreted as representing context transformations. The input and output are *truncated* method contexts that

abstract all method contexts that have the truncated strings as prefixes. That is, a truncated string A abstracts all strings in $\{A \cdot A' \mid A' \in \mathbf{Ctxt}^*\}$.

Let $\mathbf{CtxtT}^{\mathbf{c}}_{i,j} \equiv \{(A, B) \mid A, B \in \mathbf{Ctxt}^*, \|A\| \leq i \wedge \|B\| \leq j\}$ be the domain of the context-string-based abstraction of context transformations, given integers i and j. Given a pair (A, B) in $\mathbf{CtxtT}^{\mathbf{c}}_{i,j}$, let the notation $\underline{(A, B)}$ be a transformation over $\mathcal{P}(\mathbf{Ctxt}^*)$. The transformation is defined in the following way:

$$\underline{(A,B)}(X) \equiv \begin{cases} B' & \text{if } A' \cap X \neq \emptyset, \\ & \quad \text{where } A' \equiv \{A \cdot C \mid C \in \mathbf{Ctxt}^*\} \\ & \quad \text{and } B' \equiv \{B \cdot C \mid C \in \mathbf{Ctxt}^*\} \\ \emptyset & \text{otherwise} \end{cases}$$

Relations used in pointer analysis use different truncation lengths for context strings. Parameters i and j of a domain $\mathbf{CtxtT}^{\mathbf{c}}_{i,j}$ define the truncation lengths of method contexts at the source and destination of context transformations. For example, pts relates the context in which an object allocation occurs to the context in which a variable points to the object, and thus, the context string abstraction domain for pts is $\mathbf{CtxtT}^{\mathbf{c}}_{h,m}$, where h is the truncation length of strings that qualify heap allocation sites, and m is the truncation length of method contexts and strings that qualify local variables. The call-graph relation call relates a caller method context to a callee method context, and thus uses the domain $\mathbf{Ctxt}_{m,m}$ to represent context information.

The context string representation necessitates explicit enumeration of the domain and range of context transformations in the deduction rule for points-to relationships arising from heap allocation sites:

$$\frac{\mathsf{assign\_new}(H, Y, P) \quad \mathsf{reach}(P, M)}{\mathsf{pts}(Y, H, (\mathit{prefix}_h(M), M))}$$

When compared to the original rule defined in terms of context transformations in Section 3, the redundancy present in the context string representation is apparent. In the next subsection, we present our representation of context transformations as a composition of primitive transformations.

4.2 Transformer Strings

This section introduces abstractions of context transformations as *transformer strings*.

Let $\mathbf{T}_{\mathcal{W}}$ be an alphabet that consists of elements of $\mathbf{T}$ and and a "wildcard" letter "$*$" that maps any non-empty set of contexts to a set consisting of all contexts. Given a letter a in $\mathbf{T}_{\mathcal{W}}$, let the notation $\underline{a}$ be a transformation over $\mathcal{P}(\mathbf{Ctxt}^*)$. The transformations are defined in the following way:

$$\widehat{\underline{a}}(X) \equiv \{a \cdot M \mid M \in X\}.$$
$$\breve{\underline{a}}(X) \equiv \{M \mid a \cdot M \in X\}.$$
$$\underline{*}(X) \equiv \begin{cases} \mathbf{Ctxt}^* & \text{if } X \neq \emptyset. \\ \emptyset & \text{otherwise.} \end{cases}$$

Transformer strings are strings in $\mathbf{Ts} \equiv \mathbf{T}^*_{\mathcal{W}} \cup \{\bot\}$. The special element $\bot$ maps any set of contexts to an empty set of contexts: $\underline{\bot}(X) \equiv \emptyset$. The conversion of transformer strings to transformations is defined as

$$\underline{a_1 \cdot \ldots \cdot a_n} \equiv \underline{a_1} \,;\, \ldots \,;\, \underline{a_n}.$$

An important notation is the conversion of strings over $\mathbf{Ctxt}$ to strings in $\mathbf{Ts}$. Given $M \equiv m_1 \cdot \ldots \cdot m_n \in \mathbf{Ctxt}^*$ let $\widehat{M}$ and $\breve{M}$ be defined as follows:

$$\widehat{M} \equiv \widehat{m_n} \cdot \ldots \cdot \widehat{m_1} \qquad \breve{M} \equiv \breve{m_1} \cdot \ldots \cdot \breve{m_n}$$

A source of confusion may be the reversal of letters when a string in $\mathbf{Ctxt}^*$ is converted into an entry context transformation. The advantage of this notation is that given a context transformation defined as a sequence of concatenations of strings (e.g. $\breve{A}\widehat{B}$), its function inverse can be obtained by reversing the sequence of concatenations and flipping entries into exits and vice-versa (e.g. $\breve{B}\widehat{A}$), without reversing the strings A and B.

A $\mathit{match} : \mathbf{Ts} \to \mathbf{Ts}$ function is defined that reduces the length of a transformer string without modifying its interpretation as a transformation:

$$\mathit{match}(A \cdot \widehat{a} \cdot \breve{a} \cdot B) = \mathit{match}(A \cdot B)$$
$$\mathit{match}(A \cdot \widehat{a} \cdot \breve{b} \cdot B) \equiv \bot \qquad (a \neq b)$$
$$\mathit{match}(A \cdot \widehat{a} \cdot * \cdot B) = \mathit{match}(A \cdot * \cdot B)$$
$$\mathit{match}(A \cdot * \cdot \breve{a} \cdot B) = \mathit{match}(A \cdot * \cdot B)$$
$$\mathit{match}(A \cdot * \cdot * \cdot B) = \mathit{match}(A \cdot * \cdot B)$$
$$\mathit{match}(\breve{A} \cdot * \cdot \widehat{B}) = \breve{A} \cdot * \cdot \widehat{B}$$
$$\mathit{match}(\breve{A} \cdot \widehat{B}) = \breve{A} \cdot \widehat{B}$$
$$\mathit{match}(\bot) = \bot$$

There is a degree of freedom in how match is applied to strings, but it is evident that all orderings of applications result in the same string. The following lemma establishes that the three non-recursive outputs of match, strings of the form $\breve{A} \cdot * \cdot \widehat{B}$, $\breve{A} \cdot \widehat{B}$, and $\bot$, are canonical representations of their inputs:

Lemma 4.1. *For all $A, B \in \mathbf{Ts}$, the following statements are true:*

1. $\underline{A} = \underline{\mathit{match}(A)}$.
2. $\underline{A} = \underline{B} \implies \mathit{match}(A) = \mathit{match}(B)$.

Let $\mathbf{CtxtT}^{\mathbf{t}} \equiv \{\breve{A} \cdot w \cdot \widehat{B} \mid A, B \in \mathbf{Ctxt}^*, w \in \{*, \epsilon\}\}$ be the domain of *untruncated canonical transformer strings*. Let $\mathbf{CtxtT}^{\mathbf{t}}_{i,j}$ be a subset of $\mathbf{Ctxt}^{\mathbf{t}}$ that consists of strings with at most i exits and at most j entries:

$$\mathbf{CtxtT}^{\mathbf{t}}_{i,j} \equiv \{\breve{A} \cdot w \cdot \widehat{B} \mid A, B \in \mathbf{Ctxt}^*, w \in \{*, \epsilon\}, \\ \|A\| \leq i \wedge \|B\| \leq j\}$$

Let $\mathit{trunc}_{i,j}$ be a *truncation function* that maps strings from $\mathbf{CtxtT}^{\mathbf{t}}$ to $\mathbf{CtxtT}^{\mathbf{t}}_{i,j}$.

$$\mathit{trunc}_{i,j}(\breve{A} \cdot w \cdot \widehat{B}) \equiv \begin{cases} \breve{A} \cdot w \cdot \widehat{B} & \text{if } \|A\| \leq i \wedge \|B\| \leq j \\ \breve{A_i} \cdot * \cdot \widehat{B_j} & \text{otherwise, where} \\ & \quad A_i \equiv \mathit{prefix}_i(A) \text{ and} \\ & \quad B_j \equiv \mathit{prefix}_j(B) \end{cases}$$

Note that $\breve{A_i} = \mathit{prefix}_i(\breve{A})$ and $\widehat{B_j} = \mathit{drop}_j(\widehat{B})$.

The following lemma states that truncation is conservative in that feasible paths are not discarded by truncation:

Lemma 4.2. *For all A in $\mathbf{CtxtT}^{\mathbf{t}}$, for all X in $\mathcal{P}(\mathbf{Ctxt}^*)$,*

$$\underline{A}(X) \subseteq \underline{trunc_{i,j}(A)}(X).$$

5. Pointer Analysis Deduction Rules

Figure 3 presents the *base* deduction rules for a simplified context-sensitive pointer analysis. Rules for static fields, class initialization, reflection, exceptions, and other language features are excluded from this presentation due to space constraints, but are present in the evaluated implementation. The rules are parameterized by definitions of the context abstraction domain $\mathbf{CtxtT}_{i,j}$, predicate comp, and functions inv, $target$, $record$, $merge$, and $merge\_s$. The base deduction rules are *instantiated* with the definitions in Figure 4.

There are three dimensions that characterize an instantiation: context transformation abstraction, flavour of context sensitivity, and levels of context sensitivity.

The abstraction of context transformations can be either context strings or transformer strings. The choice defines the domain $\mathbf{CtxtT}_{i,j}$ of abstractions of context transformations. Predicate comp and function inv are implementations of function composition and inverse in the abstraction domain. comp is a *function-style* predicate instead of a function to prevent the derivation of facts containing a $\underline{\perp}$ context transformation. Specifically, $\mathsf{comp}(A, B, C)$ is false for all C if $\underline{A}\,;\underline{B} \equiv \underline{\perp}$. Function $target$ converts abstractions of context transformations of call-graph edges to prefixes of reachable method contexts of callee methods.

Predicate comp and function inv are polymorphic with respect to their arguments: for example, the relation comp in the STORE rule in Figure 3 is a subset of $\mathbf{CtxtT}_{h,m} \times \mathbf{CtxtT}_{m,h} \times \mathbf{CtxtT}_{h,h}$, while in the PARAM rule, the relation is a subset of $\mathbf{CtxtT}_{h,m} \times \mathbf{CtxtT}_{m,m} \times \mathbf{CtxtT}_{h,m}$.

Functions $record$, $merge$, and $merge\_s$ allow the base deduction rules to be instantiated as either a call-site-, object-, or type-sensitive analysis. Their purpose is the same as in the DOOP framework [6]. $record$ converts method contexts to abstractions of context transformations for points-to relationships arising from a heap allocation site. $merge$ computes abstractions of context-transformations for call-graph edges of virtual invocation sites. $merge\_s$ computes abstractions of context-transformations for call-graph edges of static invocation sites. Let $classOf(H)$ of a heap allocation site H be the class type in which the method that contains H is implemented.

Parameters m and h define the *levels of method contexts* and *heap contexts*, respectively. Under the context string abstraction, strings are truncated to these lengths. For example, the pts predicate relates a heap allocation site to a variable, and thus its context representation is a pair consisting of a string truncated to length h and a string truncated to length m. This definition of levels of context sensitivity is consistent with context-string-based analyses in literature [6, 8, 15]. Under the transformer string abstraction, a transformer string for the pts predicate has at most h exits and m entries. The next section relates the precision of the two abstractions under the same levels of context sensitivity, and also contains an example of a bottom-up derivation of facts using both context string and transformer string abstractions.

6. Soundness and Precision

The *context-insensitive projections* of pts, hpts, and call, are relations with the context attribute projected out: for example, given a context-sensitive points-to relation pts, its corresponding context-insensitive relation $\mathsf{pts}^{\mathbf{ci}}$ is defined as $\mathsf{pts}^{\mathbf{ci}}(Y, H) \iff \exists A : \mathsf{pts}(Y, H, A)$. Note that in a transformer string instantiation, facts containing a transformer string "$\perp$" are never derived, and thus the presence of a pts, hpts, and call fact indicates the existence of a feasible dataflow path.

Theorem 6.1. *Transformer string instantiations are sound: that is, the context-insensitive projection of a transformer string instantiation is a superset of the context-insensitive projection of a context string analysis of unbounded length, for all levels and flavours of context sensitivity.*

Theorem 6.2. *Call-site- and object-sensitive transformer string instantiations are strictly more precise than context string instantiations at the same levels of method and heap contexts.*

Figure 5 illustrates a scenario where transformer strings are more precise than context strings using one level of heap and method contexts under call-site sensitivity. The first and second columns contain derived facts using the context string abstraction and the transformer string abstraction, respectively. The third column states the deduction rule used in the derivation. With context strings, the heap objects returned from call sites m1 and m2 are not differentiated. The difference in precision arises from the different implementations of the composition operation: composing transformer string ϵ with $\widehat{\texttt{id1}}$, and then with $\widehat{\texttt{id2}}$ results in ϵ (variable points to an object allocated in the same method context as the variable), but pair-wise composing context strings $\{(\texttt{m1}, \texttt{m1}), (\texttt{m2}, \texttt{m2})\}$ with $\{(\texttt{m1}, \texttt{id1}), (\texttt{m2}, \texttt{id1})\}$, and then with $\{(\texttt{id1}, \texttt{m1}), (\texttt{id1}, \texttt{m2})\}$ results in $\{(\texttt{m1}, \texttt{m1}), (\texttt{m1}, \texttt{m2}), (\texttt{m2}, \texttt{m1}), (\texttt{m2}, \texttt{m2})\}$.

The reason the precision theorem does not hold for type-sensitive analysis can be described informally by comparing the *implied* context information of transformer strings. A fact $\mathsf{pts}(Y, H, \widehat{a})$ implies that Y in method context $a \cdot M$ points to an object allocated at H in method context M, for all reachable method contexts M of the method that contains the invocation site a (under call-site sensitivity), or the method that contains the allocation site a (under object sensitivity). Under call-site and object sensitivities, the implied context information of transformer strings is no larger than that of the explicit enumeration of context information of context strings. However, under type sensitivity, the implied context

$$\frac{\mathsf{assign\_new}(H, Y, P) \quad \mathsf{reach}(P, M) \quad A \equiv \mathit{record}(M)}{\mathsf{pts}(Y, H, A)} \text{ [NEW]}$$

$$\frac{\mathsf{pts}(Z, H, A) \quad \mathsf{assign}(Z, Y)}{\mathsf{pts}(Y, H, A)} \text{ [ASSIGN]}$$

$$\frac{\mathsf{hpts}(G, F, H, B) \quad \mathsf{hload}(G, F, Y, C) \quad \mathsf{comp}(B, C, A)}{\mathsf{pts}(Y, H, A)} \text{ [IND]}$$

$$\frac{\mathsf{pts}(Z, H, B) \quad \mathsf{actual}(Z, I, O) \quad \mathsf{call}(I, P, C) \quad \mathsf{formal}(Y, P, O) \quad \mathsf{comp}(B, C, A)}{\mathsf{pts}(Y, H, A)} \text{ [PARAM]}$$

$$\frac{\mathsf{pts}(Z, H, B) \quad \mathsf{return}(Z, P) \quad \mathsf{call}(I, P, C) \quad \mathsf{assign\_return}(I, Y) \quad \mathsf{comp}(B, \mathit{inv}(C), A)}{\mathsf{pts}(Y, H, A)} \text{ [RET]}$$

$$\frac{\mathsf{pts}(X, H, B) \quad \mathsf{store}(X, F, Z) \quad \mathsf{pts}(Z, G, C) \quad \mathsf{comp}(B, \mathit{inv}(C), A)}{\mathsf{hpts}(G, F, H, A)} \text{ [STORE]}$$

$$\frac{\mathsf{pts}(Y, G, A) \quad \mathsf{load}(Y, F, Z)}{\mathsf{hload}(G, F, Z, A)} \text{ [LOAD]}$$

$$\frac{\mathsf{virtual\_invoke}(I, Z, S) \quad \mathsf{pts}(Z, H, B) \quad \mathsf{heap\_type}(H, T) \quad \mathsf{implements}(Q, T, S) \quad \mathsf{this\_var}(Y, Q) \quad C \equiv \mathit{merge}(H, I, B) \quad \mathsf{comp}(B, C, A)}{\mathsf{pts}(Y, H, A) \quad \mathsf{call}(I, Q, C)} \text{ [VIRT]}$$

$$\frac{\mathsf{static\_invoke}(I, Q, P) \quad \mathsf{reach}(P, B) \quad A \equiv \mathit{merge\_s}(I, B)}{\mathsf{call}(I, Q, A)} \text{ [STATIC]}$$

$$\frac{\mathsf{call}(I, P, A) \quad M \equiv \mathit{target}(A)}{\mathsf{reach}(P, M)} \text{ [REACH]}$$

$$\frac{}{\mathsf{reach}(\texttt{main}, [\texttt{entry}])} \text{ [ENTRY]}$$

Method context abstraction:

$\mathbf{CtxtM} \equiv \{M \in \mathbf{Ctxt}^* \mid \|M\| \leq m\}$.

Input predicates:

$\mathsf{actual} \subseteq \mathbf{Var} \times \mathbf{Inv} \times \mathbf{Nat}$
$\mathsf{assign} \subseteq \mathbf{Var} \times \mathbf{Var}$
$\mathsf{assign\_new} \subseteq \mathbf{Heap} \times \mathbf{Var} \times \mathbf{Method}$
$\mathsf{assign\_return} \subseteq \mathbf{Inv} \times \mathbf{Var}$
$\mathsf{formal} \subseteq \mathbf{Var} \times \mathbf{Method} \times \mathbf{Nat}$
$\mathsf{heap\_type} \subseteq \mathbf{Heap} \times \mathbf{Type}$
$\mathsf{implements} \subseteq \mathbf{Method} \times \mathbf{Type} \times \mathbf{MSig}$
$\mathsf{load} \subseteq \mathbf{Var} \times \mathbf{FSig} \times \mathbf{Var}$
$\mathsf{return} \subseteq \mathbf{Var} \times \mathbf{Method}$
$\mathsf{static\_invoke} \subseteq \mathbf{Inv} \times \mathbf{Method} \times \mathbf{Method}$
$\mathsf{store} \subseteq \mathbf{Var} \times \mathbf{FSig} \times \mathbf{Var}$
$\mathsf{this\_var} \subseteq \mathbf{Var} \times \mathbf{Method}$
$\mathsf{virtual\_invoke} \subseteq \mathbf{Inv} \times \mathbf{Var} \times \mathbf{MSig}$

Derived predicates:

$\mathsf{pts} \subseteq \mathbf{Var} \times \mathbf{Heap} \times \mathbf{CtxtT}_{h,m}$
$\mathsf{hpts} \subseteq \mathbf{Heap} \times \mathbf{FSig} \times \mathbf{Heap} \times \mathbf{CtxtT}_{h,h}$
$\mathsf{hload} \subseteq \mathbf{Heap} \times \mathbf{FSig} \times \mathbf{Var} \times \mathbf{CtxtT}_{h,m}$
$\mathsf{call} \subseteq \mathbf{Inv} \times \mathbf{Method} \times \mathbf{CtxtT}_{m,m}$
$\mathsf{reach} \subseteq \mathbf{Method} \times \mathbf{CtxtM}$

Macros:

$\mathsf{comp} \subseteq \mathbf{CtxtT}_{i,j} \times \mathbf{CtxtT}_{j,k} \times \mathbf{CtxtT}_{i,k}$
$\mathit{inv} : \mathbf{CtxtT}_{i,j} \to \mathbf{CtxtT}_{j,i}$
$\mathit{target} : \mathbf{CtxtT}_{m,m} \to \mathbf{CtxtM}$
$\mathit{record} : \mathbf{CtxtM} \to \mathbf{CtxtT}_{h,m}$
merge:$\mathbf{Heap} \times \mathbf{Inv} \times \mathbf{CtxtT}_{h,m} \to \mathbf{CtxtT}_{m,m}$
$\mathit{merge\_s} : \mathbf{Inv} \times \mathbf{CtxtM} \to \mathbf{CtxtT}_{m,m}$

Figure 3. Deduction rules for pointer-analysis. h and m specify levels of heap and method context sensitivity, respectively. $0 \leq h \leq m$ is assumed for call-site sensitivity and $0 \leq h = m - 1$ is assumed for object sensitivity.

Context string

$\mathbf{CtxtT}^{\mathbf{c}}_{i,j} \equiv \{(A, B) \mid A, B \in \mathbf{Ctxt}^*, \|A\| \leq i \wedge \|B\| \leq j\}$.
$\mathsf{comp}^{\mathbf{c}}((U, V), (V, W), (U, W))$.
$\mathit{inv}^{\mathbf{c}}((U, V)) \equiv (V, U)$.
$\mathit{target}^{\mathbf{c}}((U, V)) \equiv V$.

Call-site sensitivity:
$\mathit{record}^{\mathbf{c}}(M) \equiv (\mathit{prefix}_h(M), M)$.
$\mathit{merge}^{\mathbf{c}}(H, I, (\_, M)) \equiv (M, I \cdot \mathit{prefix}_{m-1}(M))$.
$\mathit{merge\_s}^{\mathbf{c}}(I, M) \equiv (M, I \cdot \mathit{prefix}_{m-1}(M))$.

Object sensitivity:
$\mathit{record}^{\mathbf{c}}(M) \equiv (\mathit{prefix}_h(M), M)$.
$\mathit{merge}^{\mathbf{c}}(H, I, (H', M)) \equiv (M, H \cdot H')$.
$\mathit{merge\_s}^{\mathbf{c}}(I, M) \equiv (M, M)$.

Type sensitivity:
$\mathit{record}^{\mathbf{c}}(M) \equiv (\mathit{prefix}_h(M), M)$.
$\mathit{merge}^{\mathbf{c}}(H, I, (H', M)) \equiv (M, \mathit{classOf}(H) \cdot H')$.
$\mathit{merge\_s}^{\mathbf{c}}(I, M) \equiv (M, M)$.

Transformer string

$\mathbf{CtxtT}^{\mathbf{t}}_{i,j} \equiv \{\breve{A} \cdot w \cdot \widehat{B} \mid A, B \in \mathbf{Ctxt}^*, w \in \{*, \epsilon\}, \|A\| \leq i \wedge \|B\| \leq j\}$.
$\mathsf{comp}^{\mathbf{t}}(X, Y, \mathit{trunc}_{i,k}(\mathit{match}(X \cdot Y))) \iff \mathit{match}(X \cdot Y) \neq \bot$.
$\mathit{inv}^{\mathbf{t}}(\breve{A} \cdot w \cdot \widehat{B}) \equiv \breve{B} \cdot w \cdot \widehat{A}$.
$\mathit{target}^{\mathbf{t}}(\breve{A} \cdot w \cdot \widehat{B}) \equiv B$.

Call-site sensitivity:
$\mathit{record}^{\mathbf{t}}(\_) \equiv \epsilon$.
$\mathit{merge}^{\mathbf{t}}(H, I, \breve{A} \cdot w \cdot \widehat{B}) \equiv \mathit{trunc}_{m,m}(\breve{B} \cdot \widehat{B} \cdot \widehat{I})$.
$\mathit{merge\_s}^{\mathbf{t}}(I, M) \equiv \widehat{I}$.

Object sensitivity:
$\mathit{record}^{\mathbf{t}}(\_) \equiv \epsilon$.
$\mathit{merge}^{\mathbf{t}}(H, I, \breve{A} \cdot w \cdot \widehat{B}) \equiv \breve{B} \cdot w \cdot \widehat{A} \cdot \widehat{H}$.
$\mathit{merge\_s}^{\mathbf{t}}(I, M) \equiv \breve{M} \cdot \widehat{M}$.

Type sensitivity:
$\mathit{record}^{\mathbf{t}}(\_) \equiv \epsilon$.
$\mathit{merge}^{\mathbf{t}}(H, I, \breve{A} \cdot w \cdot \widehat{B}) \equiv \breve{B} \cdot w \cdot \widehat{A} \cdot \widehat{\mathit{classOf}(H)}$.
$\mathit{merge\_s}^{\mathbf{t}}(I, M) \equiv \breve{M} \cdot \widehat{M}$.

Figure 4. Definitions of non-logical symbols in Figure 3 under different abstractions and flavours of context sensitivity.

```
class T {
  static T id(T p) { return p; }
  static T m() {
      T h = new T(); // h1
      T r = id(h); // id1
      return r;
  }
  public static void main(String[] args) {
    T x = m(); // m1
    T y = m(); // m2
  }
}
```

Context string	Transformer string	Rule
reach(main, entry)	reach(main, entry)	ENTRY
call(main, m, (entry, m1))	call(main, m, $\widehat{\texttt{m1}}$)	STATIC
call(main, m, (entry, m2))	call(main, m, $\widehat{\texttt{m2}}$)	STATIC
reach(m, m1)	reach(m, m1)	REACH
reach(m, m2)	reach(m, m2)	REACH
pts(h, h1, (m1, m1))	pts(h, h1, ϵ)	NEW
pts(h, h1, (m2, m2))		NEW
call(m, id, (m1, id1))	call(m, id, $\widehat{\texttt{id1}}$)	STATIC
call(m, id, (m2, id1))		STATIC
reach(id, id1)	reach(id, id1)	REACH
pts(p, h1, (m1, id1))	pts(p, h1, $\widehat{\texttt{id1}}$)	PARAM
pts(p, h1, (m2, id1))		PARAM
pts(r, h1, (m1, m1))	pts(r, h1, ϵ)	RETURN
pts(r, h1, (m2, m1))		RETURN
pts(r, h1, (m1, m2))		RETURN
pts(r, h1, (m2, m2))		RETURN
pts(x, h1, (m1, entry))	pts(x, h1, $\widetilde{\texttt{m1}}$)	RETURN
pts(x, h1, (m2, entry))		RETURN
pts(y, h1, (m1, entry))		RETURN
pts(y, h1, (m2, entry))	pts(y, h1, $\widetilde{\texttt{m2}}$)	RETURN

Figure 5. Example illustrating precision difference using $m = 1$ and $h = 1$ levels of contexts.

information of a fact $\mathsf{pts}(Y, H, \widehat{t})$ is that Y in method context $t \cdot M$ points to an object allocated at H in method context M, for all reachable method contexts M of *any method implemented in type t*: method reachability information is *merged* by the implied interpretation. Thus, the implied context information diverges from the context string abstraction where the method contexts of the *source* of a context transformation (in this case, the method containing H) and the *destination* (the method containing Y) are explicitly enumerated, and thus are not affected by the merging of implied method reachability information that occurs under type-sensitive analysis. Consequently, the transformer string abstraction is less precise than the context string abstraction under type sensitivity.

7. Implementation

This section describes the transformation of the deduction rules presented in Figure 3 into Datalog rules. Datalog evaluates rules *bottom-up*, meaning that the derived relations pts, hpts, hload, call, and reach are initially empty sets. New facts are derived by evaluating the literals that appear in premises of rules as relational joins. Evaluation of a Datalog program is complete when no new facts can be derived [4].

Derived relations are replaced by relations that are *specialized* to a particular representation. Specialized predicates of a context string instantiation have a $\mathbf{c}$ superscript, and a $\mathbf{t}$ superscript is used for transformer string instantiations. Under a context string instantiation, context transformation attributes represented by a pair of strings are simply flattened into two attributes: for example, $\mathsf{pts}(Y, H, (U, V))$ becomes $\mathsf{pts}^{\mathbf{c}}(Y, H, U, V)$. The resulting rules are equivalent to the rules found in the DOOP framework, and employ the same efficient indexing scheme. For example, when the definitions of $\mathsf{comp}^{\mathbf{c}}$ and $inv^{\mathbf{c}}$ are inlined into the rule STORE and variables are unified, the result is the familiar rule for inferring heap-points-to facts:

$$\frac{\mathsf{pts}^{\mathbf{c}}(X, H, U, V), \mathsf{store}(X, F, Z), \mathsf{pts}^{\mathbf{c}}(Z, G, W, V)}{\mathsf{hpts}^{\mathbf{c}}(G, F, H, U, W)}$$

The order of attributes may be confusing because *points-to* relates a pointer to a pointee, while context transformations relate a pointee context to a pointer context. For example, in $\mathsf{pts}^{\mathbf{c}}(X, H, U, V)$, V is a method context of X, and U is a heap context for H.

A naive method of implementing a transformer string instantiation is to implement the two formulas "$match(A \cdot B) \neq \bot$" and "$trunc_{i,k}(match(A \cdot B))$" of $\mathsf{comp}^{\mathbf{t}}$ as a procedural function that takes two values A and B as input, checks if $match(A \cdot B) \neq \bot$, and returns $trunc_{i,k}(match(A \cdot B))$. The performance of such an implementation is significantly slower than a context string instantiation. The reason for the lower performance can be understood by inspecting the relational joins performed by a bottom-up evaluation.

Consider the context string instantiation of the INDIRECT rule in Figure 3:

$$\frac{\mathsf{hpts}^{\mathbf{c}}(\boldsymbol{G}, \boldsymbol{F}, H, U, \boldsymbol{V}), \mathsf{hload}^{\mathbf{c}}(\boldsymbol{G}, \boldsymbol{F}, Y, \boldsymbol{V}, W)}{\mathsf{pts}^{\mathbf{c}}(Y, H, U, V)}$$

Evaluation of the above rule requires a join of relations $\mathsf{hpts}^{\mathbf{c}}$ and $\mathsf{hload}^{\mathbf{c}}$. The join is performed over three variables highlighted in bold: the abstract heap object G and its heap context V being loaded, and the field signature F of the field being loaded. A standard optimization performed by a Datalog engine is to build indices for the first, second, and fifth attributes of $\mathsf{hpts}^{\mathbf{c}}$, the first, second, and fourth attributes of $\mathsf{hload}^{\mathbf{c}}$, and to use these indices in the join.

A naive implementation of a transformer string instantiation is to leave the derived relations as is and have the Datalog engine evaluate the following rule:

$$\frac{\mathsf{hpts}^{\mathbf{t}}(\boldsymbol{G}, \boldsymbol{F}, H, B), \mathsf{hload}^{\mathbf{t}}(\boldsymbol{G}, \boldsymbol{F}, X, C) \quad match(B \cdot C) \neq \bot}{\mathsf{pts}^{\mathbf{t}}(Y, H, trunc_{h,m}(match(B \cdot C)))}$$

Variables B and C need to be bound to evaluate the formula "$match(B \cdot C) \neq \bot$". The join of $\mathsf{hpts}^{\mathbf{t}}$ and $\mathsf{hload}^{\mathbf{t}}$ is performed over two variables, G and F, instead of three variables

in the context string instantiation, and consequently indices over only two attributes are employed during evaluation. The less restrictive indexing scheme drastically increases the cost of the join, and thus the analysis time.

Our technique for obtaining a more efficient indexing scheme for transformer string instantiations is to decompose transformer strings into every possible *configuration*. A configuration of a transformer string specifies its number of exits, entries, and whether it has a wildcard letter. For example, the domain of transformer strings for the pts relation, $\mathbf{CtxtT}^{\mathbf{t}}_{h,m}$, in a 2-method-1-heap (that is, $m = 2$ and $h = 1$) call-site-sensitive instantiation, has 12 configurations arising from the following combinatorial choices: two choices for the number of exits, three choices for the number of entries, and two choices of whether the string contains a wildcard letter. Each relation with a context-transformer attribute (i.e. pts, hpts, call) is replaced by multiple specialized relations, one for each configuration. Specialized relations are tagged with subscripts that characterize a configuration: strings generated by the regular expression "$\mathbf{x}^*\mathbf{w}^?\mathbf{e}^*$", where the number of "$\mathbf{x}$" letters determines the number of exits, the appearance of a "$\mathbf{w}$" letter specifies that the transformer string contains a wildcard, and the number of "$\mathbf{e}$" letters determines the number of entries. The arity of a specialized predicate for a transformer string configuration is dependent on the number of entries and exits present in the transformer string. For example, $\mathsf{pts}^{\mathbf{t}}_{\mathbf{xxwe}}$ is a subset of $\mathbf{Var} \times \mathbf{Heap} \times \mathbf{Ctxt} \times \mathbf{Ctxt} \times \mathbf{Ctxt}$, and a fact $\mathsf{pts}(Y, H, \breve{X_1} \cdot \breve{X_2} \cdot * \cdot \widehat{E_1})$, becomes $\mathsf{pts}^{\mathbf{t}}_{\mathbf{xxwe}}(Y, H, X_1, X_2, E_1)$.

Each rule is duplicated for every possible replacement of derived predicates with specialized predicates. For example, in a 1-method-1-heap instantiation that has 8 configurations of transformer strings, the IND. rule is instantiated 64 times for each configuration of variables B and C in literals $\mathsf{hpts}(G, F, H, B)$ and $\mathsf{hload}(G, F, Y, C)$. One such instantiation for the $\mathbf{xe}$ configuration of B and C is as follows:

$$\frac{\mathsf{hpts}^{\mathbf{t}}_{\mathbf{xe}}(\boldsymbol{G}, \boldsymbol{F}, H, X_1, E_1) \quad \mathsf{hload}^{\mathbf{t}}_{\mathbf{xe}}(\boldsymbol{G}, \boldsymbol{F}, X_2, E_2) \quad \mathsf{comp}^{\mathbf{t}}(\breve{X_1} \cdot \widehat{E_1}, \breve{X_2} \cdot \widehat{E_2}, A)}{\mathsf{pts}(Y, H, A)} \tag{1}$$

The $\mathsf{comp}^{\mathbf{t}}$ predicate has a declarative specification: the third attribute can be computed for every possible transformer string configuration of the first two attributes of the predicate. For example, the following are a subset of $\mathsf{comp}^{\mathbf{t}}$ clauses of a 1-method 1-heap instantiation, where the first two attributes do not contain wildcards (there are 64 clauses in total):

$\mathsf{comp}^{\mathbf{t}}(\epsilon, \epsilon, \epsilon)$. $\quad \mathsf{comp}^{\mathbf{t}}(\breve{X}, \epsilon, \breve{X})$.
$\mathsf{comp}^{\mathbf{t}}(\epsilon, \widehat{E}, \widehat{E})$. $\quad \mathsf{comp}^{\mathbf{t}}(\breve{X}, \widehat{E}, \breve{X} \cdot \widehat{E})$.
$\mathsf{comp}^{\mathbf{t}}(\epsilon, \breve{X}, \breve{X})$. $\quad \mathsf{comp}^{\mathbf{t}}(\breve{X}, \breve{Z}, \breve{X} \cdot *)$.
$\mathsf{comp}^{\mathbf{t}}(\epsilon, \breve{X} \cdot \widehat{E}, \breve{X} \cdot \widehat{E})$. $\quad \mathsf{comp}^{\mathbf{t}}(\breve{X}, \breve{Z} \cdot \widehat{E}, \breve{X} \cdot * \cdot \widehat{E})$.
$\mathsf{comp}^{\mathbf{t}}(\widehat{E}, \epsilon, \widehat{E})$. $\quad \mathsf{comp}^{\mathbf{t}}(\breve{X} \cdot \widehat{E}, \epsilon, \breve{X} \cdot \widehat{E})$.
$\mathsf{comp}^{\mathbf{t}}(\widehat{Z}, \widehat{E}, * \cdot \widehat{E})$. $\quad \mathsf{comp}^{\mathbf{t}}(\breve{X} \cdot \widehat{Z}, \widehat{E}, \breve{X} \cdot * \cdot \widehat{E})$.
$\mathsf{comp}^{\mathbf{t}}(\widehat{M}, \breve{M}, \epsilon)$. $\quad \mathsf{comp}^{\mathbf{t}}(\breve{X} \cdot \widehat{M}, \breve{M}, \breve{X})$.
$\mathsf{comp}^{\mathbf{t}}(\widehat{M}, \breve{M}\widehat{E}, \widehat{E})$. $\quad \mathsf{comp}^{\mathbf{t}}(\breve{X} \cdot \widehat{M}, \breve{M} \cdot \widehat{E}, \breve{X} \cdot \widehat{E})$.

Literal $\mathsf{comp}^{\mathbf{t}}(\breve{X_1} \cdot \widehat{E_1}, \breve{X_2} \cdot \widehat{E_2}, A)$ in equation (1) unifies only with $\mathsf{comp}^{\mathbf{t}}(\breve{X} \cdot \widehat{M}, \breve{M} \cdot \widehat{E}, \breve{X} \cdot \widehat{E})$, using unifier $[X_1 \to X, E_1 \to M, X_2 \to M, E_2 \to E]$. The following is the resulting rule after unification:

$$\frac{\mathsf{hpts}^{\mathbf{t}}_{\mathbf{xe}}(\boldsymbol{G}, \boldsymbol{F}, H, X, \boldsymbol{M}) \quad \mathsf{hload}^{\mathbf{t}}_{\mathbf{xe}}(\boldsymbol{G}, \boldsymbol{F}, \boldsymbol{M}, E)}{\mathsf{pts}^{\mathbf{t}}_{\mathbf{xe}}(Y, H, X, E)}$$

The join of $\mathsf{hpts}^{\mathbf{t}}_{\mathbf{xe}}$ and $\mathsf{hload}^{\mathbf{t}}_{\mathbf{xe}}$ can now be performed over three common attributes, attaining the same indexing efficiency as the context string instantiation.

Functions inv, $target$, $record$, $merge$, and $merge\_s$ are inlined into rules using the same method as the inlining of predicate comp.

8. Evaluation

The Datalog engine used for the evaluation of our method is a research prototype that consists of two components: The front-end performs the instantiation of the base deduction rules in Figure 3 by inlining the functions and predicates in Figure 4. The output of the front-end is a plain Datalog program. The back-end compiles Datalog to native code using the LLVM Compiler Infrastructure [7].

The experiments were performed on a Intel i7-2600 processor with 16GiB of RAM. The Datalog engine is single-threaded. The analyzed programs are from the DaCapo benchmark suite (v.2006-10-MR2) under JDK 1.6.0_43 [2]. `jython` and `hsqldb` are not evaluated because context-sensitive analyses of the two programs do not scale due to overly conservative handling of Java reflection. `lusearch` is not evaluated because it is too similar to `luindex`. We use the same *fact generator* as DOOP, which transforms Java bytecode to a set of relations using the SOOT [20] framework.

We evaluate five different flavours of context sensitivity: 1-call, 1-call+H, 1-object, 2-object+H, and 2-type+H. The first number indicates the level of method contexts m, and "+H" indicates that $h = 1$ ($h = 0$ otherwise).

Figure 6 presents our experimental results. The first numbers in each column state the sizes of the context-sensitive pts, hpts, and call relations, the sum of the sizes of the three relations, and the analysis time using the context string abstraction. The time measurements do not include the time to perform the preprocessing steps of pointer analysis, such as reading the input relations from disk and constructing the virtual dispatch table, because the work performed is invariant with respect to different instantiations of our analysis. The preprocessing steps take less than 10 seconds for all benchmarks. The percentage number that follows is the decrease in relation size and analysis time using the transformer string abstraction. No reduction in the size of the hpts relation is present under 1-call and 1-object because the relation is context-insensitive (i.e. no heap contexts). Although transformer strings are theoretically more precise than context

		1-call		1-call+H		1-object		2-object+H		2-type+H		
antlr	pts	13.3M	6.4%	41.5M	14.1%	11.6M	11.3%	17.6M	29.2%	4.1M	20.1%	1031k(+3660)
	hpts	279k	—	2349k	32.0%	170k	—	368k	18.9%	206k	5.4%	87k(+1350)
	call	377k	15.6%	377k	15.5%	1885k	9.2%	4402k	25.4%	542k	27.8%	47k(+0)
	Total	13.9M	6.5%	44.2M	15.1%	13.6M	10.8%	22.4M	28.3%	4.8M	20.4%	
	Time	7.7s	6.2%	33.5s	1.3%	11.2s	0.9%	15.1s	18.6%	4.0s	17.5%	
bloat	pts	34.0M	3.1%	149.6M	8.4%	23.4M	5.9%	152.7M	4.0%	10.7M	-12.5%	1426k(+3740)
	hpts	475k	—	11802k	13.4%	429k	—	4028k	1.8%	526k	-43.9%	261k(+2121)
	call	559k	16.5%	559k	16.5%	2791k	6.0%	39212k	3.7%	1078k	7.4%	61k(+0)
	Total	35.1M	3.3%	161.9M	8.8%	26.6M	5.8%	195.9M	3.9%	12.3M	-12.1%	
	Time	20.8s	9.3%	149.7s	-36.3%	42.5s	10.9%	878.8s	-7.2%	11.1s	-53.6%	
chart	pts	50.0M	6.2%	115.1M	23.8%	65.9M	16.1%	56.1M	41.9%	11.5M	32.7%	882k(+3880)
	hpts	419k	—	4235k	44.4%	345k	—	721k	42.3%	431k	4.0%	143k(+3576)
	call	541k	17.4%	541k	17.4%	5094k	7.9%	15520k	49.5%	1379k	35.5%	63k(+13)
	Total	50.9M	6.3%	119.9M	24.5%	71.3M	15.4%	72.4M	43.6%	13.3M	32.1%	
	Time	27.2s	7.2%	87.9s	8.0%	157.6s	9.7%	92.9s	64.3%	11.4s	29.8%	
eclipse	pts	13.0M	7.9%	60.1M	17.5%	11.0M	9.3%	44.3M	30.1%	18.7M	17.9%	625k(+7476)
	hpts	205k	—	3722k	38.3%	136k	—	806k	28.3%	731k	5.3%	99k(+2421)
	call	433k	16.7%	433k	16.7%	1579k	9.2%	9757k	27.0%	2564k	14.3%	44k(+0)
	Total	13.6M	8.0%	64.2M	18.7%	12.7M	9.2%	54.9M	29.5%	22.0M	17.0%	
	Time	7.8s	11.6%	50.9s	-0.8%	14.0s	12.2%	58.1s	40.3%	21.2s	16.1%	
luindex	pts	8.3M	7.3%	25.7M	19.2%	6.2M	10.7%	10.5M	29.2%	3.3M	26.2%	353k(+3286)
	hpts	125k	—	1219k	34.8%	86k	—	248k	26.0%	179k	8.0%	64k(+1349)
	call	330k	14.4%	330k	14.4%	880k	10.7%	2711k	26.1%	527k	29.2%	36k(+0)
	Total	8.7M	7.4%	27.3M	19.9%	7.2M	10.6%	13.5M	28.5%	4.0M	25.8%	
	Time	4.9s	8.3%	19.6s	9.9%	6.8s	10.6%	9.8s	23.7%	3.9s	26.6%	
pmd	pts	11.9M	5.8%	35.4M	16.8%	8.8M	8.9%	13.6M	26.4%	3.9M	24.8%	460k(+3371)
	hpts	151k	—	1499k	33.5%	108k	—	443k	15.9%	298k	5.1%	80k(+1402)
	call	363k	14.4%	363k	14.4%	1117k	8.7%	3309k	23.6%	580k	27.5%	43k(+0)
	Total	12.4M	6.0%	37.3M	17.5%	10.1M	8.8%	17.3M	25.6%	4.8M	23.9%	
	Time	6.4s	8.0%	24.0s	5.3%	11.5s	9.0%	12.1s	21.1%	4.3s	23.3%	
xalan	pts	12.7M	6.2%	35.1M	16.3%	15.1M	7.5%	173.8M	40.0%	5.2M	27.9%	530k(+3159)
	hpts	243k	—	2176k	36.2%	183k	—	6053k	4.7%	336k	5.9%	156k(+1809)
	call	364k	14.3%	364k	14.3%	1866k	8.1%	49297k	30.4%	816k	30.3%	42k(+0)
	Total	13.3M	6.3%	37.7M	17.4%	17.2M	7.5%	229.2M	37.0%	6.3M	27.1%	
	Time	7.0s	10.3%	30.7s	1.3%	16.2s	7.5%	897.0s	2.3%	5.5s	22.9%	
Mean	Total		6.3%		17.5%		9.8%		28.9%		20.1%	
	Time		8.7%		-0.7%		8.8%		27.1%		14.9%	

Figure 6. Number of context-sensitive facts and percentage decrease from using the transformer string abstraction.

strings under call-site- and object-sensitive analysis, the two abstractions have *exactly* the same precision (compute the same sets of context-insensitive facts) when evaluated on this set of benchmark programs. Under type-sensitive analysis (column 2-type+H), the transformer string abstraction is less precise, and a third sub-column contains the number of context-insensitive pts, hpts, and call facts using the context string abstraction. The number in parentheses states the increase in the number of facts using the transformer string abstraction. The last two rows contain the geometric mean reduction in total relation sizes and analysis times.

In the instantiations where transformer strings are as precise as context strings (call-site and object sensitivity), the numbers of facts decrease across all benchmarks. The `chart` benchmark under 2-object+H analysis has the greatest decrease in the number of facts and analysis time.

In general, the decrease in analysis time is less than the decrease in the number of facts. This is due to the occurrence of *subsuming facts*: two facts are derived where the *concretization* (the implied context information of transformer strings as context strings) of one is a superset of the other. An example are facts $\mathsf{pts}(X, H, *)$, $\mathsf{pts}(X, H, \widetilde{M_1} \cdot *)$, $\mathsf{pts}(X, H, * \cdot \widehat{M_2})$, and $\mathsf{pts}(X, H, \widetilde{M_1} \cdot * \cdot \widehat{M_2})$. Fact $\mathsf{pts}(X, H, *)$ subsumes facts $\mathsf{pts}(X, H, A)$ for all A. Facts $\mathsf{pts}(X, H, \widetilde{M_1} \cdot *)$ and $\mathsf{pts}(X, H, * \cdot \widehat{M_2})$ subsume $\mathsf{pts}(X, H, \widetilde{M_1} \cdot * \cdot \widehat{M_2})$.

```
class T {
  Object f;
  void m() {
    Object v = new Object(); // h1
    if(...) {
      f = v;
      v = f;
    }
  }
  public static void main(String[] args) {
    Object t = new T(); // h2
    t.m(); // c1
  }
}
```

Transformer string	Rule
$\mathsf{reach}(\mathtt{main}, \mathtt{entry})$	ENTRY
$\mathsf{pts}(\mathtt{t}, \mathtt{h2}, \epsilon)$	NEW
$\mathsf{call}(\mathtt{c1}, \mathtt{m}, \widehat{\mathtt{c1}})$	VIRT
$\mathsf{pts}(\mathtt{this_m}, \mathtt{h2}, \widehat{\mathtt{c1}})$	VIRT
$\mathsf{reach}(\mathtt{m}, \mathtt{c1})$	REACH
$\mathsf{pts}(\mathtt{v}, \mathtt{h1}, \epsilon)$	NEW
$\mathsf{hpts}(\mathtt{h2}, \mathtt{f}, \mathtt{h1}, \widecheck{\mathtt{c1}})$	STORE
$\mathsf{hload}(\mathtt{h2}, \mathtt{f}, \mathtt{v}, \widehat{\mathtt{c1}})$	LOAD
$\mathsf{pts}(\mathtt{v}, \mathtt{h1}, \widecheck{\mathtt{c1}} \cdot \widehat{\mathtt{c1}})$	IND

Figure 7. Points-to relationships from multiple data-flow paths.

Figure 7 illustrates how subsuming facts may arise in a 1-call+H analysis. The variable v points to an object allocated at allocation site h1 through two data-flow paths, one *local* and one *context-dependent*: the first path is a direct assignment from the allocation site, resulting in an ϵ transformer string. The second path is through an instance field of the receiver object of the invocation of m, resulting in a $\widecheck{\mathtt{c1}} \cdot \widehat{\mathtt{c1}}$ transformer string. Since all invocations of m have a receiver object, $\mathsf{pts}(\mathtt{v}, \mathtt{h1}, \widecheck{C} \cdot \widehat{C})$ will be inferred for all method contexts C of m, resulting in the same explicit enumeration of contexts as the context string representation. Although $\mathsf{pts}(\mathtt{t}, \mathtt{h2}, \epsilon)$ is just one additional fact in the transformer string representation compared to the context string representation, all facts that can be derived using $\mathsf{pts}(\mathtt{v}, \mathtt{h1}, \widecheck{C} \cdot \widehat{C})$ for some C can also be derived using $\mathsf{pts}(\mathtt{t}, \mathtt{h2}, \epsilon)$ as well, doubling the amount of work performed by our Datalog engine.

The benchmark bloat suffers the most from subsuming facts that arise from multiple data-flow paths. A significant number of points-to facts in bloat belong to code that manipulates objects of an abstract syntax tree. Whenever a node n is allocated (the tree is constructed bottom-up), the children of n have their "parent" field set to n inside a method invoked from n's constructor, which results in heap-points-to facts with transformer strings of a "**we**" configuration under 1-call+H analysis (because n was passed as a parameter through multiple invocations). Thus, loading n from the "parent" field results in points-to facts with transformer strings of a "**we**" configuration. n is also passed as a parameter to a push call of a stack data structure. The receiver variable for the push call points to an object with transformer strings of a "**xwe**" configuration. Thus, loading n from the data structure also results in points-to facts with transformer strings of a "**xwe**" configuration. Variables pointing to n do so through data-flow paths (arising partly due to imprecision inherent to a 1-call+H analysis) through both the "parent" field and through the stack data structure, resulting in a large number of subsuming facts between the two configurations, which leads to an increase in the analysis time in the 1-call+H analysis of bloat.

One method of reducing the performance penalty of subsuming facts may be to customize the Datalog engine to delete subsumed facts from its database. For example, whenever a fact $\mathsf{pts}(y, h, * \cdot \widehat{c})$ is derived, facts $\mathsf{pts}(y, h, \widecheck{X} \cdot * \cdot \widehat{c})$ for all strings X may be deleted from the database without affecting the derivation of facts through feasible data-flow paths. We did not purse this direction due to the technical complexity of implementing such a feature in our Datalog engine.

The decrease in precision when type-sensitive analysis is performed using the transformer string abstraction is marginal: a geometric mean increase of 0.7% and 1.8% in the number of context-insensitive pts and hpts facts, respectively. Only the chart benchmark has an increase in the number of context-insensitive call-edges. In bloat, the precision loss inherent in the transformer string abstraction under type sensitivity results in a large increase in the context-sensitive heap-points-to relation, with a resulting increase in analysis time.

9. Related Work

Our deduction rules are adapted from the rules in the DOOP Framework for Java Pointer Analysis [3]. DOOP supports various flavours of context sensitivity, including call-site, object, type sensitivity, and combinations thereof [6]. DOOP uses the proprietary Datalog engine LogicBlox [5]. Our exception analysis, reflection analysis, and handling of native methods are straight translations of DOOP's rules written in LogicBlox's dialect of Datalog to the dialect of our Datalog engine.

Sridharan and Bodík proposed a CFL-reachability-based demand-driven context-sensitive analysis for Java [17]. Their analysis incorporates two approximations: recursive methods are handled context-insensitively and field accesses are initially assumed to alias without checking whether they access a common object. Their *refinement* technique attempts to increase precision by gradually removing the second assumption until a client of the analysis is satisfied by answers to a given alias query. They build a context-sensitive call-graph and their analysis is call-site-sensitive.

Xu and Rountev presented an analysis that reduces the complexity of context-sensitive pointer analysis through a technique similar to the one used in our analysis [22]. They identify a *flowing point* of a points-to fact, which is

a method where cloning points-to facts into the callers of the method results in redundant context information. In our analysis, given a points-to fact $\texttt{pts}(Y, H, \widetilde{A} \cdot \widehat{B})$ of a call-site-sensitive instantiation, the parent method of the last element of A is the flowing point as defined by Xu and Rountev. Their analysis is implemented as a procedural algorithm that inlines the points-to graphs of callee methods into their callers. Allocation sites are merged when the *last-k-substring* of their context strings match. The theoretical precision difference of their context abstraction and k-limited context strings is difficult to characterize. Our contribution is that we formally define an algebraic structure of transformer strings which does not enumerate redundant context information, and show that a common set of base deduction rules can be instantiated with either the context string or transformer string abstractions, into efficient Datalog programs. Furthermore, we establish the theoretical precision difference between the two abstractions.

Tan *et al.*'s analysis uses the result of a pre-analysis to construct an *object allocation graph* [19]. Similar to how paths in a call-graph form the reachable method contexts of a call-site-sensitive analysis, paths in an object allocation graph form the reachable method contexts of an object-sensitive analysis. Using this graph, redundant context elements are identified: nodes in the graph that can be merged without merging distinct paths. Thus their analysis attains a higher precision for a given truncation length of strings of contexts. Their use of the word "redundant" to describe context elements differ from how we describe the explicit enumeration of method contexts as redundant. The elimination of redundant context elements in Tan *et al.*'s analysis improves precision while the elimination of explicit enumeration in our analysis primarily improves performance.

Binary decision diagrams (BDDs) have been extensively studied as a technique for improving the scalability of context-sensitive pointer analysis. The ability of BDDs to merge redundant context information is heavily dependent on a chosen *variable ordering*. A variable ordering that minimizes the number of BDD nodes used to represent the points-to relation has been experimentally determined to yield the best performance. A consequence of this choice is that although the facts-to-BDD-nodes ratio for the points-to relation can be as low as 100:1 (indicating a very high level of compression), the ratio for other relations, such as the call-graph edge relation, can be as high as 1:8 [3]. The choice to optimize variable ordering for the points-to relation is based on the observation that for call-site-sensitive analyses, and for object-sensitive analyses with less than two method contexts, points-to facts greatly outnumber other inferred facts. For example, in a 1-object-1-heap analysis of the `luindex` benchmark, non-points-to facts constitute less than 15% of all inferred facts. The highest level of object sensitivity in which BDD-based algorithms have scaled is 1-object-1-heap analysis. There is a peculiar change in relation sizes between 1-object-1-heap and 2-object-1-heap analysis. The size of the context-sensitive points-to relation *decreases* in size by approximately a third, which is surprising because an exponential increase is typically expected when increasing the level of context sensitivity. Moreover, the size of the context-sensitive call-graph relation increases three-fold. The proportion of non-points-to facts to all inferred facts doubles to approximately 30%. Thus, the choice of relation to use to optimize the variable ordering becomes less clear-cut. In contrast, the transformer string abstraction decreases the sizes of all relations, and the reduction is most pronounced in the 2-object-1-heap analysis, which is presently the cutting-edge analysis for Java in terms of precision that scales to moderately sized programs.

10. Conclusion and Future Work

We have presented a formulation of pointer analysis based on an algebraic structure of context transformations, where the predominant abstraction of contexts, that of context strings, is shown to be one representation of transformations. The transformer string abstraction is proposed as an alternative representation that empirically has less redundancy than the context string abstraction. Less redundancies allow precise context-sensitive analysis to take less time and memory.

A direction of future work is to evaluate the efficiency difference between the context string and transformer string abstractions under demand-driven workloads. Datalog programs that exhaustively compute information can be converted to a demand-driven program through the magic sets transformation [1]. There may be synergy between demand-driven workloads and the transformer string abstraction's ability to represent local pointer information of a method without enumerating all reachable contexts of the method.

Acknowledgments

This research was supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '86, pages 1–15, New York, NY, USA, 1986. ACM.

[2] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Lan-*

guages, and Applications, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[3] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 243–262, New York, NY, USA, 2009. ACM.

[4] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, March 1989.

[5] Todd J. Green, Molham Aref, and Grigoris Karvounarakis. LogicBlox, platform and language: A tutorial. In *Proceedings of the Second International Conference on Datalog in Academia and Industry*, Datalog 2.0'12, pages 1–8, Berlin, Heidelberg, 2012. Springer-Verlag.

[6] George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 423–434, New York, NY, USA, 2013. ACM.

[7] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[8] Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Transactions on Software Engineering and Methodology*, 18(1):3:1–3:53, October 2008.

[9] Percy Liang and Mayur Naik. Scaling abstraction refinement via pruning. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 590–601, New York, NY, USA, 2011. ACM.

[10] Percy Liang, Omer Tripp, and Mayur Naik. Learning minimal abstractions. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 31–42, New York, NY, USA, 2011. ACM.

[11] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, January 2005.

[12] Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-–12):701–726, 1998.

[13] Thomas Reps. Undecidability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems*, 22(1):162–186, January 2000.

[14] Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Citeseer, 1991.

[15] Yannis Smaragdakis, Martin Bravenboer, and Ondřej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 17–30, New York, NY, USA, 2011. ACM.

[16] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 485–495, New York, NY, USA, 2014. ACM.

[17] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 387–400, New York, NY, USA, 2006. ACM.

[18] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 59–76, New York, NY, USA, 2005. ACM.

[19] Tian Tan, Yue Li, and Jingling Xue. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *International Static Analysis Symposium*, pages 489–510. Springer, 2016.

[20] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[21] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM.

[22] Guoqing Xu and Atanas Rountev. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 225–236, New York, NY, USA, 2008. ACM.

[23] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. On abstraction refinement for program analyses in Datalog. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 239–248, New York, NY, USA, 2014. ACM.

[24] Xin Zheng and Radu Rugina. Demand-driven alias analysis for C. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 197–208, New York, NY, USA, 2008. ACM.

[25] Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 145–157, New York, NY, USA, 2004. ACM.