Efficient Pointer Analysis of Java in Logic

by

Rei Thiessen

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2017

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner	Jingling Xue Scientia Professor
Supervisor	Ondřej Lhoták Associate Professor
Internal Member	Peter Buhr Associate Professor
Internal Member	Brad Lushman Lecturer
Internal-external Member	Derek Rayside Assistant Professor

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, which 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 pointer information for a wide variety of context configurations.

Acknowledgements

First and foremost, I would like to thank my supervisor, Ondřej Lhoták, for his guidance, patience, and support during my study. His mentorship was vital to my success.

I would like to thank Peter Buhr, Brad Lushman, Derek Rayside, and Jingling Xue for serving on my Examining Committee and providing valuable feedback and guidance.

I would also like to thank Magnus Madsen for his support and for the numerous research discussions we had. You made the last few years of my study much more enjoyable. My gratitude goes to Marianna Rapoport for organizing lively lab events and to my fellow graduate students in the PLG lab, past and present, for the many enjoyable discussions.

I am grateful for the funding provided by the University of Waterloo and NSERC.

Last but not least, I would like to thank my parents for their encouragement.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

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. More precise pointer information 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 [44, 49, 19], merging of redundant pointer information [46], combining different flavours of context sensitivity [16], demand-driven algorithms that progressively refine the precision of the analysis based on the needs of a client [36], and methods of varying the level of context sensitivity of program elements [35, 47, 20, 21].

The most extensively studied approach to context-sensitive analysis is the abstraction of contexts as fixed-length (k-limited) strings called *context strings*. Two families of context-string-based analyses are *call-site-sensitive* and *object-sensitive analysis*: a call-site-sensitive analysis forms method contexts by the series of call sites that invokes a method; an object-sensitive analysis forms method contexts out of the *heap context* of receiver objects. The heap context of an object is the method context in which the object is allocated. A disadvantage of context strings is that there is a high amount of redundancy in its representation of contexts. Local flow of pointer information that is invariant with respect to a method's caller (that is, the same points-to relationships hold in all calling contexts of the method) is represented by explicit duplication of the information for all calling contexts. One approach to improving the performance of context-sensitive pointer analysis is the use of *Binary Decision Diagrams (BDDs)* [6] to represent pointer information [44, 49, 19]. BDDs are compressed representations of sets and can efficiently represent large sets that contain duplicated information. This thesis proposes an alternative strategy: the usage of an abstraction that intrinsically does not duplicate invariant points-to relationships.

We have developed a new context abstraction based on insights from the *context-free language reachability* (CFL-reachability) formulation of pointer analysis [28, 37, 36, 48]. In the CFL-reachability formulation of pointer analysis, points-to and aliasing relationships are identified by paths in a graph representation of Java programs, where nodes represent variables and heap allocation sites, and edges represent data flow through assignments. 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: one language models data flow across assignments and heap accesses (i.e., a value stored to a field must be loaded through a load of the same field, thus forming a language of *balanced field accesses*), and the other language models data flow through method calls (i.e., a data flow entering a method from one call site must exit the method from the same call site, thus forming a language of *balanced entries/exits*). We identify these paths as *transformations* over contexts, and show that the traditional representation of context information is the explicit enumeration of input-output value pairs of 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 [32], object [23], and type sensitivity [34].

A new abstraction for context-sensitive pointer information raises theoretical questions as to its expressiveness: does performing pointer analysis using the new abstraction yield a different level of precision compared to traditional context strings? In order to make a meaningful comparison of precision, we formulate an analysis where the representation of contexts is *parameterized* such that the analysis may be instantiated with different abstractions while all other aspects of the analysis are held constant. We present a parameterized set of deduction rules for pointer analysis that can be instantiated with either the traditional representation of contexts as context strings, or with our new abstraction. Our new abstraction is strictly more precise than context strings in theory, and generally is more efficient than context strings in terms of representation size and analysis time, but obtains exactly the same precision in practice under call-site and object sensitivity.

A recent trend in the field of static analysis has been an increase in the use of Datalog to specify analyses [24, 33]. The benefits of specifying an analysis in Datalog include fast prototyping, correctness, high-analysis performance, and improved reproducibility of results due to the declarative nature of the specification. An obstacle to specifying static analyses in Datalog is that complex data-structures and operations required by an static analysis may not translate well to the tuple-based representation of data in Datalog. We have developed a new tool, a Datalog engine called DLE, designed to allow concise specification and efficient evaluation of static analyses. In particular, our new abstraction of context information takes advantage of features unique to DLE.

1.2 Contributions

This thesis 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 that can be instantiated to use either the traditional context string representation or our new representation. Similar to the Doop framework [5], the rules can be instantiated as a call-site-, object-, or type-sensitive analysis.

- We evaluate the efficiency and performance of our new abstraction on large programs in the DaCapo benchmark suite [4]. The greatest improvement obtained is nearly a 3x speedup in analysis time and a 40% reduction in the number of facts computed in one benchmark when compared to the traditional context string representation.

- We present our new Datalog engine, DLE, which supports efficient representation of complex abstractions and evaluation of operations used by static analyses. We describe our technique of implementing the new transformer string abstraction in the engine that allows for its efficient evaluation.

1.3 Organization

This dissertation is organized as follows. Chapter 2 presents background material on different types of context-sensitive analysis, the context-free-language formulation of pointer analysis, and context-string-based pointer analysis. Chapter 3 presents our new abstraction of context-sensitive pointer information. Chapter 4 contains the theorems and proofs that establish the precision difference between the two representations. Chapter 5 documents DLE: our new Datalog engine that implements extensions to Datalog designed for performing static analysis. This chapter also contains a description of how the context transformation abstractions are implemented in the engine. Chapter 6 contains the evaluation of our new abstraction and implementation. Chapter 7 surveys literature related to context-sensitive pointer analysis. Finally, Chapter 8 presents our conclusion and possible directions of future work.

1.4 Typographical Conventions

Typographical conventions are as follows: Function symbols are italicized and start with a lowercase letter. Predicate symbols are typeset in sans-serif font and start with a lowercase letter. Variables that appear as terms in literals start with an uppercase letter. Code examples are typeset in typewriter font, and references within text to symbols that appear inside examples are also typeset in typewriter font.

All free variables of a mathematical formula are universally quantified.

Sequences are formed by the concatenation operator '$\cdot$'. For clarity, sequences formed by concatenation may be delimited by square brackets when they appear inline in text (e.g., $[a \cdot b \cdot c]$), but the brackets may be omitted. The operator '$\cdot$' is used to both concatenate individual letters, and to concatenate sequences. Concatenation of single-letter symbols may omit the '$\cdot$' operator. The following functions manipulate sequences:

- Let $\mathit{prefix}_i(s)$ be the prefix of s of length $\mathit{min}(\|s\|, i)$.
- Let $\mathit{drop}_i(s)$ be the suffix of s of length $\|s\| - \mathit{min}(\|s\|, i)$.
- Let $\mathit{first}(s)$ and $\mathit{last}(s)$ be the first and last letter of a non-empty string s, respectively.

Chapter 2

Background

2.1 Introduction to Context Sensitivity

There are two primary variations on context sensitivity for an object-oriented language: A *call-site-sensitive* analysis defines contexts by the call sites of invocations, while an *object-sensitive* analysis uses the heap allocation site of a receiver object to differentiate contexts [34, 23]. *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 [34].

The example Java code in Figure 2.1 illustrates the differences between the three 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`.

Figure 2.2 illustrates the direct data-flow between allocation sites and variables that arise from direct assignment, parameter-passing, and return values. Allocation sites are represented by square nodes and variables are represented by circular nodes. Edges representing interprocedural data flows are labelled by the call sites where the data flows occur.

2.1.1 Flavours of Context Sensitivity

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 *call-site-sensitive* analysis, method `id` is analyzed in three different method contexts that correspond to the invocations labelled `c1`, `c2`, and `c3`. The analysis precisely deduces that `x1` only points to `h1` and `y1` only points to `h2`.

In an *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`.

A *type-sensitive* analysis uses the class type of the method that allocated the receiver object as the context of non-static methods. Note that the choice of context is *not* the type of objects allocated at the heap allocation site, which was empirically determined to be a poor choice of contexts in terms of analysis precision [34]. Invocations of methods `id` and `id2` are each analyzed under a single context formed by type `T`.

In both object- and type-sensitive analysis, the context of an invocation of a static method reuses the same context in which the invocation occurred.

```
class T {
  Object f;
  Object id(Object p) {
      return p;
  }
  Object id2(Object q) {
    Object u = id(q);        // c1
    return u;
  }
  Object m() {
      Object v = new T();    // o1
      return v;
  }
  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 2.1: Context sensitivity code example.

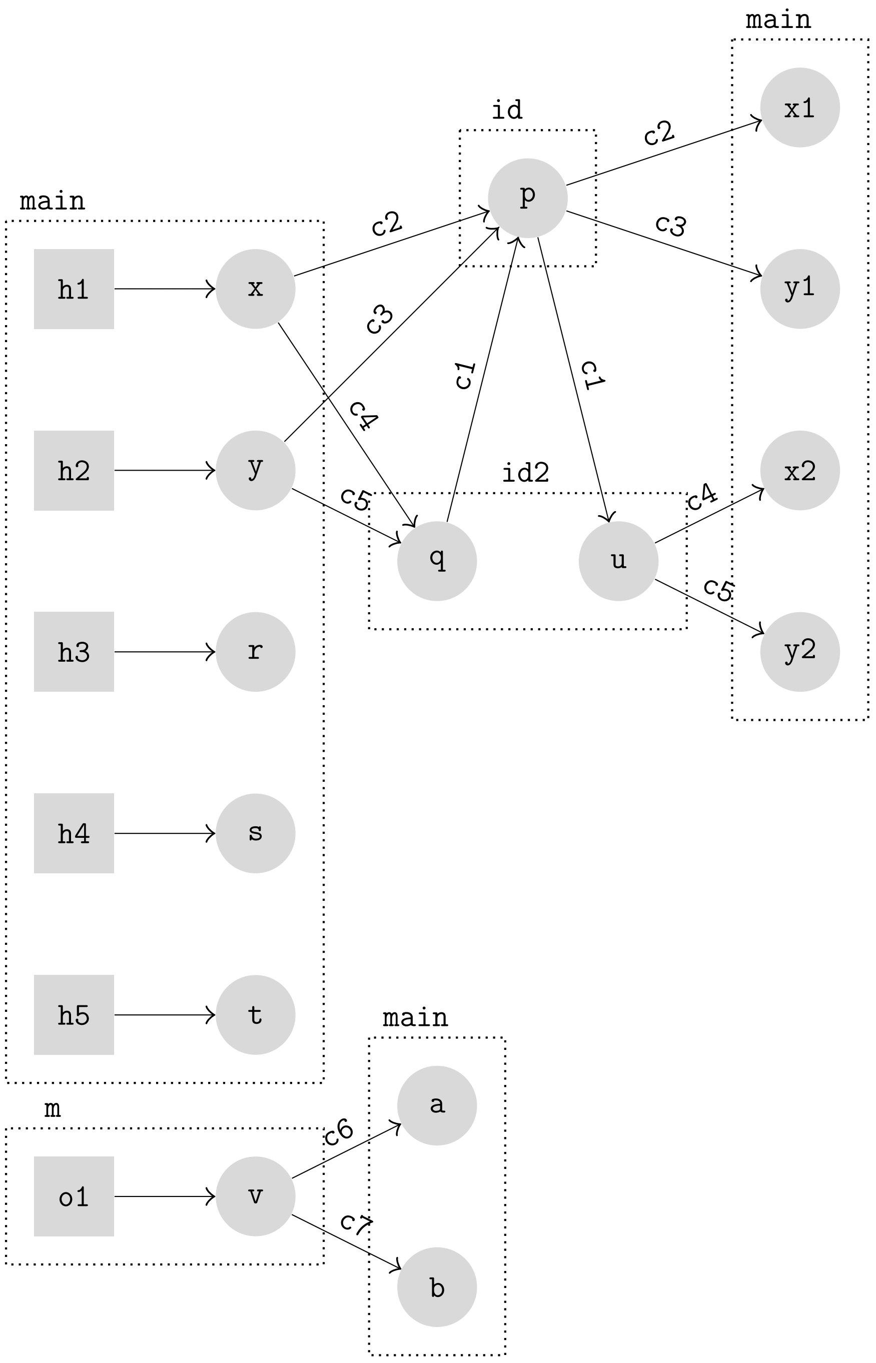


Figure 2.2: Graph showing direct data flow in the code in Figure 2.1.

2.1.2 Heap Context

Method contexts differentiate points-to sets of variables in different invocations, while *heap contexts* differentiate objects allocated in different invocations.

In Figure 2.1, without heap contexts, an analysis concludes that `a` and `b` may point to heap objects allocated at `o1` 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`. Annotating points-to relationships with heap contexts removes this imprecision: heap objects allocated at `o1` are differentiated by the method contexts of `m`, which are `c6` and `c7` under call-site sensitivity and `h4` and `h5` under object sensitivity. Either flavour concludes that `a` and `b` do not point to a common object at run-time.

2.1.3 Context-string-based Analysis

Non-demand-driven algorithms for context-sensitive pointer analysis predominantly use a *k-limited* representation of method and heap contexts, which are finite strings of *elemental contexts*. In a call-site-sensitive analysis, method contexts are *call-strings*: strings formed by return locations (call sites) of activation records of a call stack during execution. In a program with recursive calls, the call-stack has an unbounded length, and thus truncation to some k length is required to obtain a computable analysis. Context string formulations exist for a wide variety of contexts, such as call sites, heap allocation sites, class types, and combinations thereof [16]. We refer to the truncation length of context strings for method contexts as the *level of method context*, and similarly the length of context strings for heap contexts as the *level of heap context*.

We use the name **Ctxt** for the set of elemental contexts of a particular flavour of context sensitivity. For call-site-sensitive analysis, **Ctxt** is the set of call sites. For object-sensitive analysis, **Ctxt** is the set of heap allocation sites. For a type-sensitive analysis, **Ctxt** is the set of class types. Context strings are representations of contexts as strings over **Ctxt**, truncated to a finite length. We use a convention that the "top-most" elemental context appears first in a context string: for example, method `id` is invoked from call site `c1` in `id2`, and `id2` is in turn invoked from site `c4`. The method context for `id` in a call-site-sensitive analysis for this sequence of invocations is the string [`c1`·`c4`·`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 [23]. 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 [`entry`] (the special method context for entry points), and thus [`h4`·`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 [`h4`·`entry`]. This approach is a variant of object sensitivity called *full-object sensitivity*, which contrasts with *plain-object sensitivity* [34]. 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 [`h4`·`h4`·`entry`] under plain-object sensitivity. Full-object sensitivity is the variant of object sensitivity used throughout this dissertation, because full-object sensitivity has superior precision and analysis performance compared to plain-object sensitivity [5, 34]. The method context of a static invocation is the same context as the method context in which the invocation occurred.

Under full-object sensitivity, if method contexts are truncated to length m and heap contexts are truncated to length h, there are two constraints on m and h: since heap contexts are constructed from method contexts by optional truncation, we get the constraint $h \leq m$; since method contexts are constructed by prefixing the heap context of a receiver object with its allocation site, we get the constraint $m \leq h+1$. Thus, either $h = m$ or $h = m-1$. Using the latter constraint is a significantly better performance/precision trade-off than using the former, and thus we assume that the truncation lengths used under object sensitivity

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

Figure 2.3: Statements and their graph representations.

satisfy $h = m - 1$.

2.2 Context-free Language 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 2.3 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. Each statement in the program induces an edge in the PAG labelled as shown in the right-hand column in the table. Interprocedural assignments are additionally labelled below the arrow by the call sites where the assignments occur due to argument passing and return values. 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. Statements in Figure 2.3 also induce *inverse* edges: For every edge from x to y in the graph labelled l, let there be an edge from

```
class T {
  Object f;
  static Object id(Object p) {
      return p;
  }
  static Object id2(Object q) {
    Object r = id(q);        // c1
    return r;
  }
  public static void main(String[] args) {
    Object x = new T();      // g
    Object z = x;
    Object w = x;

    Object x = new Object(); // h
    z.f = x;
    Object y = w.f;

    Object u = id2(x);       // c2
    Object v = id2(x);       // c3
  }
}
```

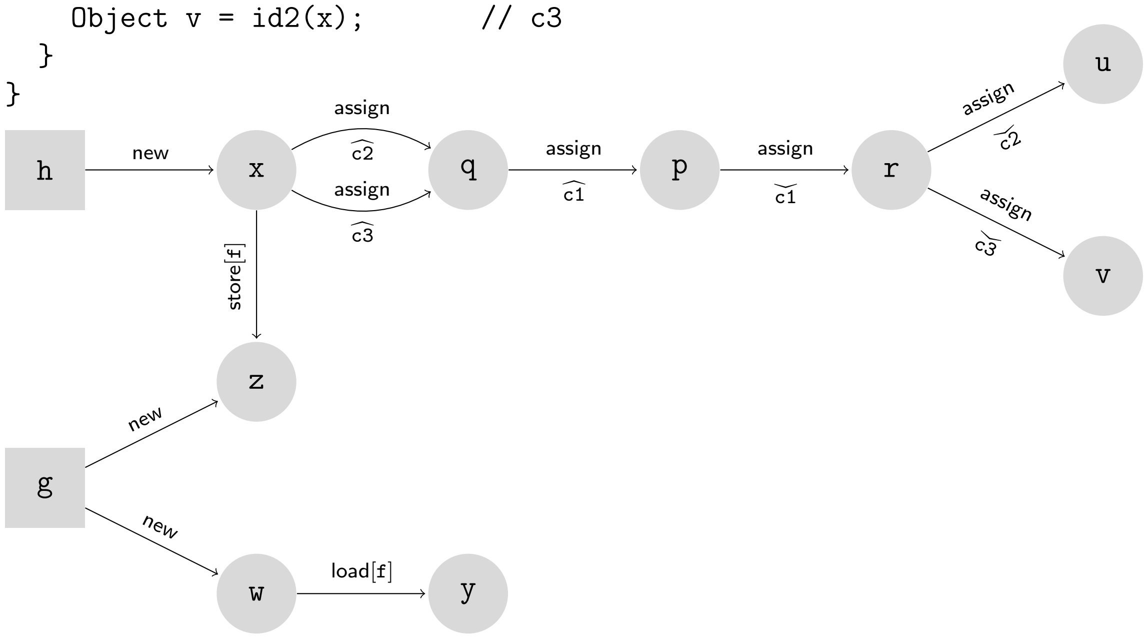


Figure 2.4: Example code and its Pointer Assignment Graph representation.

y to x in the graph labelled $\bar{l}$. Call site labels $\widehat{c}$ become $\widecheck{c}$, and vice-versa. For example, in a program with edge $\mathtt{a}_1 \xrightarrow[\widehat{c}]{\mathsf{assign}} \mathtt{f}_1$, an implicit edge $\mathtt{f}_1 \xrightarrow[\widecheck{c}]{\overline{\mathsf{assign}}} \mathtt{a}_1$ is present. Figure 2.4 is a code example with field accesses and invocations of static methods.

The *realized string* of a path is formed by concatenating the labels of traversed edges. Given a context-free language 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. For example, the two direct paths from h to u in Figure 2.4 realize the same string $[\mathsf{new}, \mathsf{assign}, \mathsf{assign}, \mathsf{assign}, \mathsf{assign}]$ over the alphabet $\{\mathsf{new}, \mathsf{assign}\}$, and realize the strings $[\widehat{\mathtt{c2}}, \widehat{\mathtt{c1}}, \widecheck{\mathtt{c1}}, \widecheck{\mathtt{c2}}]$ and $[\widehat{\mathtt{c3}}, \widehat{\mathtt{c1}}, \widecheck{\mathtt{c1}}, \widecheck{\mathtt{c2}}]$ over the alphabet $\{\widehat{\mathtt{c1}}, \widecheck{\mathtt{c1}}, \widehat{\mathtt{c2}}, \widecheck{\mathtt{c2}}, \widehat{\mathtt{c3}}, \widecheck{\mathtt{c3}}\}$. 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.

2.2.1 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 field accesses has been formulated as a CFL-reachability problem over a *balanced parentheses* language [37, 36]. 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 (nesting of balanced parentheses), 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 loads and stores:

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

FSig is a set of all field signatures. Let L_F be a language over $\mathbf{\Sigma_F}$ generated by the non-terminal $\mathsf{flowsTo}$ 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 variable f ranges over all field signatures in a program. The $\mathsf{flowsTo}$ non-terminal models the flow of values from heap allocation sites to variables.

In Figure 2.4, the direct path from h to y realizes the string $[\mathsf{new}, \mathsf{store}[\mathtt{f}], \overline{\mathsf{new}}, \mathsf{new}, \mathsf{load}[f]]$ over $\mathbf{\Sigma_F}$, which is a string generated by $\mathsf{flowsTo}$.

In a context-insensitive points-to analysis, x *points-to* h iff there exists an L_F-path from h to x [37]. 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.

2.2.2 Context-sensitive Analysis

Let $\mathbf{\Sigma_\mathcal{C}}$ be an alphabet consisting of letters $\widehat{c}$ and $\breve{c}$, where c ranges over **Inv**, the set of all call sites of a program. Let $L_\mathcal{C}$ be a language over $\mathbf{\Sigma_\mathcal{C}}$ generated by the non-terminal $\mathsf{feasible}$ defined by the following productions. The variable c ranges over all call sites in a program:

$$\begin{aligned}
\mathsf{balanced} &\rightarrow \widehat{c}\ \mathsf{balanced}\ \widecheck{c}.\\
\mathsf{balanced} &\rightarrow \mathsf{balanced}\ \mathsf{balanced} \mid \epsilon.\\
\mathsf{unbal\_exits} &\rightarrow \widecheck{c}\ \mathsf{unbal\_exits} \mid \epsilon.\\
\mathsf{unbal\_entries} &\rightarrow \widehat{c}\ \mathsf{unbal\_entries} \mid \epsilon.\\
\mathsf{feasible} &\rightarrow \mathsf{unbal\_exits}\ \mathsf{balanced}\ \mathsf{unbal\_entries}.
\end{aligned}$$

In Figure 2.4, one of the direct paths from h to u realizes the string $[\widehat{\mathsf{c2}}, \widehat{\mathsf{c1}}, \widecheck{\mathsf{c1}}, \widecheck{\mathsf{c2}}]$ over $\mathbf{\Sigma}_{\mathcal{C}}$, which is a string generated by feasible.

A path P is said to be *feasible* iff it is an L_C-path. An infeasible path characterizes data flow that 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 [28]. 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 [36]. Then L_C becomes a regular language, and thus $L_F \cap L_C$ is a context-free language, and $L_F \cap L_C$-paths can be computed.

Chapter 3

Context Transformations

The context string approach to context sensitivity partitions the unbounded number of method invocations and object allocations during an execution of a program into some finite and static partition that forms the abstraction of contexts. Points-to relationships from a variable Y to an allocation site H are tagged with pairs of strings: the method context determines the partition containing the invocation in which Y points to an object o allocated at H, and the heap context determines the partition containing the invocation that allocated the object o. Our alternative formulation instead relates the context in which an allocation of an object occurs to the method context in which a variable points to the object, by tagging pointer information with functions from contexts to contexts.

The organization of this chapter is as follows: Section 3.1 gives a high-level and informal description of our alternative formulation that uses *context transformations* to represent context information, and Section 3.2 gives a formal description of the domain of context transformations and its properties. Section 3.3 expresses pointer analysis using context transformations as a set of deduction rules. Each deduction rule models a particular language construct, such as a field load or a method invocation, and derives data-flow facts arising from the construct. Section 3.4 presents a set of *parameterized* deduction rules that can be *instantiated* to different analyses: the noncomputable analysis described in Section 3.3, and two computable analyses that uses abstractions over the domain of context transformations. One abstraction is the *explicit string* representation of context transformation, described in Subsection 3.5.1. Instantiating the rules using the explicit string representation results in a set of deduction rules identical to that of a context-string-based analysis. The other instantiation uses our new *transformer string* representation of context transformations, and is described in Subsection 3.5.2.

3.1 Intuition

An interpretation of elements of $\mathbf{\Sigma}_{\mathcal{C}}$, the alphabet of method entry/exit labels, is that they are *transformations* over call-site context strings. 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 (a feasible data-flow path) from `h` in `main` to `p` in `id` in Figure 2.4:

$$P \equiv \mathtt{h} \xrightarrow{\mathsf{new}} \mathtt{x} \xrightarrow[\widehat{\mathtt{c2}}]{\mathsf{assign}} \mathtt{q} \xrightarrow[\widehat{\mathtt{c1}}]{\mathsf{assign}} \mathtt{p}.$$

This path indicates that an object o is allocated at `h`, then method `id2` is invoked at call site `c2`, then method `id` is invoked at call site `c1`, and then the variable `p` in this invocation of `id` points to that object o. The realized string of P over $\mathbf{\Sigma}_{\mathcal{C}}$ is $\left[\widehat{\mathtt{c2}} \cdot \widehat{\mathtt{c1}}\right]$. The string can be interpreted as a function over method contexts that prefixes `c2`, then prefixes `c1` to its input. When the function is applied to context $[\mathtt{entry}]$ of `main`, we obtain a context $[\mathtt{c1} \cdot \mathtt{c2} \cdot \mathtt{entry}]$ for `id`.

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

$$P' \equiv \mathtt{p} \xrightarrow[\check{\mathtt{c1}}]{\mathsf{assign}} \mathtt{r} \xrightarrow[\check{\mathtt{c2}}]{\mathsf{assign}} \mathtt{u}.$$

This path indicates that the value of variable `p` in method `id` is returned to the invocation of `id` at `c1`, then the value is returned to the invocation of `id2` at `c2` and assigned to variable `u` in `main`. Its realized string over $\mathbf{\Sigma}_{\mathcal{C}}$ is $\left[\check{\mathtt{c1}} \cdot \check{\mathtt{c2}}\right]$. The string can be interpreted as a function over contexts that drops `c1` then drops `c2` from the front of its input. Applying it to context $[\texttt{c1}\cdot\texttt{c2}\cdot\texttt{entry}]$ of `id` yields $[\texttt{entry}]$ for `main`. We conclude that the path $P \cdot P'$ can be interpreted as an identity function: the path indicates that the variable `u` points to an object that is allocated at `h` in the same invocation of `main`.

In a traditional context-string-based analysis, the fact that `u` in context M points to `h` allocated in the same context M is redundantly enumerated for all reachable contexts M of `main`. This fact can be more compactly represented by representing context information as functions over contexts: the context in which `u` points to an object o, and the context in which o is allocated at `h`, are related by the identity function over contexts.

We interpret L_F-paths as transformations over contexts, which may include feasible paths (L_C-paths) and infeasible paths (non-L_C-paths). An infeasible path indicates a data-flow path that cannot occur during execution, and a precise context-sensitive pointer analysis must not have points-to relationships that are derived only from infeasible paths. The following is an example of an infeasible L_F-path (the entry $\widehat{\texttt{c2}}$ and exit $\widecheck{\texttt{c3}}$ are mismatched):

$$P'' \equiv \texttt{h} \xrightarrow{\textsf{new}} \texttt{x} \xrightarrow[\widehat{\textsf{c2}}]{\textsf{assign}} \texttt{q} \xrightarrow[\widehat{\textsf{c1}}]{\textsf{assign}} \texttt{p} \xrightarrow[\widecheck{\textsf{c1}}]{\textsf{assign}} \texttt{r} \xrightarrow[\widecheck{\textsf{c3}}]{\textsf{assign}} \texttt{v}.$$

To identify infeasible paths, we associate them with the *constant-error function*: a constant function whose output value is a special "error context" denoted *err*. Since a path with a sub-path that is infeasible is itself infeasible, we require all context transformations to map *err* to *err*. Then, any function composition of context transformations that includes the constant-error function is itself the constant-error function.

CFL-reachability problems correspond to *chain programs*, which are a restricted class of Datalog programs [27]. Section 3.3 encodes derivations of L_F-paths from heap allocation sites to variables as deduction rules, but before that, the domain of context transformations and its properties are formalized in the next section.

3.2 Context Transformation Domain

We define a set of transformations over contexts as an algebraic structure. Although the previous section defined context transformations as transformations over call-strings, this section generalizes to any type of context. Let $\mathbf{T} \equiv \{\widehat{a}, \breve{a} \mid a \in \mathbf{Ctxt}\}$ be the set of *primitive context transformations*, where $\widehat{a}$ is an *entry transformation* and $\breve{a}$ is an *exit transformation*. **Ctxt** is the set of elemental contexts. Let the domain of method contexts be $\mathbf{Ctxts} \equiv \mathbf{Ctxt}^* \cup \{err\}$. The primitive transformations over **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* **CtxtT** be a composition monoid formed by the closure of **T** under function composition. Let ϵ be the identity transformation. We use a *postfix notation* for function composition: $A \, ; B \equiv B \circ A$ means first apply A then B.

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

$$\widehat{M} \equiv \widehat{m_n} \, ; \ldots ; \widehat{m_1}. \qquad \breve{M} \equiv \breve{m_1} \, ; \ldots ; \breve{m_n}.$$

A source of confusion may be the reversal of composition when a string in $\mathbf{Ctxt}^*$ is converted into an entry transformation. The advantage of this notation becomes clear when we characterize the elements of **CtxtT** in the following paragraphs.

CtxtT can be shown to be an inverse semigroup: that is, for every $A \in \mathbf{CtxtT}$, there exists a unique inverse element (in the semigroup sense) $B \in \mathbf{CtxtT}$ such that $A = A ; B ; A$ and $B = B \, ; A \, ; B$. Noting that $\widehat{c}$ is an inverse of $\breve{c}$ and vice-versa, let $\widehat{c}^{-1} \equiv \breve{c}$ and

$\breve{c}^{-1} \equiv \widehat{c}$. Given $A \equiv a_1; \ldots; a_n \in \mathbf{CtxtT}$ where a_i ranges over primitive transformations, let $A^{-1} \equiv a_n^{-1}; \ldots; a_1^{-1}$. It is evident that A^{-1} is an inverse of A, and thus $\mathbf{CtxtT}$ forms a regular semigroup. Showing that the idempotents of $\mathbf{CtxtT}$ (elements x such that $x\,;x = x$) commute is sufficient to establish the uniqueness of inverses.

Noting that $(\widehat{c}\,;\breve{c})$ is the identity function, and that $(\widehat{c}\,;\breve{e})$ where $c \neq e$ is the constant-error function $\mathit{errF} \equiv (\lambda x.\ \mathit{err})$, any composition of primitive transformations containing a pair of an entry immediately followed by an exit can be simplified to some equivalent shorter composition of transformations. Thus, all elements of $\mathbf{CtxtT}$ are equivalent to either the composition of a sequence of exit transformations followed by a sequence of entry transformations, or equal to errF. Thus, non-errF context transformations have the following specification (let X and E range over $\mathbf{Ctxt}^*$):

$$\breve{X}\ ;\widehat{E} \equiv \lambda M.\ \begin{cases} E \cdot \mathit{drop}_{\|X\|}(M) & \text{if } X = \mathit{prefix}_{\|X\|}(M) \\ \mathit{err} & \text{otherwise.} \end{cases}$$

Thus, the only non-errF idempotents are of the following form:

$$\breve{X}\ ;\widehat{X} = \lambda M.\ \begin{cases} M & \text{if } X = \mathit{prefix}_{\|X\|}(M) \\ \mathit{err} & \text{otherwise.} \end{cases}$$

Then, the composition of two non-errF idempotents have the following specification:

$$\breve{X}\ ;\widehat{X}\ ;\breve{Y}\ ;\widehat{Y} \ = \lambda M.\ \begin{cases} M & \text{if } X = \mathit{prefix}_{\|X\|}(M) \\ & \quad \wedge\ Y = \mathit{prefix}_{\|Y\|}(M) \\ \mathit{err} & \text{otherwise.} \end{cases}$$

Thus, non-errF idempotents commute. Clearly, errF commutes. Thus, all idempotents of $\mathbf{CtxtT}$ commute. And thus, inverses of $\mathbf{CtxtT}$ are unique.

Using the observation that all non-errF transformations can be expressed as a composition of exit transformations followed by a composition of entry transformations, we can decompose any non-$errF$ context transformation A into two strings $X, Y \in \mathbf{Ctxt}^*$ such that $\check{X} \mathbin{;} \widehat{Y} = A$.

3.3 Pointer Analysis using Context Transformations

We express pointer analysis as a set of deduction rules, where each rule models a particular Java language construct. The premise of the rule describes the state of a program before execution of a language construct, and the conclusion of the rule must soundly describe the state of the program after execution of the construct. Rules are motivated by example dynamic executions of program constructs described using method contexts (unbounded strings of elemental contexts whose type depends on the flavour of context sensitivity). From the example dynamic executions, we infer static deduction rules that soundly models the dynamic behaviour.

The analysis described in this section is not a computable analysis. A computable analysis is given in Section 3.5. The call string variation of the analysis is as precise as a context string analysis using unbounded call strings, which is known to be non-computable [31].

The input to our analysis is a set of input relations, which describe the Java program under analysis. These relations are described in the next subsection.

3.3.1 Input Predicates

Figure 3.1 presents the input domains and predicates used by our analysis. Relations corresponding to these predicates form the input to our analysis. We use the same input schema as the Doop Framework [5]: our analysis implementation uses the same *fact generator* as that of Doop, which converts a Java class file into relations using the Soot Framework [42].

Domains:
FSig : Static and instance field signatures.
Heap : Heap allocation sites.
Inv : Invocation sites.
Method : Methods definitions.
MSig : Method signatures.
Type : Types.
Var : Variables.

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

Figure 3.1: Input domains and predicates.

Method definitions:	
// Q ≡ <class T: T_{ret} Q_{id}(T_i ...)>	
class T { T_{ret} Q_{id}(T_i f_i ...) {...} }	formal(f_i, Q, i)
	implements(Q, T, S)
	this_var(Q, this)
Statements (in a method P):	
y = z;	assign(z, y, P)
y = new T; // H	assign_new(H, y, P)
	heap_type(H, T)
// F ≡ <class T: T_f F_{id}>	
y = z.F_{id};	load(z, F, y, P)
y = z[...];	load(z, arr, y, P)
y = T.F_{id};	load_s(F, y, P)
y.F_{id} = z;	store(z, F, y, P)
y[...] = z;	store(z, arr, y, P)
T.F_{id} = z;	store_s(z, F, P)
return z;	return(z, P)
// S ≡ <class T: T_{ret} S_{id}(T_i ...)>	
y = S_{id}(a_i ...); // I	static_invoke(I, Q, P)
y = z.S_{id}(a_i ...); // I	virtual_invoke(I, z, S, P)
	actual(a_i, I, i)
	assign_return(y, I, P)

Figure 3.2: Translation of Java constructs to relations.

Figure 3.2 presents the translation of Java language constructs to relations. We differentiate between the names (identifiers) of methods and fields ($\texttt{Q}_{id}$, $\texttt{S}_{id}$, and $\texttt{F}_{id}$) and the signatures of methods and fields (`Q`, `S`, and `F`): different signatures may share the same identifier. Although most Java language constructs correspond directly to input predicates (e.g. assignment, load, and store statements), some are deconstructed into lower-level operations: for example, heap allocation statements such as "`y = new T(a);`" are converted into an heap allocation operation (identified by predicate $\mathsf{assign\_new}$), and an invocation of `T`'s constructor method. Invocations of constructor methods differ from static invocations in that they specify a receiver object and differ from virtual invocations in that they are not dynamically dispatched. Thus, the implementation uses a separate predicate to describe *special invoke* instructions. We omit this detail from this document.

Virtual invocations specify their invoked methods through method signatures. A method signature consists of a return type, a method name, and a type for each parameter. Although Java does not permit overloaded methods that differ only by their return types, the Java Virtual Machine (JVM) [22] permits it and uses it to implement overriding of methods with covariant return types. The $\mathsf{implements}$ relation includes inherited methods: that is, if P implements method signature S in type T, and T' is a direct subclass of T that does not have a method that overrides signature S, then $\mathsf{implements}(P, T', S)$ is true.

Array accesses are handled as field accesses. Our abstract analysis does not distinguish objects stored to different indices of an array. The "field" being accessed by an array access of any index is a special field signature "`arr`".

3.3.2 Deduction Rules

The result of the analysis is contained in the *derived relations* pts (points-to relation) and call (call-graph edge relation). These relations are derived by a set of deduction rules described in this subsection. The derived relations are described below:

- $\mathsf{pts}(Y, H, A)$ indicates that if an object o is allocated at H in method context M (an unbounded string of elemental contexts), then variable Y may point to o in context $A(M)$, where A is a context transformation. We say that Y points to H *under* the context transformation A.

- $\mathsf{call}(I, P, A)$ indicates that call site I in method context M may invoke method P with context $A(M)$.

Heap allocation sites and assignments.

The following rule models heap allocations:

$$\frac{\begin{array}{l}\mathsf{assign\_new}(H, Y, P) \\ \mathsf{call}(\_, P, \_)\end{array}}{\mathsf{pts}(Y, H, \epsilon)}$$

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. 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. The presence of the $\mathsf{call}(\_, P, \_)$ literal in the body ensures that points-to facts are only derived in *reachable methods* (either the entry method of a program or a method invoked by a reachable method).

Intraprocedural assignments do not alter the context under which a points-to relationship holds:

$$\frac{\begin{array}{l}\mathsf{pts}(Z, H, A) \\ \mathsf{assign}(Z, Y, \_)\end{array}}{\mathsf{pts}(Y, H, A)}$$

$\mathsf{assign}(Z, Y, P)$ indicates that variable Z is assigned to Y in method P.

Field accesses of heap objects.

Heap points-to relationships arising from stores to fields of heap objects (*instance fields*) can be described as a sequence of events during an execution of a program:

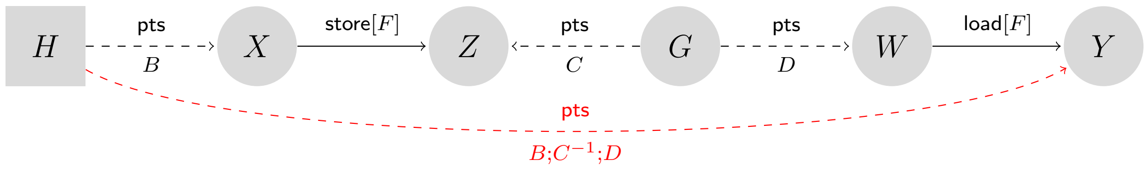
- a *pointee* object is allocated at an allocation site H in context M_H;
- a *base* object is allocated at allocation site G in context M_G;
- variable X points to the pointee object in context M_{XZ};
- variable Z, which is in the same method as X, points to the base object in the same context M_{XZ};
- and through X and Z, the pointee object is stored to field F of the base object.

Expressed in terms of pts facts, we must have $\mathsf{pts}(X, H, B)$ and $\mathsf{pts}(Z, G, C)$, such that $B(M_H) = M_{XZ}$ and $C(M_G) = M_{XZ}$. Thus, $(B\,;C^{-1})(M_H) = M_G$.

Suppose a variable W points to the base object in context M_W. Through W, the value of field F is loaded and stored into variable Y. Then, Y points to the pointee object. There must exist D such that $\mathsf{pts}(W, G, D)$ and $D(M_G) = M_W$. Then, $(B\,;C^{-1}\,;D)(M_H) = M_W$. Thus, we derive the fact $\mathsf{pts}(Y, H, B\,;C^{-1}\,;D)$. We infer the following rule for handling field accesses:

$$\frac{\begin{array}{l}\mathsf{pts}(X, H, B)\\ \mathsf{store}(X, F, Z, \_)\\ \mathsf{pts}(Z, G, C)\\ \mathsf{load}(W, F, Y, \_)\\ \mathsf{pts}(W, G, D)\\ B\,;C^{-1}\,;D \neq errF\end{array}}{\mathsf{pts}(Y, H, B\,;C^{-1}\,;D)}$$

The following graph illustrates the deduction rule. Derived predicates are depicted with dashed lines and input predicates are depicted with solid lines. Relations in the premise are coloured black and the relation in the conclusion is coloured red. The labels above the edges indicate the types of relationships that the edges convey, and the labels below are the context transformations associated with the relationships:



Static field accesses.

Accessing static fields does not require a base object, and thus static fields can be accessed from any method and from any reachable context of a method. If a variable W points to a *pointee* object allocated at allocation site H in context M_H, then there must exist a context transformation B such that $\mathsf{pts}(W, H, B)$ and $B(M_H) \neq \mathit{err}$. Literal $\mathsf{store\_s}(W, F, \_)$ indicates that the value of W is stored to a static field F, and $\mathsf{load\_s}(F, Z, P)$ indicates that static field F is loaded into variable Z in method P. Thus, variable Z may point to the pointee object in any reachable context of P.

To soundly model static field accesses, we require the derivation of a fact $\mathsf{pts}(Z, H, A)$ such that $A(M_H)$ ranges over all reachable method contexts of P. If we let C range over all context transformations, then $(B; C)(M_H)$ ranges over all method contexts. Thus, we infer the following rule for handling static field accesses:

$$\frac{\begin{array}{l}\mathsf{pts}(W, H, B)\\ \mathsf{store\_s}(W, F, \_)\\ \mathsf{load\_s}(F, Z, P)\\ \mathsf{call}(\_, P, \_)\\ C \in \mathbf{CtxtT}\\ B\,;C \neq errF\end{array}}{\mathsf{pts}(Z, H, B\,;C)}$$

The presence of the literal $\mathsf{call}(\_, P, \_)$ in the body ensures that points-to facts are not derived in unreachable methods.

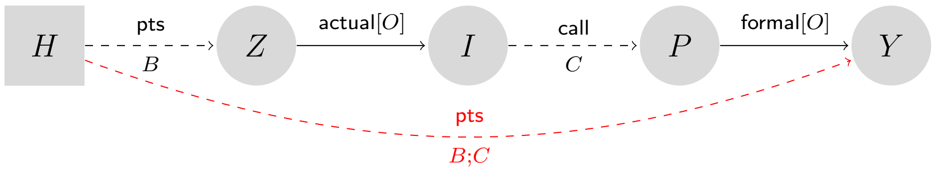
Parameter passing and return values.

Constructing a PAG representation with interprocedural *assign* edges, as in Figure 2.3, 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 [19]. Our analysis handles parameter passing using the derived predicate call, instead of using a pre-constructed call graph.

If variable Z in context M_Z points to an object o allocated at H in context M_H, there must exist B such that $\mathsf{pts}(Z, H, B)$ and $B(M_H) = M_Z$. If call site I in context M_Z invokes a method P with context M_P, then there must exist C such that $\mathsf{call}(I, P, C)$ and $C(M_Z) = M_P$. If variable Z is the O^{th} actual argument of I, then the O^{th} formal argument of P may point to o. Thus, we infer the following rule for handling parameter passing:

$$\frac{\begin{array}{l}\mathsf{pts}(Z,H,B)\\ \mathsf{actual}(Z,I,O)\\ \mathsf{call}(I,P,C)\\ \mathsf{formal}(Y,P,O)\\ B\,;C \neq errF\end{array}}{\mathsf{pts}(Y,H,B\,;C)}$$

The following graph illustrates the deduction rule. The label $\mathsf{actual}[O]$ above the edge from Z to I indicates that Z is the O^{th} actual argument of the invocation I, and the label $\mathsf{formal}[O]$ above the edge from P to Y indicates that Y is the O^{th} formal parameter of the method P:

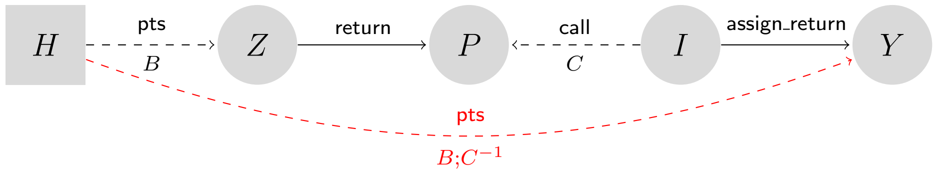


Likewise return values are handled by the following rule:

$$\frac{\begin{array}{l}\mathsf{pts}(Z,H,B)\\ \mathsf{return}(Z,P)\\ \mathsf{call}(I,P,C)\\ \mathsf{assign\_return}(I,Y,\_)\\ B\,;C^{-1} \neq errF\end{array}}{\mathsf{pts}(Y,H,B\,;C^{-1})}$$

Literal $\mathsf{return}(Z,P)$ indicates that variable Z is the return value of P, and $\mathsf{assign\_return}(I, Y,\_)$ indicates that the call site I assigns its return value to Y.

The following graph illustrates the deduction rule:



Call-graph edge derivation under call-site sensitivity.

Two types of call sites are *virtual invokes* and *static invokes*. Under call-site sensitivity, context transformations for static invocations are easily inferred: for each method context M of a method P that contains a static invoke I of a method Q, I invokes Q with context $I \cdot M$. Thus, we derive the fact $\mathsf{call}(I, Q, \widehat{I})$:

$$\frac{\begin{array}{l}\mathsf{static\_invoke}(I, Q, P) \\ \mathsf{call}(\_, P, \_)\end{array}}{\mathsf{call}(I, Q, \widehat{I})}$$

Handling virtual invocations is more difficult because the methods that they invoke depend on points-to relationships of their receiver variables, and points-to relationships are context-dependent. Literal $\mathsf{virtual\_invoke}(I, Z, S, P)$ indicates that call site I in method P invokes method signature S on the receiver object specified by variable Z. Suppose that a receiver object is allocated at site H in context M_H, and variable Z in method P points to the object in context M_Z. Then, there must exist B such that $\mathsf{pts}(Z, H, B)$ and $B(M_H) = M_Z$. Suppose that a virtual invoke I invokes method Q using Z as its receiver variable. According to the definition of call-site sensitivity, we must derive a fact $\mathsf{call}(I, Q, D)$ such that $D(M_Z) = I \cdot M_Z$. Deriving the same fact $\mathsf{call}(I, Q, \widehat{I})$ as static invocations is tempting. However, this derivation leads to imprecise results: suppose W is an actual argument of I, Y is W's corresponding formal argument in the invoked method, and we have a fact $\mathsf{pts}(W, H', B')$ such that the images of B and B' are disjoint (excluding *err*). Then, deriving $\mathsf{pts}(Y, H', B' \,;\widehat{I})$ is imprecise because the facts $\mathsf{pts}(Z, H, B)$ and $\mathsf{pts}(W, H', B')$ do not indicate that Z and W points to their respective objects *in the same method context*.

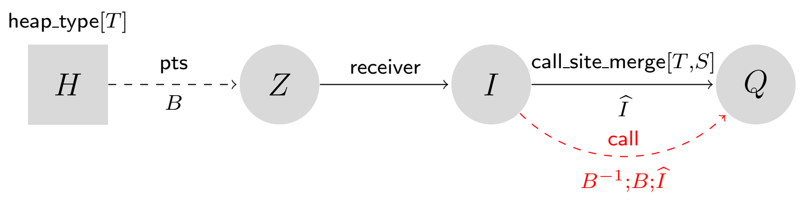
We use the following reasoning to infer the following rule:

- $(B^{-1} \,;\, B)(M_Z) = M_Z$,
- for all context transformations C such that the images of B and C are disjoint (excluding err), then $C \,;\, B^{-1} = errF$,
- and thus $B^{-1} \,;\, B \,;\, \widehat{I}$ has the desired properties of soundness and precision.

$$\frac{\begin{array}{l}\mathsf{virtual\_invoke}(I, Z, S, P)\\ \mathsf{pts}(Z, H, B)\\ \mathsf{heap\_type}(H, T)\\ \mathsf{implements}(Q, T, S)\end{array}}{\mathsf{call}(I, Q, B^{-1} \,;\, B \,;\, \widehat{I})}$$

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.

The following graph illustrates the deduction rule. The label $\mathsf{heap\_type}[T]$ above the node named H indicates that $\mathsf{heap\_type}(H, T)$ is true. The edge labelled $\mathsf{call\_site\_merge}[T, S]$ is a figurative edge that conveys how method contexts are derived for virtual invocations under call-site sensitivity. The label $\mathsf{call\_site\_merge}[T, S]$ above the edge from I to Q indicates that $\mathsf{virtual\_invoke}(I, Z, S, P) \wedge \mathsf{implements}(Q, T, S)$ is true.



Call-graph 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 [23]. 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 every 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 then derives points-to relationships through infeasible paths.

If M is a reachable context of P, then there must exist A such that $\mathsf{call}(\_, P, A)$ and $A(N) = M$ for some N. In a similar reasoning as the one used for the case of call-graph edge derivation under call-site sensitivity, using the context transformation $A^{-1}\,;A$ has the desired property of being idempotent and filtering out data-flow paths that end in a method context that is not in the image of A (i.e., not a reachable method context of P).

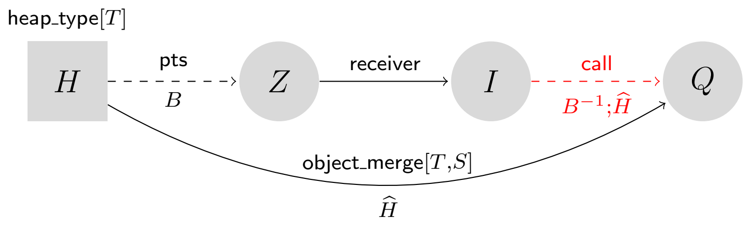
Thus, we infer the following rule for handling static invocations under object sensitivity:

$$\frac{\begin{array}{l}\mathsf{static\_invoke}(I, Q, P)\\ \mathsf{call}(\_, P, A)\end{array}}{\mathsf{call}(I, Q, A^{-1}\,; A)}$$

Suppose that a receiver object is allocated at site H in context M_H, and variable Z in method P points to the object in context M_Z. Then, there must exist a context transformation B such that $\mathsf{pts}(Z, H, B)$ and $B(M_H) = M_Z$. Suppose that a virtual invoke I invokes method Q using Z as its receiver variable. Then the context of the invoked method is $H \cdot M_H$ under object sensitivity. A context transformation A is desired such that $A(M_Z) = H \cdot M_H$ and $B'\,; A = errF$ for all B' that has a disjoint image with respect to B (excluding err). Since $B^{-1}(M_Z) = M_H$, $A \equiv B^{-1}\,;\widehat{H}$ satisfies these requirements. Thus, we infer the following rule for handling virtual invocations under object sensitivity:

$$\frac{\begin{array}{l}\mathsf{virtual\_invoke}(I, Z, S, P)\\ \mathsf{pts}(Z, H, B)\\ \mathsf{heap\_type}(H, T)\\ \mathsf{implements}(Q, T, S)\end{array}}{\mathsf{call}(I, Q, B^{-1}\,;\,\widehat{H})}$$

The following graph illustrates the deduction rule. The edge labelled $\mathsf{object\_merge}[T, S]$ is a figurative edge that conveys how method contexts are derived for virtual invocations under object sensitivity. The label $\mathsf{object\_merge}[T, S]$ above the edge from H to Q indicates that $\mathsf{virtual\_invoke}(I, Z, S, P) \wedge \mathsf{implements}(Q, T, S)$ is true.



Call-graph edge derivation under type sensitivity.

Deduction rules for type-sensitive analysis are similar to that of object sensitive analysis: The only difference is that the class type containing the method containing the allocation site of a receiver object is used as context, instead of the allocation site itself. Let $\mathit{classOf}(H)$ of a heap allocation site H be the class type of the method that contains H:

$$\frac{\begin{array}{l}\mathsf{virtual\_invoke}(I, Z, S, P)\\ \mathsf{pts}(Z, H, B)\\ \mathsf{heap\_type}(H, T)\\ \mathsf{implements}(Q, T, S)\end{array}}{\mathsf{call}(I, Q, B^{-1}\,;\,\widehat{\mathit{classOf}(H)})}$$

Static invocations are handled by the same deduction rule as in object-sensitive analysis.

The next section summarizes the deduction rules presented above, but in a modified form: operations on context transformations are refactored as parameters that can be *instantiated* with a definition depending on the choice of representation (either explicit string or transformer string). Furthermore, parts of deduction rules that differ among different context sensitivities (call-site, object, and type sensitivity) are also refactored in a design that is similar to Doop [5].

3.4 Parameterized Deduction Rules

Figure 3.3 presents the parameterized deduction rules for a context-sensitive pointer analysis. The presentation is simplified by omitting rules that handle class initialization, reflection, native code simulation, and exceptions, but they are present in the evaluated implementation. These constructs are handled in the same way as they are handled in the Doop framework [5]. Non-logical symbols that are parameters to the deduction rules have a "$\square$" superscript.

Figure 3.4 defines the instantiation of parameters in Figure 3.3 to the *context transformation instantiation*, which uses the context transformation domain **CtxtT** directly, and is semantically identical to the analysis described in Section 3.3 but differs syntactically. Rules are refactored to use the four core relations of pointer analysis that frequently appear in literature to gauge analysis precision and complexity: the points-to, heap points-to, call graph, and method reachability relations. The following are the differences:

- The rule for handling accesses to instance fields is refactored into two rules using a *heap points-to* relation $\mathsf{hpts}^\square$. A fact $\mathsf{hpts}^\square(G, F, H, A)$ indicates that if an object o is allocated at H in method context M, then field F of an object allocated at G in method context $A(M)$ may point to o.
- A new relation $\mathsf{reach}^\square$ describes the reachable method contexts of methods using

$$\frac{\begin{array}{l}\mathsf{assign\_new}(H, Y, P)\\ \mathsf{reach}^{\Box}(P, M)\\ A \equiv \mathit{record}^{\Box}(M)\end{array}}{\mathsf{pts}^{\Box}(Y, H, A)}\ [\textsc{New}]$$

$$\frac{\begin{array}{l}\mathsf{pts}^{\Box}(Z, H, A)\\ \mathsf{assign}(Z, Y, \_)\end{array}}{\mathsf{pts}^{\Box}(Y, H, A)}\ [\textsc{Assign}]$$

$$\frac{\begin{array}{l}\mathsf{pts}^{\Box}(X, H, B)\\ \mathsf{store}(X, F, Z, \_)\\ \mathsf{pts}^{\Box}(Z, G, C)\\ \mathsf{comp}^{\Box}(B, \mathit{inv}^{\Box}(C), A)\end{array}}{\mathsf{hpts}^{\Box}(G, F, H, A)}\ [\textsc{Store}]$$

$$\frac{\begin{array}{l}\mathsf{pts}^{\Box}(X, H, B)\\ \mathsf{store\_s}(X, F, \_)\\ \mathsf{load\_s}(F, Y, P)\\ \mathsf{reach}^{\Box}(P, \_)\\ \mathsf{any}^{\Box}(P, B, A)\end{array}}{\mathsf{pts}^{\Box}(Y, H, A)}\ [\textsc{FieldS}]$$

$$\frac{\begin{array}{l}\mathsf{hpts}^{\Box}(G, F, H, B)\\ \mathsf{load}(W, F, Y, \_)\\ \mathsf{pts}^{\Box}(W, G, C)\\ \mathsf{comp}^{\Box}(B, C, A)\end{array}}{\mathsf{pts}^{\Box}(Y, H, A)}\ [\textsc{Field}]$$

$$\frac{\begin{array}{l}\mathsf{pts}^{\Box}(Z, H, B)\\ \mathsf{return}(Z, P)\\ \mathsf{call}^{\Box}(I, P, C)\\ \mathsf{assign\_return}(I, Y, \_)\\ \mathsf{comp}^{\Box}(B, \mathit{inv}^{\Box}(C), A)\end{array}}{\mathsf{pts}^{\Box}(Y, H, A)}\ [\textsc{Return}]$$

$$\frac{\begin{array}{l}\mathsf{pts}^{\Box}(Z, H, B)\\ \mathsf{actual}(Z, I, O)\\ \mathsf{call}^{\Box}(I, P, C)\\ \mathsf{formal}(Y, P, O)\\ \mathsf{comp}^{\Box}(B, C, A)\end{array}}{\mathsf{pts}^{\Box}(Y, H, A)}\ [\textsc{Param}]$$

$$\frac{\begin{array}{l}\mathsf{virtual\_invoke}(I, Z, S, \_)\\ \mathsf{pts}^{\Box}(Z, H, B)\\ \mathsf{heap\_type}(H, T)\\ \mathsf{implements}(Q, T, S)\\ \mathsf{this\_var}(Y, Q)\\ C \equiv \mathit{merge}^{\Box}(H, I, B)\\ \mathsf{comp}^{\Box}(B, C, A)\end{array}}{\begin{array}{l}\mathsf{pts}^{\Box}(Y, H, A)\\ \mathsf{call}^{\Box}(I, Q, C)\end{array}}\ [\textsc{Virt}]$$

$$\frac{\begin{array}{l}\mathsf{static\_invoke}(I, Q, P)\\ \mathsf{reach}^{\Box}(P, B)\\ A \equiv \mathit{merge\_s}^{\Box}(I, B)\end{array}}{\mathsf{call}^{\Box}(I, Q, A)}\ [\textsc{Static}]$$

$$\frac{\begin{array}{l}\mathsf{call}^{\Box}(I, P, A)\\ M \equiv \mathit{target}^{\Box}(A)\end{array}}{\mathsf{reach}^{\Box}(P, M)}\ [\textsc{Reach}]$$

$$\frac{}{\mathsf{reach}^{\Box}(\mathtt{main}, [\mathtt{entry}])}\ [\textsc{Entry}]$$

Figure 3.3: Parameterized deduction rules for pointer analysis.

Concrete Context Transformation Instantiation

$\mathsf{pts}^{\mathbf{c}} \subseteq \mathbf{Var} \times \mathbf{Heap} \times \mathbf{CtxtT}.$

$\mathsf{hpts}^{\mathbf{c}} \subseteq \mathbf{Heap} \times \mathbf{FSig} \times \mathbf{Heap} \times \mathbf{CtxtT}.$

$\mathsf{call}^{\mathbf{c}} \subseteq \mathbf{Inv} \times \mathbf{Method} \times \mathbf{CtxtT}.$

$\mathsf{reach}^{\mathbf{c}} \subseteq \mathbf{Method} \times \mathbf{Ctxts}.$

$\mathsf{comp}^{\mathbf{c}} \subseteq \mathbf{CtxtT} \times \mathbf{CtxtT} \times \mathbf{CtxtT}.$

$\mathit{inv}^{\mathbf{c}} : \mathbf{CtxtT} \to \mathbf{CtxtT}.$

$\mathsf{any}^{\mathbf{c}} \subseteq \mathbf{Method} \times \mathbf{CtxtT} \times \mathbf{CtxtT}.$

$\mathit{target}^{\mathbf{c}} : \mathbf{CtxtT} \to \mathbf{Ctxts}.$

$\mathit{record}^{\mathbf{c}} : \mathbf{Ctxts} \to \mathbf{CtxtT}.$

$\mathit{merge}^{\mathbf{c}\square} : \mathbf{Heap} \times \mathbf{Inv} \times \mathbf{Ctxts} \to \mathbf{CtxtT}.$

$\mathit{merge\_s}^{\mathbf{c}\square} : \mathbf{Inv} \times \mathbf{Ctxts} \to \mathbf{CtxtT}.$

$\mathsf{comp}^{\mathbf{c}}(A, B, A\,;B) \iff (A\,;B) \neq (\lambda x.\ \mathit{err}).$

$\mathit{inv}^{\mathbf{c}}(B) \equiv B^{-1}.$

$\mathsf{any}^{\mathbf{c}}(P, A, B) \iff \exists C \in \mathbf{CtxtT}, B = A\,;C \wedge B \neq \mathit{errF}.$

$\mathit{target}^{\mathbf{c}}(\check{X}\,;\widehat{Y}) \equiv Y.$

$\mathit{record}^{\mathbf{c}}(\_) \equiv \epsilon.$

Call-site sensitivity: $\mathit{merge}^{\mathbf{cs}}(H, I, B) \equiv B^{-1}\,;B\,;\widehat{I}.$ $\mathit{merge\_s}^{\mathbf{cs}}(I, M) \equiv \widehat{I}.$

Object sensitivity: $\mathit{merge}^{\mathbf{co}}(H, I, B) \equiv B^{-1}\,;\widehat{H}.$ $\mathit{merge\_s}^{\mathbf{co}}(I, M) \equiv \widecheck{M}\,;\widehat{M}.$

Type sensitivity: $\mathit{merge}^{\mathbf{cy}}(H, I, B) \equiv B^{-1}\,;\widehat{\mathit{classOf}(H)}.$ $\mathit{merge\_s}^{\mathbf{cy}}(I, M) \equiv \widecheck{M}\,;\widehat{M}.$

Figure 3.4: Definitions of non-logical symbols in Figure 3.3 under a context transformation instantiation.

partial method contexts, which are derived by *projecting out* information that is unnecessary from context transformations of call-graph edges.

In Section 3.3, the $\mathsf{call}^{\Box}$ predicate appears in bodies of rules for three purposes:

1. interprocedural data flow (parameter passing and return values),
2. to determine if a method is reachable (for heap allocation sites and static invocations),
3. and to construct context transformations for static invocations under object and type sensitivity.

The second case checks only that a call-graph edge exists, disregarding the context transformation. In the last case, we can obtain an equivalent formulation that does not require the sequences of exits of context transformations of call-graph edges: Given a context transformation A, the rule for static invocations under object and type sensitivity computes $A^{-1}\,;A$. Let $\overset{\smile}{A_{\mathbf{x}}}\,;\overset{\frown}{A_{\mathbf{e}}} \equiv A$. Then $\overset{\smile}{A_{\mathbf{e}}}\,;\overset{\frown}{A_{\mathbf{e}}} = A^{-1}\,;A$.

Given a context transformation $\overset{\smile}{A_{\mathbf{x}}}\,;\overset{\frown}{A_{\mathbf{e}}}$ of a call-graph edge from call site I to P, we say that $A_{\mathbf{e}}$ (which is a string over $\mathbf{Ctxt}$) is the *partial method context* of P, indicated by a fact $\mathsf{reach}^{\Box}(P, A_{\mathbf{e}})$.

The parameterized symbols are described below in terms of the context transformation instantiation:

- $\mathsf{comp}^{\mathbf{c}}$ performs function composition of context transformations: $\mathsf{comp}^{\mathbf{c}}(A, B, C)$ iff $C = A; B$ and C is not the constant-error function $errF$. Composition is expressed as a predicate instead of a function to prevent the derivation of facts that contain $errF$, which signify points-to, heap points-to, and call-graph edge relationships arising from infeasible data-flow paths.
- $inv^{\mathbf{c}}$ is function inverse.

- $\mathsf{any}^{\mathbf{c}}$ is used to model loads of static fields.

- $\mathit{target}^{\mathbf{c}}$ converts call-graph edges to partial method contexts: given a context transformation A for a call-graph edge, $\mathit{target}^{\mathbf{c}}(A)$ is a partial method context N such that for all method contexts M such that $A(M) \neq \mathit{err}$, N is a prefix of $A(M)$.

- $\mathit{record}^{\mathbf{c}}$ converts partial method contexts into context transformations for points-to relationships arising at allocation sites. Since a variable that is assigned the result of a heap allocation always points to an object allocated in the same method context as the variable, the output of $\mathit{record}^{\mathbf{c}}$ is the identity function. The parameterized function $\mathit{record}^{\square}$ takes a partial method context as a parameter because the *explicit string* abstraction defined in Section 3.5.1 requires it.

- $\mathit{merge}^{\mathbf{c}\square}$ and $\mathit{merge\_s}^{\mathbf{c}\square}$ compute abstractions of context transformations for call-graph edges of virtual and static invocation sites, respectively. A second superscript differentiates the different flavours of sensitivities: **s** for call-site sensitivity, **o** for object sensitivity, and **y** for type sensitivity.

The names *record*, *merge*, and *merge_s* originate from the Doop framework [5].

A computable analysis requires an abstraction over the context transformation domain. In the next section, the non-logical symbols above are given definitions under the two abstractions of context transformations. Superscripts differentiate the two abstractions: **e** for the explicit string representation and **t** for the transformer string representation.

3.5 Abstraction

This section describes the two representations of context transformations: the traditional explicit string abstraction and our new transformer string 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. Elements of the explicit string and transformer string abstraction domains abstract *sets* of context transformations. We describe abstractions through a *concretization* function $\gamma^{\mathbf{c}}$ that maps elements X of an abstraction domain to sets of elements in the concrete domain (the context transformation domain) that are abstracted by X.

3.5.1 Explicit Strings

Pairs of context strings used in traditional points-to analysis can be interpreted as the explicit enumeration of input and output pairs of context transformations truncated to certain lengths. Different truncation lengths for the input and output strings determine the levels of context sensitivity for the method and heap contexts (defined in Section 2.1.3). The input and output strings of explicit strings form equivalence classes over untruncated method contexts: a truncated string x represents all strings with a prefix x.

Let $\mathbf{CtxtT}^{\mathbf{e}}_{i,j} \equiv \{(A, B) \mid A \in \mathbf{Ctxt}^*, B \in \mathbf{Ctxt}^*, \|A\| \leq i \wedge \|B\| \leq j\}$ be the domain of *explicit strings*, given integers i and j. Given a pair (A, B) in $\mathbf{CtxtT}^{\mathbf{e}}_{i,j}$, let its concretization into a set of context transformations be defined in the following way:

$$\begin{aligned}
\gamma^{\mathbf{c}}((A, B)) = \{&\check{A} \,;\, C \,;\, \widehat{B} \mid C \in \mathbf{CtxtT} \setminus \{errF\}, \|A\| = i \wedge \|B\| = j\} \\
\cup\ \{&\check{A} \,;\, \widehat{E} \,;\, \widehat{B} \mid E \in \mathbf{Ctxt}^*, \|A\| < i \wedge \|B\| = j\} \\
\cup\ \{&\check{A} \,;\, \check{X} \,;\, \widehat{B} \mid X \in \mathbf{Ctxt}^*, \|A\| = i \wedge \|B\| < j\} \\
\cup\ \{&\check{A} \,;\, \widehat{B} \mid \|A\| < i \wedge \|B\| < j\}.
\end{aligned}$$

Thus, for all $T \in \gamma^{\mathbf{c}}((A, B))$, if a method context M has a prefix A, then $T(M)$ is a method context with prefix B.

Relations used in pointer analysis use different truncation lengths for explicit strings. Parameters i and j of a domain $\mathbf{CtxtT}^{\mathbf{e}}_{i,j}$ 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 explicit string abstraction domain for pts is $\mathbf{CtxtT}^{\mathsf{e}}_{h,m}$, 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 domain $\mathbf{CtxtT}^{\mathsf{e}}_{i,j}$ has strings that are shorter than the truncation lengths, but these strings are only used to represent untruncated method contexts that are shorter than the truncation lengths. Short untruncated method contexts appear in *shallow* method invocations close to the entry point of a program.

Figure 3.5 presents the definitions for the parameterized non-logical symbols in Figure 3.3 using the explicit string abstraction of context transformations. Integers m and h define the levels of method and heap contexts, respectively, and are a part of the parameters of an instantiation.

Predicate $\mathsf{comp}^{\mathsf{e}}$ and function inv^{e} are polymorphic with respect to their arguments: for example, the relation $\mathsf{comp}^{\mathsf{e}}$ in the instantiated STORE rule in Figure 3.3 is a subset of $\mathbf{CtxtT}^{\mathsf{e}}_{h,m} \times \mathbf{CtxtT}^{\mathsf{e}}_{m,h} \times \mathbf{CtxtT}^{\mathsf{e}}_{h,h}$, while in the instantiated PARAM rule, the relation is a subset of $\mathbf{CtxtT}^{\mathsf{e}}_{h,m} \times \mathbf{CtxtT}^{\mathsf{e}}_{m,m} \times \mathbf{CtxtT}^{\mathsf{e}}_{h,m}$.

The interpretations of the $\mathsf{pts}^{\mathsf{e}}$, $\mathsf{hpts}^{\mathsf{e}}$, $\mathsf{call}^{\mathsf{e}}$, and $\mathsf{reach}^{\mathsf{e}}$ predicates are as follows:

- A fact $\mathsf{pts}^{\mathsf{e}}(Y, H, (U, V))$ indicates that variable Y in a method context with prefix U points to a heap object allocated at H in a method context with prefix V.
- A fact $\mathsf{hpts}^{\mathsf{e}}(G, F, H, (U, V))$ indicates that a heap object allocated at G in a method context with prefix V has a field F that points to a heap object allocated at H in a method context with prefix U.
- A fact $\mathsf{call}^{\mathsf{e}}(I, P, (U, V))$ indicates that invoke instruction I in a method context with prefix U invokes procedure P with a method context with prefix V.

Explicit String Instantiation

$\mathbf{CtxtT}^{\mathbf{e}}_{i,j} \equiv \{(A, B) \mid A \in \mathbf{Ctxt}^*, B \in \mathbf{Ctxt}^*, \|A\| \le i \wedge \|B\| \le j\}.$

$\mathsf{pts}^{\mathbf{e}} \subseteq \mathbf{Var} \times \mathbf{Heap} \times \mathbf{CtxtT}^{\mathbf{e}}_{h,m}.$
$\mathsf{hpts}^{\mathbf{e}} \subseteq \mathbf{Heap} \times \mathbf{FSig} \times \mathbf{Heap} \times \mathbf{CtxtT}^{\mathbf{e}}_{h,h}.$
$\mathsf{call}^{\mathbf{e}} \subseteq \mathbf{Inv} \times \mathbf{Method} \times \mathbf{CtxtT}^{\mathbf{e}}_{m,m}.$
$\mathsf{reach}^{\mathbf{e}} \subseteq \mathbf{Method} \times \mathbf{Ctxts}.$

$\mathsf{comp}^{\mathbf{e}} \subseteq \mathbf{CtxtT}^{\mathbf{e}}_{i,j} \times \mathbf{CtxtT}^{\mathbf{e}}_{j,k} \times \mathbf{CtxtT}^{\mathbf{e}}_{i,k}.$
$\mathit{inv}^{\mathbf{e}} : \mathbf{CtxtT}^{\mathbf{e}}_{i,j} \to \mathbf{CtxtT}^{\mathbf{e}}_{j,i}.$
$\mathsf{any}^{\mathbf{e}} \subseteq \mathbf{Method} \times \mathbf{CtxtT}^{\mathbf{e}}_{h,m} \times \mathbf{CtxtT}^{\mathbf{e}}_{h,m}.$
$\mathit{target}^{\mathbf{e}} : \mathbf{CtxtT}^{\mathbf{e}}_{m,m} \to \mathbf{Ctxts}.$

$\mathit{record}^{\mathbf{e}} : \mathbf{Ctxts} \to \mathbf{CtxtT}^{\mathbf{e}}_{h,m}.$
$\mathit{merge}^{\mathbf{e}\square} : \mathbf{Heap} \times \mathbf{Inv} \times \mathbf{Ctxts} \to \mathbf{CtxtT}^{\mathbf{e}}_{m,m}.$
$\mathit{merge\_s}^{\mathbf{e}\square} : \mathbf{Inv} \times \mathbf{Ctxts} \to \mathbf{CtxtT}^{\mathbf{e}}_{m,m}.$

$\mathsf{comp}^{\mathbf{e}}((U, V), (V, W), (U, W)).$
$\mathit{inv}^{\mathbf{e}}((U, V)) \equiv (V, U).$
$\mathsf{any}^{\mathbf{e}}(P, (U, V), (U, M)) \iff \mathsf{reach}^{\mathbf{e}}(P, M).$
$\mathit{target}^{\mathbf{e}}((U, V)) \equiv V.$
$\mathit{record}^{\mathbf{e}}(M) \equiv (\mathit{prefix}_h(M), M).$

Call-site sensitivity:	$\mathit{merge}^{\mathbf{es}}(H, I, (\_, M)) \equiv (M, I \cdot \mathit{prefix}_{m-1}(M)).$ $\mathit{merge\_s}^{\mathbf{es}}(I, M) \equiv (M, I \cdot \mathit{prefix}_{m-1}(M)).$
Object sensitivity:	$\mathit{merge}^{\mathbf{eo}}(H, I, (H', M)) \equiv (M, H \cdot H').$ $\mathit{merge\_s}^{\mathbf{eo}}(I, M) \equiv (M, M).$
Type sensitivity:	$\mathit{merge}^{\mathbf{ey}}(H, I, (H', M)) \equiv (M, \mathit{classOf}(H) \cdot H').$ $\mathit{merge\_s}^{\mathbf{ey}}(I, M) \equiv (M, M).$

Figure 3.5: Definitions of non-logical symbols in Figure 3.3 under an explicit string instantiation.

- A fact $\mathsf{reach}^{\mathsf{e}}(P, M)$ indicates that procedure P is invoked with a method context with prefix M.

The rules of an explicit string instantiation of Figure 3.3 are identical to the rules of the traditional context-string-based analysis described in Section 2.1.3.

A redundancy in information representation can be observed in the instantiated rules. Consider, for example, the New rule where the definition of *record*$^{\mathsf{e}}$ is inlined into the rule and terms are unified:

$$\frac{\begin{array}{l}\mathsf{assign\_new}(H, Y, P)\\ \mathsf{reach}^{\mathsf{e}}(P, M)\end{array}}{\mathsf{pts}^{\mathsf{e}}(Y, H, (\mathit{prefix}_h(M), M))}$$

The rule enumerates all reachable method contexts of a method, when the context transformation being expressed is simply the identity function.

Another rule that performs redundant enumeration is the FieldS rule, which models program executions where a heap object is loaded from a static field:

$$\frac{\begin{array}{l}\mathsf{pts}^{\mathsf{e}}(X, H, (U, V))\\ \mathsf{store\_s}(X, F, \_)\\ \mathsf{load\_s}(F, Y, P)\\ \mathsf{reach}^{\mathsf{e}}(P, M)\end{array}}{\mathsf{pts}^{\mathsf{e}}(Y, H, (U, M))}$$

The variable Y in method P points-to an object allocated at H in context U, in every method context M of P.

The next subsection introduces our new abstraction that is able to represent context information with less redundancy.

3.5.2 Transformer Strings

This section introduces our abstraction of context transformations as *transformer strings*. Proofs of lemmas are at the end of this subsection.

Let $\mathbf{T}_{\mathcal{W}} \equiv \{\widehat{a}, \breve{a}, * \mid a \in \mathbf{Ctxt}\}$ be an alphabet that consists of letters representing entry and exit transformations and a "wildcard" letter "$*$" that represents all non-$errF$ context transformations. The concretization of elements of $\mathbf{T}_{\mathcal{W}}$ is defined in the following way:

$$\gamma^{\mathbf{c}}(\widehat{a}) \equiv \{\widehat{a}\}.$$
$$\gamma^{\mathbf{c}}(\breve{a}) \equiv \{\breve{a}\}.$$
$$\gamma^{\mathbf{c}}(*) \equiv \mathbf{CtxtT} \setminus \{errF\}.$$

Transformer strings are strings in $\mathbf{Ts} \equiv \mathbf{T}_{\mathcal{W}}^{*} \cup \{\bot\}$. The special element $\bot$ represents $errF$. The concretization of transformer strings is defined as

$$\gamma^{\mathbf{c}}(a_1 \cdot \ldots \cdot a_n) \equiv \{a'_1 \,;\ldots;\, a'_n \mid a'_1 \in \gamma^{\mathbf{c}}(a_1), \ldots, a'_n \in \gamma^{\mathbf{c}}(a_n)\}.$$

Naturally, the concretization of an empty string ϵ is the set containing only the identity function, and the concretization of $\bot$ is the set containing only $errF$.

We use the following notation (similar to that used for context transformations) to convert a string $M \equiv m_1 \cdot \ldots \cdot m_n \in \mathbf{Ctxt}^{*}$ into transformer strings $\widehat{M}$ and $\breve{M}$:

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

The *match* : $\mathbf{Ts} \to \mathbf{Ts}$ function defined below reduces the length of a transformer string without modifying its interpretation as a transformation:

$$\begin{aligned}
&match(A\cdot\widehat{a}\cdot\breve{a}\cdot B) = match(A\cdot B).\\
&match(A\cdot\widehat{a}\cdot\breve{b}\cdot B) \equiv \bot. \qquad (a \neq b)\\
&match(A\cdot\widehat{a}\cdot *\cdot B) = match(A\cdot *\cdot B).\\
&match(A\cdot *\cdot\breve{a}\cdot B) = match(A\cdot *\cdot B).\\
&match(A\cdot *\cdot *\cdot B) = match(A\cdot *\cdot B).\\
&match(\breve{A}\cdot *\cdot\widehat{B}) = \breve{A}\cdot *\cdot\widehat{B}.\\
&match(\breve{A}\cdot\widehat{B}) = \breve{A}\cdot\widehat{B}.\\
&match(\bot) = \bot.
\end{aligned}$$

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 final string. The following two lemmas establish 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 3.5.1. *For all $A \in \mathbf{Ts}$, $\gamma^{\mathbf{c}}(A) = \gamma^{\mathbf{c}}(match(A))$.*

Lemma 3.5.2. *For all $A, B \in \mathbf{Ts}$, $\gamma^{\mathbf{c}}(A) = \gamma^{\mathbf{c}}(B) \implies match(A) = match(B)$.*

Let $\mathbf{CtxtT^t} \equiv \{\breve{A}\cdot w\cdot\widehat{B} \mid A, B \in \mathbf{Ctxt}^*, w \in \{*, \epsilon\}\}$ be the domain of *untruncated canonical transformer strings*. The deduction rules of Figure 3.4 filter out the *errF* transformation, and thus the $\bot$ element from $\mathbf{Ts}$ is not present in the $\mathbf{CtxtT^t}$ domain. Let $\mathbf{CtxtT}^{\mathbf{t}}_{i,j}$ be a subset of $\mathbf{Ctxt^t}$ that consists of strings with at most i exits and at most j entries (henceforth, the domain of *transformer strings*):

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

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

$$trunc_{i,j}(\breve{A}\cdot w\cdot\widehat{B}) \equiv \begin{cases} \breve{A}\cdot w\cdot\widehat{B} & \text{if } \|A\| \leq i \wedge \|B\| \leq j \\ \breve{A_i}\cdot *\cdot\widehat{B_j} & \text{otherwise, where} \\ & \quad A_i \equiv prefix_i(A) \text{ and} \\ & \quad B_j \equiv prefix_j(B) \end{cases}$$

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, meaning that context transformations representing feasible paths are not discarded by truncation:

Lemma 3.5.3. *For all A in* $\mathbf{CtxtT^t}$,

$$\gamma^{\mathbf{c}}(A) \subseteq \gamma^{\mathbf{c}}(\mathit{trunc}_{i,j}(A)).$$

A common notation used in this dissertation is the decomposition of a transformer string into its *exit part*, *wildcard part*, and *entry part*. For example, given a transformer string $A \equiv \breve{x}_1 \cdot \breve{x}_2 \cdot \widehat{e_1} \cdot \widehat{e_2}$ and letting $\breve{A_{\mathbf{x}}} \cdot A_{\mathbf{w}} \cdot \widehat{A_{\mathbf{e}}} \equiv A$, then $A_{\mathbf{x}} = x_1 \cdot x_2$, $A_{\mathbf{w}} = \epsilon$, and $A_{\mathbf{e}} = e_2 \cdot e_1$. If A has a wildcard letter, then $A_{\mathbf{w}} = *$.

Figure 3.6 contains the definitions of the parameterized non-logical symbols in Figure 3.3 for an instantiation of the analysis using the transformer string abstraction of context transformations. For example, the New rule instantiates into the following rule:

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

The variable that ranges over method contexts of P is no longer used in the conclusion of the rule, and enumeration of method contexts is unnecessary. The instantiation of the FieldS rule benefits from the wildcard letter abstraction:

$$\frac{\mathsf{pts^t}(X, H, \breve{B_{\mathbf{x}}} \cdot B_{\mathbf{w}} \cdot \widehat{B_{\mathbf{e}}}) \quad \mathsf{store\_s}(X, F, \_) \quad \mathsf{load\_s}(F, Y, P) \quad \mathsf{reach^t}(P, \_)}{\mathsf{pts^t}(Y, H, \mathit{match}(\breve{B_{\mathbf{x}}} \cdot *))}$$

Since the wildcard letter represents an arbitrary sequence of entries, a single fact is sufficient to express that an object allocated at H in some method context M such that $\breve{B_{\mathbf{x}}}(M) \neq \mathit{err}$ can be loaded in any method context of method P.

Transformer String Instantiation

$$\mathbf{CtxtT}^{\mathbf{t}}_{i,j} \equiv \{\widecheck{A_{\mathbf{x}}} \cdot A_{\mathbf{w}} \cdot \widehat{A_{\mathbf{e}}} \mid A_{\mathbf{x}}, A_{\mathbf{e}} \in \mathbf{Ctxt}^{*}, A_{\mathbf{w}} \in \{*, \epsilon\}, \|A_{\mathbf{x}}\| \leq i \wedge \|A_{\mathbf{e}}\| \leq j\}.$$

$\mathsf{pts}^{\mathbf{t}} \subseteq \mathbf{Var} \times \mathbf{Heap} \times \mathbf{CtxtT}^{\mathbf{t}}_{h,m}.$

$\mathsf{hpts}^{\mathbf{t}} \subseteq \mathbf{Heap} \times \mathbf{FSig} \times \mathbf{Heap} \times \mathbf{CtxtT}^{\mathbf{t}}_{h,h}.$

$\mathsf{call}^{\mathbf{t}} \subseteq \mathbf{Inv} \times \mathbf{Method} \times \mathbf{CtxtT}^{\mathbf{t}}_{m,m}.$

$\mathsf{reach}^{\mathbf{t}} \subseteq \mathbf{Method} \times \mathbf{Ctxts}.$

$\mathsf{comp}^{\mathbf{t}} \subseteq \mathbf{CtxtT}^{\mathbf{t}}_{i,j} \times \mathbf{CtxtT}^{\mathbf{t}}_{j,k} \times \mathbf{CtxtT}^{\mathbf{t}}_{i,k}.$

$\mathit{inv}^{\mathbf{t}} : \mathbf{CtxtT}^{\mathbf{t}}_{i,j} \rightarrow \mathbf{CtxtT}^{\mathbf{t}}_{j,i}.$

$\mathsf{any}^{\mathbf{t}} \subseteq \mathbf{Method} \times \mathbf{CtxtT}^{\mathbf{t}}_{h,m} \times \mathbf{CtxtT}^{\mathbf{t}}_{h,m}.$

$\mathit{target}^{\mathbf{t}} : \mathbf{CtxtT}^{\mathbf{t}}_{m,m} \rightarrow \mathbf{Ctxts}.$

$\mathit{record}^{\mathbf{t}} : \mathbf{Ctxts} \rightarrow \mathbf{Ctxt}_{h,m}.$

$\mathit{merge}^{\mathbf{t}\Box} : \mathbf{Heap} \times \mathbf{Inv} \times \mathbf{Ctxts} \rightarrow \mathbf{CtxtT}^{\mathbf{t}}_{m,m}.$

$\mathit{merge\_s}^{\mathbf{t}\Box} : \mathbf{Inv} \times \mathbf{Ctxts} \rightarrow \mathbf{CtxtT}^{\mathbf{t}}_{m,m}.$

$\mathsf{comp}^{\mathbf{t}}(A, B, \mathit{trunc}_{i,k}(\mathit{match}(A \cdot B))) \iff \mathit{match}(A \cdot B) \neq \bot.$

$\mathit{inv}^{\mathbf{t}}(\widecheck{B_{\mathbf{x}}} \cdot B_{\mathbf{w}} \cdot \widehat{B_{\mathbf{e}}}) \equiv \widecheck{B_{\mathbf{e}}} \cdot B_{\mathbf{w}} \cdot \widehat{B_{\mathbf{x}}}.$

$\mathsf{any}^{\mathbf{t}}(\_, \widecheck{B_{\mathbf{x}}} \cdot B_{\mathbf{w}} \cdot \widehat{B_{\mathbf{e}}}, \mathit{match}(\widecheck{B_{\mathbf{x}}} \cdot *)).$

$\mathit{target}^{\mathbf{t}}(\widecheck{B_{\mathbf{x}}} \cdot B_{\mathbf{w}} \cdot \widehat{B_{\mathbf{e}}}) \equiv B_{\mathbf{e}}.$

$\mathit{record}^{\mathbf{t}}(\_) \equiv \epsilon.$

Call-site sensitivity:
$\mathit{merge}^{\mathbf{tc}}(H, I, \widecheck{B_{\mathbf{x}}} \cdot B_{\mathbf{w}} \cdot \widehat{B_{\mathbf{e}}}) \equiv \mathit{trunc}_{m,m}(\widecheck{B_{\mathbf{e}}} \cdot \widehat{B_{\mathbf{e}}} \cdot \widehat{I}).$
$\mathit{merge\_s}^{\mathbf{tc}}(I, M) \equiv \widehat{I}.$

Object sensitivity:
$\mathit{merge}^{\mathbf{to}}(H, I, \widecheck{B_{\mathbf{x}}} \cdot B_{\mathbf{w}} \cdot \widehat{B_{\mathbf{e}}}) \equiv \widecheck{B_{\mathbf{e}}} \cdot B_{\mathbf{w}} \cdot \widehat{B_{\mathbf{x}}} \cdot \widehat{H}.$
$\mathit{merge\_s}^{\mathbf{to}}(I, M) \equiv \widecheck{M} \cdot \widehat{M}.$

Type sensitivity:
$\mathit{merge}^{\mathbf{ty}}(H, I, \widecheck{B_{\mathbf{x}}} \cdot B_{\mathbf{w}} \cdot \widehat{B_{\mathbf{e}}}) \equiv \widecheck{B_{\mathbf{e}}} \cdot B_{\mathbf{w}} \cdot \widehat{B_{\mathbf{x}}} \cdot \widehat{\mathit{classOf}}(H).$
$\mathit{merge\_s}^{\mathbf{ty}}(I, M) \equiv \widecheck{M} \cdot \widehat{M}.$

Figure 3.6: Definitions of non-logical symbols in Figure 3.3 under a transformer string instantiation.

Proofs

Proof of Lemma 3.5.1

Proof. $\gamma^{\mathbf{c}}(\bot) = \gamma^{\mathbf{c}}(match(\bot))$ is a trivial case. For strings in $\mathbf{T}^*_{\mathcal{W}}$, use strong induction on the lengths of strings:

Base case: $\gamma^{\mathbf{c}}(match(\epsilon)) = \gamma^{\mathbf{c}}(\epsilon)$.

Induction case: Suppose that for all X such that $\|X\| < n$, $\gamma^{\mathbf{c}}(X) = \gamma^{\mathbf{c}}(match(X))$. We must show that for all Y such that $\|Y\| = n$, $\gamma^{\mathbf{c}}(Y) = \gamma^{\mathbf{c}}(match(Y))$.

It is evident that, in Y, there exists either

1. an entry followed by a matching exit,
2. an entry followed by a non-matching exit,
3. an entry followed by a wildcard,
4. a wildcard followed by an exit,
5. a wildcard followed by a wildcard,

or Y can be written as $\breve{A}\cdot * \cdot \widehat{B}$ or $\breve{A}\cdot \widehat{B}$. The five recursive cases are handled below:

1. *Case:* $Y = A\cdot\widehat{a}\cdot\breve{a}\cdot B$.
   $\gamma^{\mathbf{c}}(\widehat{a}\cdot\breve{a}) = \{\widehat{a}\,;\breve{a}\} = \{\epsilon\}$. Thus, $\gamma^{\mathbf{c}}(A\cdot\widehat{a}\cdot\breve{a}\cdot B) = \gamma^{\mathbf{c}}(A\cdot B)$. We have $\gamma^{\mathbf{c}}(A\cdot B) = \gamma^{\mathbf{c}}(match(A\cdot B))$ from the induction hypothesis. Thus the lemma follows.
2. *Case:* $Y = A\cdot\widehat{a}\cdot\breve{b}\cdot B$ where $a \neq b$.
   We have $\gamma^{\mathbf{c}}(\widehat{a}\cdot\breve{b}) = \{\widehat{a}\,;\breve{b}\} = \{errF\} = \gamma^{\mathbf{c}}(\bot)$. Thus, $\gamma^{\mathbf{c}}(A\cdot\widehat{a}\cdot\breve{b}\cdot B) = \gamma^{\mathbf{c}}(\bot)$.
3. *Case:* $Y = A\cdot\widehat{a}\cdot *\cdot B$.
   We have $\gamma^{\mathbf{c}}(\widehat{a}\cdot *) = \{\widehat{a}\,; C \mid C \in \mathbf{CtxtT} \setminus \{errF\}\}$. Let $D \in \mathbf{CtxtT} \setminus \{errF\}$.

There exists M such that $D(M) \neq \mathit{err}$. Then $(\breve{a}\,;D)(a \cdot M) \neq \mathit{err}$ and thus $\breve{a}\,;D \in \mathbf{CtxtT} \setminus \{\mathit{errF}\}$. Thus $D \in \{\widehat{a}\,;C \mid C \in \mathbf{CtxtT} \setminus \{\mathit{errF}\}\}$ because $\widehat{a}\,;\breve{a}\,;D = D$. Thus, $\gamma^{\mathbf{c}}(\widehat{a} \cdot *) = \gamma^{\mathbf{c}}(*)$, and using the induction hypothesis, the lemma follows.

4. *Case:* $Y = A \cdot * \cdot \breve{a} \cdot B$.
   We have $\gamma^{\mathbf{c}}(* \cdot \breve{a}) = \{C\,;\breve{a} \mid C \in \mathbf{CtxtT} \setminus \{\mathit{errF}\}\}$. Let $D \in \mathbf{CtxtT} \setminus \{\mathit{errF}\}$. There exists M such that $D(M) \neq \mathit{err}$. Then $(D\,;\widehat{a})(M) \neq \mathit{err}$ and thus $D\,;\widehat{a} \in \mathbf{CtxtT} \setminus \{\mathit{errF}\}$. Thus $D \in \{C\,;\breve{a} \mid C \in \mathbf{CtxtT} \setminus \{\mathit{errF}\}\}$ because $D\,;\widehat{a}\,;\breve{a} = D$. Thus, $\gamma^{\mathbf{c}}(* \cdot \breve{a}) = \gamma^{\mathbf{c}}(*)$, and using the induction hypothesis, the lemma follows.

5. *Case:* $Y = A \cdot * \cdot * \cdot B$.
   Trivially, $\gamma^{\mathbf{c}}(* \cdot *) = \gamma^{\mathbf{c}}(*)$. Using the induction hypothesis, the lemma follows.

$\square$

Proof of Lemma 3.5.2

Proof. We will use the following notation to apply a method context to a set of context transformations to obtain a set of method contexts: Given a set of context transformations X and $M \in \mathbf{Ctxt}^*$, let $\underline{X}(M) \equiv \{x(M) \mid x \in X, x(M) \neq \mathit{err}\}$.

The contrapositive is shown: $\mathit{match}(A) \neq \mathit{match}(B) \implies \gamma^{\mathbf{c}}(A) \neq \gamma^{\mathbf{c}}(B)$.

Let $\breve{A_{\mathbf{x}}} \cdot A_{\mathbf{w}} \cdot \widehat{A_{\mathbf{e}}} \equiv \mathit{match}(A)$, $\breve{B_{\mathbf{x}}} \cdot B_{\mathbf{w}} \cdot \widehat{B_{\mathbf{e}}} \equiv \mathit{match}(B)$, and let $\mathit{match}(A) \neq \mathit{match}(B)$. Clearly, $A_{\mathbf{e}} \in \underline{\gamma^{\mathbf{c}}(A)}(A_{\mathbf{x}})$ and $B_{\mathbf{e}} \in \underline{\gamma^{\mathbf{c}}(B)}(B_{\mathbf{x}})$.

If $A_{\mathbf{w}} = \epsilon \wedge B_{\mathbf{w}} = *$, then $\|\gamma^{\mathbf{c}}(A)\| = 1$ and $\|\gamma^{\mathbf{c}}(B)\| = \infty$ and thus, $\gamma^{\mathbf{c}}(A) \neq \gamma^{\mathbf{c}}(B)$. Similarly if $A_{\mathbf{w}} = * \wedge B_{\mathbf{w}} = \epsilon$ then $\gamma^{\mathbf{c}}(A) \neq \gamma^{\mathbf{c}}(B)$.

Suppose $A_{\mathbf{w}} = B_{\mathbf{w}}$. If $A_{\mathbf{e}} \neq B_{\mathbf{e}} \wedge \|A_{\mathbf{e}}\| \leq \|B_{\mathbf{e}}\|$, then $A_{\mathbf{e}} \notin \underline{\gamma^{\mathbf{c}}(B)}(A_{\mathbf{x}})$, because all strings in $\underline{\gamma^{\mathbf{c}}(B)}(A_{\mathbf{x}})$ have a prefix that is as long as but not equal to $A_{\mathbf{e}}$. Similarly, if $A_{\mathbf{e}} \neq B_{\mathbf{e}} \wedge \|A_{\mathbf{e}}\| \geq \|B_{\mathbf{e}}\|$, then $B_{\mathbf{e}} \notin \underline{\gamma^{\mathbf{c}}(A)}(B_{\mathbf{x}})$.

Suppose $A_\mathbf{e} = B_\mathbf{e}$. $A_\mathbf{e} \notin \underline{\gamma^\mathbf{c}(B)}(A_\mathbf{x})$ implies $\gamma^\mathbf{c}(A) \neq \gamma^\mathbf{c}(B)$ because $A_\mathbf{e} \in \underline{\gamma^\mathbf{c}(A)}(A_\mathbf{x})$. $A_\mathbf{e} \in \underline{\gamma^\mathbf{c}(B)}(A_\mathbf{x})$ implies $B_\mathbf{x}$ is a prefix of $A_\mathbf{x}$. $A \neq B$ implies $A_\mathbf{x} \neq B_\mathbf{x}$, thus $B_\mathbf{x}$ is a proper prefix of $A_\mathbf{x}$. Then $\underline{\gamma^\mathbf{c}(\overset{\smile}{A_\mathbf{x}})}(B_\mathbf{x}) = \emptyset$, and thus $\underline{\gamma^\mathbf{c}(A)}(B_\mathbf{x}) = \emptyset$. Thus $\gamma^\mathbf{c}(A) \neq \gamma^\mathbf{c}(B)$. ◻

Proof of Lemma 3.5.3

Proof. Let $A \equiv \overset{\smile}{A_\mathbf{x}} \cdot A_\mathbf{w} \cdot \overset{\frown}{A_\mathbf{e}}$. If $\mathit{trunc}_{i,j}(A) = A$, then the lemma follows.

Let $\mathit{trunc}_{i,j}(A) = \overset{\smile}{C_\mathbf{x}} \cdot * \cdot \overset{\frown}{C_\mathbf{e}}$, where $C_\mathbf{x} = \mathit{prefix}_i(A_\mathbf{x})$ and $C_\mathbf{e} = \mathit{prefix}_j(A_\mathbf{e})$. Then $C_\mathbf{x} \cdot \mathit{drop}_i(A_\mathbf{x}) = A_\mathbf{x}$ and $C_\mathbf{e} \cdot \mathit{drop}_j(A_\mathbf{e}) = A_\mathbf{e}$. We have

$$\gamma^\mathbf{c}(\overset{\smile}{\mathit{drop}_i(A_\mathbf{x})} \cdot A_\mathbf{w} \cdot \overset{\frown}{\mathit{drop}_j(A_\mathbf{e})}) \subseteq \mathbf{CtxtT} \setminus \{\mathit{errF}\} = \gamma^\mathbf{c}(*).$$

Prefixing $\overset{\smile}{C_\mathbf{x}}$ and appending $\overset{\frown}{C_\mathbf{e}}$ to both sides of $\gamma^\mathbf{c}(\overset{\smile}{\mathit{drop}_i(A_\mathbf{x})} \cdot A_\mathbf{w} \cdot \overset{\frown}{\mathit{drop}_j(A_\mathbf{e})}) \subseteq \gamma^\mathbf{c}(*)$, we get $\gamma^\mathbf{c}(\overset{\smile}{A_\mathbf{x}} \cdot A_\mathbf{w} \cdot \overset{\frown}{A_\mathbf{e}}) \subseteq \gamma^\mathbf{c}(\overset{\smile}{C_\mathbf{x}} \cdot * \cdot \overset{\frown}{C_\mathbf{e}})$. ◻

Chapter 4

Soundness and Precision

This chapter presents proofs that pointer analysis using context transformations is sound, the transformer string abstraction is sound, and compares the precision of the two abstractions of context transformations.

Section 4.1 defines the meaning of analysis precision, and states the precision difference between the explicit string and transformer string instantiations of the rules in Figure 3.3. Section 4.2 presents lemmas that are useful in subsequent sections. Section 4.3 develops the proof that the context transformation instantiations and the transformer string abstraction are sound. Section 4.4 develops the proof that the transformer string abstraction is as precise as the explicit string abstraction under call-site and object sensitivity.

4.1 Main Results

Each of the two abstractions is parameterized by two levels of context sensitivity, m for method contexts and h for heap contexts, which determine the truncation length of strings defined in Figures 3.5 and 3.6. When we compare the precision of the two abstractions, we assume that both abstractions have been instantiated with the same values of the parameters m and h.

Definition 4.1.1. Let the *context-sensitive result* of an analysis be denoted $\mathcal{C}^{\square\square}_{m,h}$, and be a set that consists of $\mathsf{pts}^{\square}$, $\mathsf{hpts}^{\square}$, $\mathsf{call}^{\square}$, and $\mathsf{reach}^{\square}$ facts in the minimum model of an instantiation of the rules in Figure 3.3. The first superscript of $\mathcal{C}$ is either **c**, **t**, **e** to indicate an instantiation using the context transformation domain, the transformer string abstraction, or the explicit string abstraction, respectively. The second superscript is either **s**, **o**, or **y** to indicate call-site, object, or type sensitivity, respectively. Integers m and h are levels of method and heap contexts, respectively.

Definition 4.1.2. The *context-insensitive projections* of an analysis are relations derived by the following rules, where the context attribute is projected out:

$$\begin{aligned}
&\mathsf{pts}^{\square\mathbf{i}}(Y, H) \Leftarrow \exists A : \mathsf{pts}^{\square}(Y, H, A).\\
&\mathsf{hpts}^{\square\mathbf{i}}(G, F, H) \Leftarrow \exists A : \mathsf{hpts}^{\square}(G, F, H, A).\\
&\mathsf{call}^{\square\mathbf{i}}(I, P) \Leftarrow \exists A : \mathsf{call}^{\square}(I, P, A)\\
&\mathsf{reach}^{\square\mathbf{i}}(P) \Leftarrow \exists A : \mathsf{reach}^{\square}(P, A)
\end{aligned}$$

Let the *context-insensitive result* of an analysis be denoted $\mathcal{A}^{\square\square}_{m,h}$ and be a set that consists of $\mathsf{pts}^{\square\mathbf{i}}$, $\mathsf{hpts}^{\square\mathbf{i}}$, $\mathsf{call}^{\square\mathbf{i}}$, and $\mathsf{reach}^{\square\mathbf{i}}$ facts in the minimum model of an instantiation of the rules in Figure 3.3 and the rules stated above that derive the context-insensitive projections. The superscripts and subscripts have the same meaning as in the definition of $\mathcal{C}$.

For one instantiation to be *as precise* compared to another is to have context-insensitive projections of pts, hpts, and call that are subsets of the other.

We state the main precision result:

Theorem 4.1.1. *Call-site- and object-sensitive transformer string instantiations are as precise as explicit string instantiations at the same levels of method and heap contexts. That is, for all m and h, $\mathcal{A}^{\mathbf{t}\square}_{m,h} \subseteq \mathcal{A}^{\mathbf{e}\square}_{m,h}$ for $\square \in \{\mathbf{s}, \mathbf{o}\}$.*

```
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
  }
}
```

Explicit string	Transformer string	Rule
$\mathsf{reach}(\mathtt{main}, \mathtt{entry})$	$\mathsf{reach}(\mathtt{main}, \mathtt{entry})$	Entry
$\mathsf{call}(\mathtt{main}, \mathtt{m}, (\mathtt{entry}, \mathtt{m1}))$	$\mathsf{call}(\mathtt{main}, \mathtt{m}, \widehat{\mathtt{m1}})$	Static
$\mathsf{call}(\mathtt{main}, \mathtt{m}, (\mathtt{entry}, \mathtt{m2}))$	$\mathsf{call}(\mathtt{main}, \mathtt{m}, \widehat{\mathtt{m2}})$	Static
$\mathsf{reach}(\mathtt{m}, \mathtt{m1})$	$\mathsf{reach}(\mathtt{m}, \mathtt{m1})$	Reach
$\mathsf{reach}(\mathtt{m}, \mathtt{m2})$	$\mathsf{reach}(\mathtt{m}, \mathtt{m2})$	Reach
$\mathsf{pts}(\mathtt{h}, \mathtt{h1}, (\mathtt{m1}, \mathtt{m1}))$	$\mathsf{pts}(\mathtt{h}, \mathtt{h1}, \epsilon)$	New
$\mathsf{pts}(\mathtt{h}, \mathtt{h1}, (\mathtt{m2}, \mathtt{m2}))$		New
$\mathsf{call}(\mathtt{m}, \mathtt{id}, (\mathtt{m1}, \mathtt{id1}))$	$\mathsf{call}(\mathtt{m}, \mathtt{id}, \widehat{\mathtt{id1}})$	Static
$\mathsf{call}(\mathtt{m}, \mathtt{id}, (\mathtt{m2}, \mathtt{id1}))$		Static
$\mathsf{reach}(\mathtt{id}, \mathtt{id1})$	$\mathsf{reach}(\mathtt{id}, \mathtt{id1})$	Reach
$\mathsf{pts}(\mathtt{p}, \mathtt{h1}, (\mathtt{m1}, \mathtt{id1}))$	$\mathsf{pts}(\mathtt{p}, \mathtt{h1}, \widehat{\mathtt{id1}})$	Param
$\mathsf{pts}(\mathtt{p}, \mathtt{h1}, (\mathtt{m2}, \mathtt{id1}))$		Param
$\mathsf{pts}(\mathtt{r}, \mathtt{h1}, (\mathtt{m1}, \mathtt{m1}))$	$\mathsf{pts}(\mathtt{r}, \mathtt{h1}, \epsilon)$	Return
$\mathsf{pts}(\mathtt{r}, \mathtt{h1}, (\mathtt{m2}, \mathtt{m1}))$		Return
$\mathsf{pts}(\mathtt{r}, \mathtt{h1}, (\mathtt{m1}, \mathtt{m2}))$		Return
$\mathsf{pts}(\mathtt{r}, \mathtt{h1}, (\mathtt{m2}, \mathtt{m2}))$		Return
$\mathsf{pts}(\mathtt{x}, \mathtt{h1}, (\mathtt{m1}, \mathtt{entry}))$	$\mathsf{pts}(\mathtt{x}, \mathtt{h1}, \widecheck{\mathtt{m1}})$	Return
$\mathsf{pts}(\mathtt{x}, \mathtt{h1}, (\mathtt{m2}, \mathtt{entry}))$		Return
$\mathsf{pts}(\mathtt{y}, \mathtt{h1}, (\mathtt{m1}, \mathtt{entry}))$		Return
$\mathsf{pts}(\mathtt{y}, \mathtt{h1}, (\mathtt{m2}, \mathtt{entry}))$	$\mathsf{pts}(\mathtt{y}, \mathtt{h1}, \widecheck{\mathtt{m2}})$	Return

Figure 4.1: Example illustrating the precision difference between the explicit string and transformer string abstractions using $m = 1$ and $h = 1$ levels of call-site sensitivity.

The proof is in Section 4.4.

The converse is not true. The counterexample that shows that the explicit string instantiation is not as precise as the transformer string instantiation is in Figure 4.1, which uses one level of heap and method contexts under call-site sensitivity. The first and second columns contain derived facts using the explicit string abstraction and the transformer string abstraction, respectively. The third column states the deduction rule used in the derivation. With explicit strings, the heap objects returned from call sites `m1` and `m2` are not differentiated.

The precision result holds for call-site- and object-sensitive analysis, but it is not true for type-sensitive analysis. Figures 4.2 and 4.3 are examples where transformer strings are less precise than explicit strings under type-sensitive analysis. The derivation of reach facts has been omitted for brevity. The imprecision arises from multiple allocation sites mapping to a single type context: that is, *classOf* is not one-to-one. Section 4.4.3 introduces a concept called *derivability*, which is a property satisfied by call-site- and object-sensitive analysis and used in the proof of the precision result, but is not satisfied by type-sensitive analysis.

4.1.1 Outline of Proofs

In summary, this chapter develops the proofs of the following inequalities:

$$\mathcal{A}^{\mathbf{e}\square}_{\infty,\infty} \subseteq \mathcal{A}^{\mathbf{c}\square} \subseteq \mathcal{A}^{\mathbf{t}\square}_{m,h} \subseteq \mathcal{A}^{\mathbf{e}\square}_{m,h}.$$

$\mathcal{A}^{\mathbf{e}\square}_{\infty,\infty}$ is the analysis result of a context-sensitive analysis using strings of contexts of unbounded length. We first show that analysis using context transformations is sound: that is $\mathcal{A}^{\mathbf{e}\square}_{\infty,\infty} \subseteq \mathcal{A}^{\mathbf{c}\square}$. Then, the result that analysis using transformer strings is sound (i.e. $\mathcal{A}^{\mathbf{c}\square} \subseteq \mathcal{A}^{\mathbf{t}\square}_{m,h}$) follows from the fact that the truncation function defined in Section 3.5.2 is conservative. The last inequality, $\mathcal{A}^{\mathbf{t}\square}_{m,h} \subseteq \mathcal{A}^{\mathbf{e}\square}_{m,h}$, which expresses the precision difference between the two abstractions, is true only for call-site- and object-sensitive analysis.

```
class S {
  public void s() {
    T p = new T(); // Ts
    p.g();     // sg
  }
}

class T {
  public void t() {
    T q = new T(); // Tt
    q.h();     // th
  }
  public void g() {
    T a = new T();      // Tg
    Object x = a.id(a); // gid
  }
  public void h() {
    T b = new T();      // Th
    Object y = b.id(b); // hid
  }
  Object id(Object o) {
    return o;
  }
}

class M {
  public static void main(String[] args) {
    S s = new S(); // M1
    s.s();         // ms

    T t = new T(); // M2
    t.t();         // mt
  }
}
```

Figure 4.2: Example illustrating the precision difference between the explicit string and transformer string abstractions using $m = 2$ and $h = 1$ levels of type sensitivity (Part 1).

Explicit string	Transformer string	Rule
$\mathsf{pts}(\mathtt{s}, \mathtt{M1}, (\mathtt{entry}, \mathtt{entry}))$	$\mathsf{pts}(\mathtt{s}, \mathtt{M1}, \epsilon)$	New
$\mathsf{pts}(\mathtt{t}, \mathtt{M2}, (\mathtt{entry}, \mathtt{entry}))$	$\mathsf{pts}(\mathtt{t}, \mathtt{M2}, \epsilon)$	New
$\mathsf{call}(\mathtt{ms}, \mathtt{s}, (\mathtt{entry}, \mathtt{M}\cdot\mathtt{entry}))$	$\mathsf{call}(\mathtt{ms}, \mathtt{s}, \widehat{\mathtt{M}})$	Virt
$\mathsf{call}(\mathtt{mt}, \mathtt{t}, (\mathtt{entry}, \mathtt{M}\cdot\mathtt{entry}))$	$\mathsf{call}(\mathtt{mt}, \mathtt{t}, \widehat{\mathtt{M}})$	Virt
$\mathsf{pts}(\mathtt{p}, \mathtt{Ss}, (\mathtt{M}, \mathtt{M}\cdot\mathtt{entry}))$	$\mathsf{pts}(\mathtt{p}, \mathtt{Ss}, \epsilon)$	New
$\mathsf{pts}(\mathtt{q}, \mathtt{Tt}, (\mathtt{M}, \mathtt{M}\cdot\mathtt{entry}))$	$\mathsf{pts}(\mathtt{q}, \mathtt{Tt}, \epsilon)$	New
$\mathsf{call}(\mathtt{sg}, \mathtt{g}, (\mathtt{M}\cdot\mathtt{entry}, \mathtt{S}\cdot\mathtt{M}))$	$\mathsf{call}(\mathtt{sg}, \mathtt{g}, \widehat{\mathtt{S}})$	Virt
$\mathsf{call}(\mathtt{th}, \mathtt{h}, (\mathtt{M}\cdot\mathtt{entry}, \mathtt{T}\cdot\mathtt{M}))$	$\mathsf{call}(\mathtt{th}, \mathtt{h}, \widehat{\mathtt{T}})$	Virt
$\mathsf{pts}(\mathtt{a}, \mathtt{Tg}, (\mathtt{S}, \mathtt{S}\cdot\mathtt{M}))$	$\mathsf{pts}(\mathtt{a}, \mathtt{Tg}, \epsilon)$	New
$\mathsf{pts}(\mathtt{b}, \mathtt{Th}, (\mathtt{T}, \mathtt{T}\cdot\mathtt{M}))$	$\mathsf{pts}(\mathtt{b}, \mathtt{Th}, \epsilon)$	New
$\mathsf{call}(\mathtt{gid}, \mathtt{id}, (\mathtt{S}\cdot\mathtt{M}, \mathtt{T}\cdot\mathtt{S}))$	$\mathsf{call}(\mathtt{gid}, \mathtt{id}, \widehat{\mathtt{T}})$	Virt
$\mathsf{call}(\mathtt{hid}, \mathtt{id}, (\mathtt{T}\cdot\mathtt{M}, \mathtt{T}\cdot\mathtt{T}))$	$\mathsf{call}(\mathtt{hid}, \mathtt{id}, \widehat{\mathtt{T}})$	Virt
$\mathsf{pts}(\mathtt{o}, \mathtt{Tg}, (\mathtt{S}, \mathtt{T}\cdot\mathtt{S}))$	$\mathsf{pts}(\mathtt{o}, \mathtt{Tg}, \widehat{\mathtt{T}})$	Param
$\mathsf{pts}(\mathtt{o}, \mathtt{Th}, (\mathtt{T}, \mathtt{T}\cdot\mathtt{T}))$	$\mathsf{pts}(\mathtt{o}, \mathtt{Th}, \widehat{\mathtt{T}})$	Param
$\mathsf{pts}(\mathtt{x}, \mathtt{Tg}, (\mathtt{S}, \mathtt{S}\cdot\mathtt{M}))$	$\mathsf{pts}(\mathtt{x}, \mathtt{Tg}, \epsilon)$	Return
	$\mathsf{pts}(\mathtt{x}, \mathtt{Th}, \epsilon)$	Return
$\mathsf{pts}(\mathtt{y}, \mathtt{Th}, (\mathtt{T}, \mathtt{T}\cdot\mathtt{M}))$	$\mathsf{pts}(\mathtt{y}, \mathtt{Th}, \epsilon)$	Return
	$\mathsf{pts}(\mathtt{y}, \mathtt{Tg}, \epsilon)$	Return

Figure 4.3: Example illustrating the precision difference between the explicit string and transformer string abstractions using $m = 2$ and $h = 1$ levels of type sensitivity (Part 2).

To prove inequalities between different abstractions, specifically, $\mathcal{A}^{\mathbf{e}\Box}_{\infty,\infty} \subseteq \mathcal{A}^{\mathbf{c}\Box}$ and $\mathcal{A}^{\mathbf{t}\Box}_{m,h} \subseteq \mathcal{A}^{\mathbf{e}\Box}_{m,h}$, we define and use a *concretization function* that transforms context transformations and transformer strings into sets of explicit strings. By showing that a context transformation instantiation's analysis result *concretizes* to a superset of an unbounded explicit string instantiation's context-sensitive analysis result, we establish the same inequality for the context-insensitive analysis results. Likewise, to prove the precision property, we show that the analysis result of a transformer string instantiation concretizes to a subset of the analysis result of an explicit string instantiation at the same levels of method and heap contexts.

4.2 General Properties

This section describes general properties and results that are useful in subsequent sections.

Section 3.5.2 defined a notation for converting a string $M \equiv m_1 \cdot \ldots \cdot m_n \in \mathbf{Ctxt}^*$ into transformer strings $\widehat{M}$ and $\breve{M}$:

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

A consequence of reversing the order of elements in $\widehat{M}$ is that a string of entries $\widehat{A_{\mathbf{e}}}$ *matches* a string of exits $\breve{B_{\mathbf{x}}}$, that is $match(\widehat{A_{\mathbf{e}}} \cdot \breve{B_{\mathbf{x}}}) \neq \bot$, if and only if one of $A_{\mathbf{e}}$ and $B_{\mathbf{x}}$ is a prefix of the other.

Definition 4.2.1. Let $X \cong Y$ iff X is a prefix of Y or Y is a prefix of X. In other words, $\mathit{prefix}_{min(\|X\|,\|Y\|)}(X) = \mathit{prefix}_{min(\|X\|,\|Y\|)}(Y)$.

Lemma 4.2.1. *$match(\widehat{A_{\mathbf{e}}} \cdot \breve{B_{\mathbf{x}}}) \neq \bot$ iff $A_{\mathbf{e}} \cong B_{\mathbf{x}}$.*

Proof. By inspection of *match*. □

4.3 Soundness

This section shows that pointer analysis using context transformations is sound. We show that the context transformation instantiations defined in Figure 3.4 concretize to a superset of their respective *untruncated explicit string instantiations*.

Pointer analysis using untruncated explicit strings is known to be sound, but the analysis (including its context-insensitive projections) is not computable in programs with recursive function calls [31]. Untruncated explicit string instantiations are obtained from Figure 3.5 by letting levels of context sensitivity m and h be ∞ (the function *prefix* becomes the identity function).

4.3.1 Untruncated Concretization

Definition 4.3.1. Let γ_∞ concretize context transformations into untruncated explicit strings:

$$\gamma_\infty(A) \equiv \{(M, A(M)) \mid M \in \mathbf{Ctxt}^* \wedge A(M) \neq \mathit{err}\}.$$

Definition 4.3.2. Let $\gamma^{\mathbf{M}}_\infty$ concretize partial method contexts into untruncated method contexts:

$$\gamma^{\mathbf{M}}_\infty(N) \equiv \{N \cdot M \mid M \in \mathbf{Ctxt}^*\}.$$

Definition 4.3.3. Given an interpretation $\mathcal{I}$ of derived relations of an untruncated transformer string instantiation of the rules in Figure 3.3 (i.e., $\mathsf{pts}^{\mathbf{t}}$, $\mathsf{hpts}^{\mathbf{t}}$, $\mathsf{call}^{\mathbf{t}}$, and $\mathsf{reach}^{\mathbf{t}}$), let the *untruncated explicit string concretization of the interpretation*, denoted $\gamma^{\mathbf{I}}_\infty(\mathcal{I})$, be a set of facts defined as follows:

$$\begin{aligned}\gamma^{\mathbf{I}}_\infty(\mathcal{I}) \equiv\ & \{\mathsf{pts}^{\mathbf{e}}(Y,H,B) \mid \mathsf{pts}^{\mathbf{t}}(Y,H,A) \wedge B \in \gamma_\infty(A)\}\\ & \cup \{\mathsf{hpts}^{\mathbf{e}}(G,F,H,B) \mid \mathsf{hpts}^{\mathbf{t}}(G,F,H,A) \wedge B \in \gamma_\infty(A)\}\\ & \cup \{\mathsf{call}^{\mathbf{e}}(I,Q,B) \mid \mathsf{call}^{\mathbf{t}}(I,Q,A) \wedge B \in \gamma_\infty(A)\}\\ & \cup \{\mathsf{reach}^{\mathbf{e}}(P,M) \mid \mathsf{reach}^{\mathbf{t}}(P,N) \wedge M \in \gamma^{\mathbf{M}}_\infty(N)\}\end{aligned}$$

Note that the context transformation instantiation never derives an $errF$ transformation, and for all non-$errF$ transformations A, $\gamma_\infty(A) \neq \emptyset$. Thus, if $\gamma_\infty^{\mathbf{I}}(\mathcal{C}^{\mathbf{c}\square}) \supseteq \mathcal{C}^{\mathbf{e}\square}_{\infty,\infty}$, then for any fact containing an explicit string B in $\mathcal{C}^{\mathbf{e}\square}_{\infty,\infty}$, there must exist a corresponding fact in $\mathcal{C}^{\mathbf{c}\square}$ containing a context transformation A such that $B \in \gamma_\infty(A)$. Likewise, if fact $\mathsf{reach}^{\mathbf{e}}(P, M)$ is in $\mathcal{C}^{\mathbf{e}\square}_{\infty,\infty}$, then there must exist a partial method context N such that $\mathsf{reach}^{\mathbf{c}}(P, N)$ is in $\mathcal{C}^{\mathbf{c}\square}$ and $M \in \gamma_\infty^{\mathbf{M}}(N)$. Thus, $\gamma_\infty^{\mathbf{I}}(\mathcal{C}^{\mathbf{c}\square}) \supseteq \mathcal{C}^{\mathbf{e}\square}_{\infty,\infty}$ implies $\mathcal{A}^{\mathbf{c}\square} \supseteq \mathcal{A}^{\mathbf{e}\square}_{\infty,\infty}$.

4.3.2 Superset Theorem

We show that $\mathcal{C}^{\mathbf{c}\square}$ and $\mathcal{C}^{\mathbf{e}\square}_{\infty,\infty}$ satisfy $\gamma_\infty^{\mathbf{I}}(\mathcal{C}^{\mathbf{c}\square}) \supseteq \mathcal{C}^{\mathbf{e}\square}_{\infty,\infty}$ by induction on the derivation of explicit and transformer strings according to the rules in Figure 3.3. The following are *operations* on explicit and transformer strings: functions $inv^{\square}$, $target^{\square}$, $record^{\square}$, $merge^{\square\square}$, and $merge\_s^{\square\square}$. The first two arguments of $\mathsf{any}^{\square}$ and $\mathsf{comp}^{\square}$ literals in Figure 3.3 always appear as arguments in a different literal in the bodies of rules. Thus, $\mathsf{any}^{\square}$ and $\mathsf{comp}^{\square}$ can be thought of as operations from their first two arguments to their last arguments. We define convenience functions for concretizing the last arguments of these predicates:

$$\gamma_\infty^{\mathsf{comp}}(A, B) \equiv \bigcup\{\gamma_\infty(C) \mid \mathsf{comp}^{\mathbf{c}}(A, B, C)\}$$
$$\gamma_\infty^{\mathsf{any}}(P, A) \equiv \bigcup\{\gamma_\infty(B) \mid \mathsf{any}^{\mathbf{c}}(P, A, B)\}$$

The next three lemmas state that operations on explicit strings and transformer strings preserve the superset inequality.

Lemma 4.3.1. *For all $A \in$ **CtxtT** and methods P, the following inequalities hold:*

1. $\gamma_\infty(inv^{\mathbf{c}}(A)) \supseteq \{inv^{\mathbf{e}}(A') \mid A' \in \gamma_\infty(A)\}$.

2. $\gamma_\infty^{\mathsf{any}}(P, A) \supseteq \{B \mid A' \in \gamma_\infty(A) \wedge \mathsf{any}^{\mathbf{e}}(P, A', B)\}$.

3. $\gamma_\infty^{\mathbf{M}}(target^{\mathbf{c}}(A)) \supseteq \{target^{\mathbf{e}}(B) \mid B \in \gamma_\infty(A)\}$.

Proof.

1. We must show that $\gamma_\infty(A^{-1}) \subseteq \{(Y, X) \mid (X, Y) \in \gamma_\infty(A)\}$:

   Let $(X, Y) \in \gamma_\infty(A)$. Then, $A(X) = Y$ and $Y \neq errF$. Thus, $A^{-1}(Y) = X$ and $(Y, X) \in \gamma_\infty(A^{-1})$.

2. We must show that

   $$\bigcup\{\gamma_\infty(A \,;C) \mid C \in \mathbf{CtxtT}\} \supseteq \{(X, M) \mid (X, Y) \in \gamma_\infty(A) \wedge \mathsf{reach}^{\mathbf{e}}(P, M)\}$$

   for any interpretation of predicate $\mathsf{reach}^{\mathbf{e}}$:

   Let $(X, Y) \in \gamma_\infty(A)$ and let $M \in \mathbf{Ctxt}^*$. Then there exists $C \in \mathbf{CtxtT}$ such that $(A\,;C)(X) = M$. Thus $(X, M) \in \bigcup\{\gamma_\infty(A\,;C) \mid C \in \mathbf{CtxtT}\}$. Thus, $\bigcup\{\gamma_\infty(A\,;C) \mid C \in \mathbf{CtxtT}\} \supseteq \{(X, M) \mid (X, Y) \in \gamma_\infty(A) \wedge M \in \mathbf{Ctxt}^*\}$ and the lemma follows.

3. We must show that $\gamma_\infty^{\mathbf{M}}(Y) \supseteq \{Y' \mid (X', Y') \in \gamma_\infty(\check{X}\,;\widehat{Y})\}$:

   Let $(X', Y') \in \gamma_\infty(\check{X}\,;\widehat{Y})$. Then X is a prefix of X' and Y is a prefix of Y'. Thus, $Y' \in \gamma_\infty^{\mathbf{M}}(Y)$.

$\square$

Lemma 4.3.2. *For all $A, B \in \mathbf{CtxtT}$, the following inequality holds:*

$$\gamma_\infty^{\mathsf{comp}}(A, B) \supseteq \{C' \mid A' \in \gamma_\infty(A) \wedge B' \in \gamma_\infty(B) \wedge \mathsf{comp}^{\mathbf{e}}(A', B', C')\}.$$

Proof. The right-hand-side of the inequality is the set $\{(X, Z) \mid (X, Y) \in \gamma_\infty(A) \wedge (Y, Z) \in \gamma_\infty(B)\}$. Let $(X, Z) \in \{(X, Z) \mid (X, Y) \in \gamma_\infty(A) \wedge (Y, Z) \in \gamma_\infty(B)\}$. Then there exists Y such that $(X, Y) \in \gamma_\infty(A)$ and $(Y, Z) \in \gamma_\infty(B)$. Thus $(A\,;B)(X) = Z$ and $(X, Z) \in \gamma_\infty^{\mathsf{comp}}(A, B)$. $\square$

Lemma 4.3.3. *For all $B \in \mathbf{CtxtT}$, heap allocation sites H, invocation sites I, and partial method contexts $N \in \mathbf{Ctxt}^*$, the following inequalities hold:*

1. $\gamma_\infty(\mathit{record}^{\mathbf{c}}(N)) \supseteq \{\mathit{record}^{\mathbf{e}}(M) \mid M \in \gamma_\infty^{\mathbf{M}}(N)\}$.

2. $\gamma_\infty(\mathit{merge}^{\mathbf{c}\square}(H, I, B)) \supseteq \{\mathit{merge}^{\mathbf{e}\square}(H, I, (X', Y')) \mid (X', Y') \in \gamma_\infty(B)\}$.

3. $\gamma_\infty(\mathit{merge\_s}^{\mathbf{c}\square}(I, N)) \supseteq \{\mathit{merge\_s}^{\mathbf{e}\square}(I, M) \mid M \in \gamma_\infty^{\mathbf{M}}(N)\}$.

Proof.

1. $\gamma_\infty(\epsilon) = \{(M, M) \mid M \in \mathbf{Ctxt}^*\} \supseteq \{(M, M) \mid M \in \gamma_\infty^{\mathbf{M}}(N)\}$ follows trivially from Definition 4.3.1.

2. (a) **Call-site sensitivity**. We must show that $\gamma_\infty(B^{-1} ; B ; \widehat{I}) \supseteq \{(Y', I \cdot Y') \mid (X', Y') \in \gamma_\infty(B)\}$:

   Let $(Y, I \cdot Y) \in \{(Y', I \cdot Y') \mid (X', Y') \in \gamma_\infty(B)\}$. Thus $(X, Y) \in \gamma_\infty(B)$ for some X. Thus $B(X) = Y$ and $(B^{-1} ; B)(Y) = Y$. Thus $(B^{-1} ; B ; \widehat{I})(Y) = I \cdot Y$ and $(Y, I \cdot Y) \in \gamma_\infty(B^{-1} ; B ; \widehat{I})$.

   (b) **Object sensitivity**. We must show that $\gamma_\infty(B^{-1} ; \widehat{H}) \supseteq \{(Y', H \cdot X') \mid (X', Y') \in \gamma_\infty(B)\}$:

   Let $(Y, H \cdot X) \in \{(Y', H \cdot X') \mid (X', Y') \in \gamma_\infty(B)\}$. Thus $(X, Y) \in \gamma_\infty(B)$. Thus $B(X) = Y$ and $(B^{-1})(Y) = X$. Thus $(B^{-1} ; \widehat{H})(Y) = H \cdot X$ and $(Y, H \cdot X) \in \gamma_\infty(B^{-1} ; \widehat{H})$.

   (c) **Type sensitivity**: Same reasoning as 2b.

3. (a) **Call-site sensitivity**: Same reasoning as 2a.

   (b) **Object sensitivity**. We must show that $\gamma_\infty(\breve{N} \cdot \widehat{N}) \supseteq \{(M, M) \mid M \in \gamma_\infty^{\mathbf{M}}(N)\}$:

   Let $(Y, Y) \in \{(M, M) \mid M \in \gamma_\infty^{\mathbf{M}}(N)\}$. Then, N is a prefix of Y. Thus, $(Y, Y) \in \gamma_\infty(\breve{N} \cdot \widehat{N})$.

   (c) **Type sensitivity**: Same reasoning as 3b.

□

Theorem 4.3.4. *$\mathcal{A}^{\mathbf{e}\square}_{\infty,\infty} \subseteq \mathcal{A}^{\mathbf{c}\square}$, for all flavours of context sensitivity.*

Proof. $\mathcal{C}^{\mathbf{e}\square}_{\infty,\infty} \subseteq \mathcal{C}^{\mathbf{c}\square}$ follows from Lemmas 4.3.1, 4.3.2, and 4.3.3 by induction on the derivation of explicit and transformer strings according to the rules in Figure 3.3. Thus $\mathcal{A}^{\mathbf{e}\square}_{\infty,\infty} \subseteq \mathcal{A}^{\mathbf{c}\square}$. □

Theorem 4.3.5. $\mathcal{A}^{\mathbf{c}\square} \subseteq \mathcal{A}^{\mathbf{t}\square}_{m,h}$, *for all flavours of context sensitivity.*

Proof. Follows from the lemmas that *match* and *trunc* are conservative with respect to $\gamma^{\mathbf{c}}$ (Lemmas 3.5.1 and 3.5.3) and by inspection of the definitions of non-logical symbols in Figures 3.4 and 3.6. □

4.4 Precision

The proof for the precision theorem proceeds similarly to that of the soundness theorem: first we define a concretization function and then we show that the operations on explicit and transformer strings preserve a subset inequality with respect to the concretization function. However, the two steps are more complicated. In order to prove the subset inequality, the concretization function must be dependent on the *reachable method contexts* of methods (an interpretation of $\mathsf{reach}^{\mathbf{e}}$). The definition of the concretization function requires a property of transformer strings called *consistency* that associates a *source* and *destination* method to all prefixes and suffixes of transformer strings. The proof for the subset inequality requires another property, called *derivability*, that is a constraint between the transformer strings derived by the rules in Figure 3.3 and the partial method contexts derived by the same rules.

This section applies only to call-site- and object-sensitive analysis.

4.4.1 Consistency

The *parent method* of a call site or allocation site is the method that contains the site. The entry point context `entry`, which technically is neither a call site or an allocation site in a program, requires special treatment:

Definition 4.4.1. `entry` is a special method context for entry points. Let $\mathit{parent}(\texttt{entry}) \equiv \texttt{entryM}$, where `entryM` is a dummy method. We assume that `entryM` is reachable with an empty method context in an explicit string instantiation: that is, we have a clause "$\mathsf{reach}^{\mathsf{e}}(\texttt{entryM}, [\,])$." in every explicit string instantiation. When quantifying over all "methods", we are quantifying over all methods of a program, which do not include `entryM`.

Definition 4.4.2. Transformer strings have *source* and *destination* methods that are implied by the relation in which the strings appears:

- In $\mathsf{pts}(Y, H, A)$, the source of A is H's parent method and the destination is Y's parent method.
- In $\mathsf{hpts}(G, F, H, A)$, the source of A is H's parent method and the destination is G's parent method.
- In $\mathsf{call}(I, P, A)$, the source of A is I's parent method and the destination is method P.

Definition 4.4.3. Under call-site and object sensitivity, let the *remainder* of a partial method context M of a method P be a method defined as follows:

$$\mathit{rem}(P, M) \equiv \begin{cases} P & \text{if } M = \epsilon \\ \mathit{parent}(\mathit{last}(M)) & \text{otherwise} \end{cases}.$$

Using *rem* we wish to associate all prefixes and suffixes of a transformer string with source and destination methods. For example, given a sequence of exit transformations $\breve{a}\cdot\breve{b}$ from P (the source) to Q (the destination), let the following source and destination methods be implied for each prefix of $\breve{a}\cdot\breve{b}$:

- ϵ from P to $\mathit{rem}(P, \epsilon) = P$;

- $\breve{a}$ from P to $\mathit{rem}(P, a) = \mathit{parent}(a)$;
- $\breve{a} \cdot \breve{b}$ from P to $\mathit{rem}(P, a \cdot b) = \mathit{parent}(b)$.

For this definition to be *consistent*, we require $\mathit{parent}(b) = Q$. In general, for a transformer string $\overset{\smile}{A_{\mathbf{x}}} \cdot \widehat{A_{\mathbf{e}}}$ from P to Q, we require $\mathit{rem}(P, A_{\mathbf{x}}) = \mathit{rem}(Q, A_{\mathbf{e}})$. This restriction does not apply to strings with a wildcard: that is, the methods on either "sides" of the wildcard do not have to be the same method.

Definition 4.4.4. A transformer string $A \equiv \overset{\smile}{A_{\mathbf{x}}} \cdot A_{\mathbf{w}} \cdot \widehat{A_{\mathbf{e}}}$ from method P to Q is *consistent* if $A_{\mathbf{w}} = *$, or if $\mathit{rem}(P, A_{\mathbf{x}}) = \mathit{rem}(Q, A_{\mathbf{e}})$. If A is consistent and $A_{\mathbf{w}} = \epsilon$, let $\mathit{base}(A, P, Q) \equiv \mathit{rem}(P, A_{\mathbf{x}}) = \mathit{rem}(Q, A_{\mathbf{e}})$ be A's *base method.*

In order to show that all transformer strings derived by the deduction rules are consistent, we require another property that all derived partial method contexts are non-empty: that is, we never derive a fact $\mathsf{call}^{\mathbf{t}}(I, P, \overset{\smile}{A_{\mathbf{x}}} \cdot A_{\mathbf{w}} \cdot \widehat{A_{\mathbf{e}}})$ such that $\|A_{\mathbf{e}}\| = 0$. This requirement is the rationale for having a special entry point method context `entry`, and deriving $\mathsf{reach}^{\mathbf{t}}(\texttt{main}, [\texttt{entry}])$ instead of $\mathsf{reach}^{\mathbf{t}}(\texttt{main}, [\,])$.

Lemma 4.4.1. *None of the transformer string instantiations derive an empty partial method context.*

Proof. By inspection of functions $\mathit{merge}^{\mathbf{t}\square}$ and $\mathit{merge\_s}^{\mathbf{t}\square}$ in Figure 3.6, and the Entry rule in Figure 3.3. $\square$

The next three lemmas state that operations on transformer strings preserve the consistency property.

Lemma 4.4.2.

1. *$\widehat{\texttt{entry}}$ from `entryM` to `main` is consistent.*

2. *For all consistent transformer strings $A \in \mathbf{CtxtT}^{\mathbf{t}}_{i,j}$ from P to Q, $inv^{\mathbf{t}}(A)$ from Q to P is consistent.*

3. *For all consistent transformer strings $A \in \mathbf{CtxtT}^{\mathbf{t}}_{h,m}$ from P to any method, if $\mathsf{any}^{\mathbf{t}}(R, A, B)$ then B from P to R is consistent.*

Proof.

1. Follows trivially from Definition 4.4.1.

2. $A_{\mathbf{x}}^{-1} = A_{\mathbf{e}}$ and $A_{\mathbf{e}}^{-1} = A_{\mathbf{x}}$, and thus the lemma follows.

3. B must have a wildcard letter, and thus the lemma follows.

□

Lemma 4.4.3. *For all consistent transformer strings $A \equiv \overset{\smile}{A_{\mathbf{x}}} \cdot A_{\mathbf{w}} \cdot \widehat{A_{\mathbf{e}}} \in \mathbf{CtxtT}^{\mathbf{t}}_{i,j}$ from P to Q and $B \equiv \overset{\smile}{B_{\mathbf{x}}} \cdot B_{\mathbf{w}} \cdot \widehat{B_{\mathbf{e}}} \in \mathbf{CtxtT}^{\mathbf{t}}_{j,k}$ from Q to R such that $match(A \cdot B) \neq \bot$, $trunc_{i,k}(match(A \cdot B))$ is consistent.*

Proof. If a transformer string is truncated, then it contains a wildcard, and thus the lemma follows.

Suppose $match(A \cdot B)$ is not truncated. Either $\|A_{\mathbf{e}}\| \leq \|B_{\mathbf{x}}\|$ or $\|A_{\mathbf{e}}\| \geq \|B_{\mathbf{x}}\|$. Suppose $\|A_{\mathbf{e}}\| \leq \|B_{\mathbf{x}}\|$. Then, $match(A \cdot B)$ can be written as $\overset{\smile}{A_{\mathbf{x}}} \cdot \widehat{B'_{\mathbf{x}}} \cdot \widehat{B_{\mathbf{e}}}$, where $A_{\mathbf{e}} \cdot B'_{\mathbf{x}} \equiv B_{\mathbf{x}}$.

Suppose $\|B'_{\mathbf{x}}\| = 0$. Then, $A_{\mathbf{e}} = B_{\mathbf{x}}$. Then $rem(P, A_{\mathbf{x}}) = rem(Q, A_{\mathbf{e}}) = rem(Q, B_{\mathbf{x}}) = rem(R, B_{\mathbf{e}})$ because A and B are consistent. Thus, $\overset{\smile}{A_{\mathbf{x}}} \cdot \widehat{B_{\mathbf{e}}}$ is consistent.

Suppose $\|B'_{\mathbf{x}}\| > 0$. Then, $last(A_{\mathbf{x}} \cdot B'_{\mathbf{x}}) = last(B_{\mathbf{x}})$. Because B is consistent, we have $rem(Q, B_{\mathbf{x}}) = rem(R, B_{\mathbf{e}})$. Thus, $rem(P, A_{\mathbf{x}} \cdot B'_{\mathbf{x}}) = rem(R, B_{\mathbf{e}})$ and $\overset{\smile}{A_{\mathbf{x}}} \cdot \overset{\smile}{B'_{\mathbf{x}}} \cdot \widehat{B_{\mathbf{e}}}$ is consistent.

The case of $\|A_{\mathbf{e}}\| \geq \|B_{\mathbf{x}}\|$ follows the same reasoning as above. □

Lemma 4.4.4. *For all consistent transformer strings* $B \equiv \breve{B_{\mathbf{x}}} \cdot B_{\mathbf{w}} \cdot \widehat{B_{\mathbf{e}}} \in \mathbf{CtxtT}^{\mathbf{t}}_{h,m}$ *from* N *to* P*, heap allocation site* H *in* N*, invocation* I *in* P*, and* $M \in \mathbf{Ctxt}^*$ *such that* $\|M\| > 0$*,*

1. $\mathit{record}^{\mathbf{t}}(M)$ *from* P *to* P *is consistent,*
2. $\mathit{merge}^{\mathbf{t}\square}(H, I, B)$ *from* P *to* Q *is consistent, and*
3. $\mathit{merge\_s}^{\mathbf{t}\square}(I, M)$ *from* P *to* Q *is consistent.*

Proof.

1. Follows trivially from definitions.
2. (a) **Call-site sensitivity**. We must show that $\mathit{trunc}_{m,m}(\breve{B_{\mathbf{e}}} \cdot \widehat{B_{\mathbf{e}}} \cdot \widehat{I})$ is consistent:

   If truncation occurs, then the string is consistent. If $B_{\mathbf{e}} = \epsilon$, $\mathit{rem}(P, B_{\mathbf{e}}) = P$, and $\mathit{rem}(Q, I \cdot B_{\mathbf{e}}) = \mathit{parent}(I) = P$. Otherwise, $\mathit{rem}(P, B_{\mathbf{e}}) = \mathit{rem}(Q, I \cdot B_{\mathbf{e}}) = \mathit{parent}(\mathit{last}(B_{\mathbf{e}}))$.

   (b) **Object sensitivity**. We must show that $\breve{B_{\mathbf{e}}} \cdot B_{\mathbf{w}} \cdot \widehat{B_{\mathbf{x}}} \cdot \widehat{H}$ is consistent:

   If $B_{\mathbf{w}} = *$, then the string is consistent. Otherwise, suppose $B_{\mathbf{w}} = \epsilon$. We have $\mathit{rem}(N, B_{\mathbf{x}}) = \mathit{rem}(P, B_{\mathbf{e}})$ because B is consistent.

   Suppose $B_{\mathbf{x}} = \epsilon$. Then, $\mathit{rem}(Q, H \cdot B_{\mathbf{x}}) = \mathit{parent}(H) = N$ and $\mathit{rem}(P, B_{\mathbf{e}}) = \mathit{rem}(N, B_{\mathbf{x}}) = N$. Thus, $\mathit{rem}(P, B_{\mathbf{e}}) = \mathit{rem}(Q, H \cdot B_{\mathbf{x}})$.

   Suppose $B_{\mathbf{x}} \neq \epsilon$. We have $\mathit{rem}(Q, H \cdot B_{\mathbf{x}}) = \mathit{parent}(\mathit{last}(B_{\mathbf{x}})) = \mathit{rem}(N, B_{\mathbf{x}})$. Thus, $\mathit{rem}(P, B_{\mathbf{e}}) = \mathit{rem}(Q, H \cdot B_{\mathbf{x}})$.
3. (a) **Call-site sensitivity**. We must show that $\widehat{I}$ is consistent:

   $\mathit{rem}(P, \epsilon) = P$ and $\mathit{rem}(Q, I) = \mathit{parent}(I) = P$.

   (b) **Object sensitivity**. We must show that $\breve{M} \cdot \widehat{M}$ is consistent:

   Since $M \neq \epsilon$, $\mathit{rem}(P, M) = \mathit{rem}(Q, M) = \mathit{parent}(\mathit{last}(M))$.

□

Theorem 4.4.5. *All transformer strings in the minimum model of a call-site- or object-sensitive transformer string instantiation of the rules in Figure 3.3 are consistent.*

Proof. Follows from Lemmas 4.4.2, 4.4.3, and 4.4.4 by induction on the derivation of transformer strings according to the rules in Figure 3.3. □

4.4.2 Truncated Concretization

This section presents a mapping from truncated transformer strings to sets of truncated explicit strings. Unlike the concretization function defined in Section 4.3, the mapping is dependent on an interpretation of predicate $\mathsf{reach}^{\mathbf{e}}$. This dependence is required to prove the precision property. Let m and h, where $m > 0$ and $0 \leq h \leq m$, determine the levels of method and heap contexts, respectively. Under object sensitivity, m and h are constrained by $h = m - 1$ (see Section 2.1.3).

Definition 4.4.5. Given i and j and a transformer string $A \equiv \overset{\smile}{A_{\mathbf{x}}} \cdot A_{\mathbf{w}} \cdot \widehat{A_{\mathbf{e}}}$ from method P to method Q and an interpretation of predicate $\mathsf{reach}^{\mathbf{e}}$, let the concretization of A into a set of explicit strings, a subset of $\mathbf{CtxtT}^{\mathbf{e}}_{i,j}$, be defined as follows:

$$
\begin{aligned}
&\gamma_{i,j}(\overset{\smile}{A_{\mathbf{x}}} \cdot \widehat{A_{\mathbf{e}}}, P, Q) = \\
&\qquad \{(\mathit{prefix}_i(A_{\mathbf{x}} \cdot M), \mathit{prefix}_j(A_{\mathbf{e}} \cdot M)) \\
&\qquad\qquad \mid \mathsf{reach}^{\mathbf{e}}(\mathit{base}(A, P, Q), M)\}. \\
&\gamma_{i,j}(\overset{\smile}{A_{\mathbf{x}}} \cdot * \cdot \widehat{A_{\mathbf{e}}}, P, Q) = \\
&\qquad \{(\mathit{prefix}_i(A_{\mathbf{x}} \cdot M), \mathit{prefix}_j(A_{\mathbf{e}} \cdot N)) \\
&\qquad\qquad \mid\ \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(P, A_{\mathbf{x}}), M) \\
&\qquad\qquad \wedge \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, A_{\mathbf{e}}), N)\}.
\end{aligned}
$$

The concretization of partial method contexts is defined as follows:

Definition 4.4.6. Given a partial method context $M \in \mathbf{Ctxts^t}$ of a method P and an interpretation of predicate $\mathsf{reach^e}$, let the concretization of M into a set of method contexts, a subset of $\mathbf{Ctxts^e}$, be defined as follows:

$$\gamma^{\mathbf{M}}(P, M) \equiv \{\mathit{prefix}_m(M \cdot M') \mid \mathsf{reach^e}(\mathit{rem}(P, M), M')\}.$$

Definition 4.4.7. Given an interpretation $\mathcal{I}$ of derived relations of a call-site- or object-sensitive transformer string instantiation using h and m levels of heap and method context, of the rules in Figure 3.3, let the *explicit string concretization of the interpretation*, denoted $\gamma^{\mathbf{I}}(\mathcal{I})$, be a set of facts defined as follows:

$$\begin{aligned}\gamma^{\mathbf{I}}(\mathcal{I}) \equiv\ & \{\mathsf{pts^e}(Y, H, A') \mid \mathsf{pts^t}(Y, H, A) \wedge A' \in \gamma_{h,m}(A, \mathit{parent}(H), \mathit{parent}(Y))\} \\ & \cup \{\mathsf{hpts^e}(G, F, H, A') \mid \mathsf{hpts^t}(G, F, H, A) \wedge A' \in \gamma_{h,h}(A, \mathit{parent}(H), \mathit{parent}(G))\} \\ & \cup \{\mathsf{call^e}(I, Q, A') \mid \mathsf{call^t}(I, Q, A) \wedge A' \in \gamma_{m,m}(A, \mathit{parent}(I), Q)\} \\ & \cup \{\mathsf{reach^e}(P, A') \mid \mathsf{reach^t}(P, A) \wedge A' \in \gamma^{\mathbf{M}}(P, A)\}.\end{aligned}$$

4.4.3 Derivability

Definition 4.4.8. M is a *reachable method context* of a method P with respect to an interpretation of $\mathsf{reach^e}$ iff $\mathsf{reach^e}(P, M)$.

Transformer strings have a *derivability* property with respect to interpretations of $\mathsf{reach^e}$. Informally, the derivable property creates a constraint between an interpretation of the $\mathsf{reach^e}$ predicate and derivable transformer strings: for example, if a transformer string $A \equiv \breve{x} \cdot \breve{y} \cdot \breve{z} \cdot \widehat{a} \cdot \widehat{b} \cdot \widehat{c}$ from P to Q is derivable, then there must be a reachable context of P with prefix $x \cdot y \cdot z$ and a reachable context of Q with prefix $c \cdot b \cdot a$. Another property of derivable transformer strings is that if A is derivable and M is a reachable method context of P, then for all $A' \in \gamma^{\mathbf{c}}(A)$, $A'(M)$ is a reachable method context of Q.

Definition 4.4.9. A transformer string $A \equiv \overset{\smile}{A_{\mathbf{x}}} \cdot A_{\mathbf{w}} \cdot \widehat{A_{\mathbf{e}}}$ is *derivable* with respect to an interpretation of predicate $\mathsf{reach^e}$ if

1. there exists a reachable context of $\mathit{rem}(P, A_{\mathbf{x}})$,
2. for all reachable contexts M of $\mathit{rem}(P, A_{\mathbf{x}})$, for all $0 \leq l \leq \|A_{\mathbf{x}}\|$, there exists a reachable context M' of $\mathit{rem}(P, \mathit{prefix}_l(A_{\mathbf{x}}))$ such that $M' = \mathit{prefix}_m(\mathit{drop}_l(A_{\mathbf{x}}) \cdot M)$,
3. there exists a reachable context of $\mathit{rem}(Q, A_{\mathbf{e}})$, and
4. for all reachable contexts N of $\mathit{rem}(Q, A_{\mathbf{e}})$, for all $0 \leq l \leq \|A_{\mathbf{e}}\|$, there exists a reachable context N' of $\mathit{rem}(Q, \mathit{prefix}_l(A_{\mathbf{e}}))$ such that $N' = \mathit{prefix}_m(\mathit{drop}_l(A_{\mathbf{e}}) \cdot N)$.

Example. If $\breve{a} \cdot \widehat{b}$ is a derivable transformer string from P to R, then there must exist a reachable context of $Q \equiv \mathit{base}(\breve{a} \cdot \widehat{b}, P, R)$, and for all reachable contexts M of Q, $\mathit{prefix}_m(a \cdot M)$ is a reachable method context of P and $\mathit{prefix}_m(b \cdot M)$ is a reachable method context of R.

Definition 4.4.10. A partial method context M of method P is *derivable* with respect to an interpretation of predicate $\mathsf{reach}^{\mathsf{e}}$ if the transformer string $\widehat{M}$ from $\mathit{rem}(P, M)$ to P is derivable with respect to the interpretation.

The next five lemmas state that operations on transformer strings preserve the derivability property.

Lemma 4.4.6.

1. *$\widehat{\texttt{entry}}$ from $\texttt{entryM}$ to $\texttt{main}$ is derivable with respect to an interpretation $\mathcal{I} \equiv \{\mathsf{reach}^{\mathsf{e}}(\texttt{main}, [\texttt{entry}]), \mathsf{reach}^{\mathsf{e}}(\texttt{entryM}, [\,])\}$.*
2. *For all transformer strings $A \in \mathbf{CtxtT}^{\mathbf{t}}_{i,j}$ from P to Q that are derivable with respect to $\mathcal{I}$, $\mathit{inv}^{\mathsf{t}}(A)$ from Q to P is derivable with respect to $\mathcal{I}$.*
3. *For all reachable methods P, Q, and R, for all transformer strings $A \in \mathbf{CtxtT}^{\mathbf{t}}_{h,m}$ from P to Q that are derivable with respect to $\mathcal{I}$, and for all $B \in \mathbf{CtxtT}^{\mathbf{t}}_{h,m}$ such that $\mathsf{any}^{\mathsf{t}}(R, A, B)$, B from P to R is derivable with respect to $\mathcal{I}$.*

4. *For all transformer strings $A \in \mathbf{CtxtT}^{\mathbf{t}}_{i,j}$ from P to Q that are derivable with respect to $\mathcal{I}$, target$^{\mathbf{t}}(A)$ is derivable with respect to $\mathcal{I}$.*

Proof.

1. Follows trivially from definitions.
2. Follows from symmetry of Definition 4.4.9.
3. Let $A \equiv \overset{\smile}{A_{\mathbf{x}}} \cdot A_{\mathbf{w}} \cdot \overset{\frown}{A_{\mathbf{e}}}$. Then $B = \overset{\smile}{A_{\mathbf{x}}} \cdot *$. Definitions 4.4.9.3 and 4.4.9.4 follow trivially from the fact that R is reachable. Definitions 4.4.9.1 and 4.4.9.2 follow from the fact that A is derivable.
4. Follows trivially from definitions.

$\square$

Lemma 4.4.7. *For all derivable transformer strings $A \equiv \overset{\smile}{A_{\mathbf{x}}} \cdot A_{\mathbf{w}} \cdot \overset{\frown}{A_{\mathbf{e}}}$ from P to Q and for all $i, j \geq 0$, $trunc_{i,j}(A)$ is derivable .*

Proof. If truncation does not occur, then the lemma trivially follows. Otherwise, let $C_{\mathbf{x}} \equiv prefix_i(A_{\mathbf{x}})$ and $C_{\mathbf{e}} \equiv prefix_j(A_{\mathbf{e}})$. Then, $trunc_{i,j}(A) = \overset{\smile}{C_{\mathbf{x}}} \cdot * \cdot \overset{\frown}{C_{\mathbf{e}}}$. Proof that $trunc_{i,j}(A)$ satisfies Definitions 4.4.9.1 and 4.4.9.2 is presented; Definitions 4.4.9.3 and 4.4.9.4 follow by symmetry.

Because A is derivable, there exists a reachable context N of $rem(P, A_{\mathbf{x}})$. Then, $\forall l : 0 \leq l \leq A_{\mathbf{x}}, \exists N' : \mathsf{reach}^{\mathbf{e}}(rem(P, prefix_l(A_{\mathbf{x}})), N') \wedge N' = prefix_m(drop_l(A_{\mathbf{x}}) \cdot N)$. Since $C_{\mathbf{x}} \cdot drop_i(A_{\mathbf{x}}) = A_{\mathbf{x}}$, we get $\forall l : 0 \leq l \leq \|C_{\mathbf{x}}\|, \exists N' : \mathsf{reach}^{\mathbf{e}}(rem(P, prefix_l(C_{\mathbf{x}})), N') \wedge N' = prefix_m(drop_l(C_{\mathbf{x}}) \cdot drop_i(A_{\mathbf{x}}) \cdot N)$. Thus, there exists a reachable context N' of $rem(P, C_{\mathbf{x}})$ such that $N' = prefix_m(drop_i(A_{\mathbf{x}}) \cdot N)$. Furthermore, $\forall l : 0 \leq l \leq \|C_{\mathbf{x}}\|, \exists N'' : \mathsf{reach}^{\mathbf{e}}(rem(P, prefix_l(C_{\mathbf{x}})), N'') \wedge N'' = prefix_m(drop_l(C_{\mathbf{x}}) \cdot N')$. $\square$

Lemma 4.4.8. *Let* $A \equiv \overset{\smile}{A_{\mathbf{x}}} \cdot A_{\mathbf{w}} \cdot \widehat{A_{\mathbf{e}}} \in \mathbf{CtxtT}^{\mathbf{t}}_{i,j}$ *from* P *to* Q *and* $B \equiv \overset{\smile}{B_{\mathbf{x}}} \cdot B_{\mathbf{w}} \cdot \widehat{B_{\mathbf{e}}} \in \mathbf{CtxtT}^{\mathbf{t}}_{j,k}$ *from* Q *to* R *be consistent transformer strings derivable with respect to* $\mathcal{I}$*, where* $i, j, k \leq m$*. If* $match(A \cdot B) \neq \bot$*, then* $match(A \cdot B)$ *is derivable with respect to* $\mathcal{I}$*.*

Proof.

1. Suppose $\|A_{\mathbf{e}}\| \leq \|B_{\mathbf{x}}\|$.
2. $A_{\mathbf{e}} \cong B_{\mathbf{x}}$.
3. Let $B'_{\mathbf{x}} \equiv drop_{\|A_{\mathbf{e}}\|}(B_{\mathbf{x}})$. [Lemma 4.2.1]
4. $B_{\mathbf{x}} = A_{\mathbf{e}} \cdot B'_{\mathbf{x}}$.
5. *Case*: $A_{\mathbf{w}} = \epsilon$.
   (a) $match(A \cdot B) = \overset{\smile}{A_{\mathbf{x}}} \cdot \overset{\smile}{B'_{\mathbf{x}}} \cdot \widehat{B_{\mathbf{e}}}$.
   (b) $rem(P, A_{\mathbf{x}}) = rem(Q, A_{\mathbf{e}})$. [$A$ is consistent]
   (c) Inst. N: $\mathsf{reach}^{\mathbf{e}}(rem(Q, B_{\mathbf{x}}), N)$. [Def. 4.4.9.1]
   (d) $B'_{\mathbf{x}} = \epsilon \rightarrow rem(P, A_{\mathbf{x}} \cdot B'_{\mathbf{x}}) = rem(Q, A_{\mathbf{e}}) = rem(Q, A_{\mathbf{e}} \cdot B'_{\mathbf{x}})$. [5b]
   (e) $B'_{\mathbf{x}} \neq \epsilon \rightarrow rem(P, A_{\mathbf{x}} \cdot B'_{\mathbf{x}}) = last(B'_{\mathbf{x}}) = rem(Q, A_{\mathbf{e}} \cdot B'_{\mathbf{x}})$.
   (f) $rem(P, A_{\mathbf{x}} \cdot B'_{\mathbf{x}}) = rem(Q, B_{\mathbf{x}})$.
   (g) $match(A \cdot B)$ satisfies Def. 4.4.9.1.

   (h) $\forall l : 0 \leq l \leq \|B_{\mathbf{x}}\|,$
   $\exists N' : \mathsf{reach}^{\mathbf{e}}(rem(Q, prefix_l(B_{\mathbf{x}})), N')$
   $\wedge\ N' = prefix_m(drop_l(B_{\mathbf{x}}) \cdot N)$. [Def. 4.4.9.2]
   (i) Inst. O: $\mathsf{reach}^{\mathbf{e}}(rem(R, B_{\mathbf{e}}), O)$. [Def. 4.4.9.3]
   (j) $match(A \cdot B)$ satisfies Def. 4.4.9.3.

(k) $\forall l : 0 \le l \le \|B_{\mathbf{e}}\|$,
$\exists N' : \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(R, \mathit{prefix}_l(B_{\mathbf{e}})), N')$
$\wedge\ N' = \mathit{prefix}_m(\mathit{drop}_l(B_{\mathbf{e}}) \cdot O)$. [Def. 4.4.9.4]

(l) $\mathit{match}(A \cdot B)$ satisfies Def. 4.4.9.4.

(m) Inst. M: $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, \mathit{prefix}_{\|A_{\mathbf{e}}\|}(B_{\mathbf{x}})), M)$
$\wedge\ M = \mathit{prefix}_m(\mathit{drop}_{\|A_{\mathbf{e}}\|}(B_{\mathbf{x}}) \cdot N)$. [5h]

(n) $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, A_{\mathbf{e}}), M) \wedge M = \mathit{prefix}_m(B'_{\mathbf{x}} \cdot N)$. [Substitute 4]

(o) $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(P, A_{\mathbf{x}}), M)$. [Subst. 5b]

(p) $\forall l : 0 \le l \le \|A_{\mathbf{x}}\|$,
$\exists M' : \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(P, \mathit{prefix}_l(A_{\mathbf{x}})), M')$
$\wedge\ M' = \mathit{prefix}_m(\mathit{drop}_l(A_{\mathbf{x}}) \cdot M)$. [Def. 4.4.9.2]

(q) $\forall l : 0 \le l \le \|B_{\mathbf{x}}\| - \|A_{\mathbf{e}}\|$,
$A_{\mathbf{e}} \cdot \mathit{prefix}_l(B'_{\mathbf{x}}) = \mathit{prefix}_{l+\|A_{\mathbf{e}}\|}(B_{\mathbf{x}})$
$\wedge\ \mathit{drop}_l(B'_{\mathbf{x}}) = \mathit{drop}_{l+\|A_{\mathbf{e}}\|}(B_{\mathbf{x}})$. [Manipulation of 4]

(r) $\forall l : 0 \le l \le \|B_{\mathbf{x}}\| - \|A_{\mathbf{e}}\|$,
$\exists N' : \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, A_{\mathbf{e}} \cdot \mathit{prefix}_l(B'_{\mathbf{x}})), N')$
$\wedge\ N' = \mathit{prefix}_m(\mathit{drop}_l(B'_{\mathbf{x}}) \cdot N)$. [Subst. 5q into 5h]

(s) $\forall l : 0 \le l \le \|B_{\mathbf{x}}\| - \|A_{\mathbf{e}}\|$,
$\exists N' : \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(P, A_{\mathbf{x}} \cdot \mathit{prefix}_l(B'_{\mathbf{x}})), N')$
$\wedge\ N' = \mathit{prefix}_m(\mathit{drop}_l(B'_{\mathbf{x}}) \cdot N)$. [Subst. 5b]

(t) $\forall l : 0 \le l \le \|B_{\mathbf{x}}\| - \|A_{\mathbf{e}}\|$,
$A_{\mathbf{x}} \cdot \mathit{prefix}_l(B'_{\mathbf{x}}) = \mathit{prefix}_{l+\|A_{\mathbf{x}}\|}(A_{\mathbf{x}} \cdot B'_{\mathbf{x}})$
$\wedge\ \mathit{drop}_l(B'_{\mathbf{x}}) = \mathit{drop}_{l+\|A_{\mathbf{x}}\|}(A_{\mathbf{x}} \cdot B'_{\mathbf{x}})$. [Manip. of *prefix* and *drop*]

(u) $\forall l : \|A_{\mathbf{x}}\| \le l \le \|A_{\mathbf{x}}\| + \|B_{\mathbf{x}}\| - \|A_{\mathbf{e}}\|$,
$\exists N' : \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(P, \mathit{prefix}_l(A_{\mathbf{x}} \cdot B'_{\mathbf{x}})), N')$
$\wedge\ N' = \mathit{prefix}_m(\mathit{drop}_l(A_{\mathbf{x}} \cdot B'_{\mathbf{x}}) \cdot N)$. [Adjust l; Subst. 5t into 5s]

(v) $\forall l : 0 \leq l \leq \|A_{\mathbf{x}} \cdot B'_{\mathbf{x}}\|$,

$\exists N' : \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(P, \mathit{prefix}_l(A_{\mathbf{x}} \cdot B'_{\mathbf{x}})), N')$

$\wedge\ N' = \mathit{prefix}_m(\mathit{drop}_l(A_{\mathbf{x}} \cdot B'_{\mathbf{x}}) \cdot N)$. [5p and 5u]

(w) $\mathit{match}(A \cdot B)$ satisfies Def. 4.4.9.2.

Case: $A_{\mathbf{w}} = *$.

(a) $\mathit{match}(A \cdot B) = \smile{A_{\mathbf{x}}} \cdot * \cdot \widehat{B_{\mathbf{e}}}$.

(b) $\exists M : \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(P, A_{\mathbf{x}}), M)$. [Def. 4.4.9.1].

(c) $\exists N : \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(R, B_{\mathbf{e}}), N)$. [Def. 4.4.9.3].

(d) $\forall l : 0 \leq l \leq \|A_{\mathbf{x}}\|, \exists M' :$

$\mathsf{reach}^{\mathbf{e}}(P, \mathit{prefix}_l(A_{\mathbf{x}})) \wedge M' = \mathit{prefix}_m(\mathit{drop}_l(A_{\mathbf{x}}) \cdot M)$. [Def. 4.4.9.2].

(e) $\forall l : 0 \leq l \leq \|B_{\mathbf{e}}\|, \exists N' :$

$\mathsf{reach}^{\mathbf{e}}(R, \mathit{prefix}_l(B_{\mathbf{e}})) \wedge N' = \mathit{prefix}_m(\mathit{drop}_l(B_{\mathbf{e}}) \cdot N)$. [Def. 4.4.9.4].

(f) $\mathit{match}(A \cdot B)$ satisfies Def. 4.4.9.

6. The case of $\|A_{\mathbf{e}}\| \geq \|B_{\mathbf{x}}\|$ follows the same reasoning as above.

□

Lemma 4.4.9. *For all partial method contexts M of P derivable under an interpretation $\mathcal{I}$, $\mathit{record}^{\mathbf{t}}(M)$ from P to P is derivable with respect to $\mathcal{I}$.*

Proof. Follows trivially from definitions. □

The Static and Virt rules in Figure 3.3 derive a new transformer string for a $\mathsf{call}^{\mathbf{t}}$ fact. The $\mathsf{call}^{\mathbf{t}}$ fact in turn derives a $\mathsf{reach}^{\mathbf{t}}$ fact. The transformer string for the $\mathsf{call}^{\mathbf{t}}$ fact is shown to be derivable with respect to an interpretation of $\mathsf{reach}^{\mathbf{e}}$, which includes the concretization of the $\mathsf{reach}^{\mathbf{t}}$ fact:

Lemma 4.4.10. *For all consistent transformer strings* $B \equiv \breve{B_{\mathbf{x}}} \cdot B_{\mathbf{w}} \cdot \widehat{B_{\mathbf{e}}} \in \mathbf{CtxtT}^{\mathbf{t}}_{h,m}$ *from* R *to* P *that are derivable with respect to an interpretation* $\mathcal{I}$*, heap allocation sites* H *in* R*, and invocations* I *in* P*, let* $\mathcal{I}'$ *be an interpretation defined as follows:*

$$\mathcal{I}' \equiv \mathcal{I} \cup \{\mathsf{reach}^{\mathbf{e}}(Q, O) \mid O \in \gamma^{\mathbf{M}}(Q, \mathit{target}(\mathit{merge}^{\mathbf{t}\Box}(H, I, B)))\}.$$

Then, $\mathit{merge}^{\mathbf{t}\Box}(H, I, B)$ *from* P *to* Q *is derivable with respect to* $\mathcal{I}'$*.*

Proof.

Call-site sensitivity. We must show that $\mathit{trunc}_{m,m}(\breve{B_{\mathbf{e}}} \cdot \widehat{B_{\mathbf{e}}} \cdot \widehat{I})$ is derivable:

1. Let $C \equiv \breve{C_{\mathbf{x}}} \cdot \widehat{C_{\mathbf{e}}} \equiv \breve{B_{\mathbf{e}}} \cdot \widehat{B_{\mathbf{e}}} \cdot \widehat{I}$.
2. Inst. M: $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(R, B_{\mathbf{e}}), M)$. [Def. 4.4.9.1]
3. $\forall l : 0 \leq l \leq \|B_{\mathbf{e}}\|, \exists N' : \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(R, \mathit{prefix}_l(B_{\mathbf{e}})), N')$
   $\wedge\ N' = \mathit{prefix}_m(\mathit{drop}_l(B_{\mathbf{e}}) \cdot M)$. [Def. 4.4.9.2]
4. Inst. N: $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(P, B_{\mathbf{e}}), N)$. [Def. 4.4.9.3]
5. C satisfies Def. 4.4.9.1.
6. $\forall l : 0 \leq l \leq \|B_{\mathbf{e}}\|, \exists N' : \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(P, \mathit{prefix}_l(B_{\mathbf{e}})), N')$
   $\wedge\ N' = \mathit{prefix}_m(\mathit{drop}_l(B_{\mathbf{e}}) \cdot N)$. [Def. 4.4.9.4]
7. C satisfies Def. 4.4.9.2.
8. $\mathit{target}(C) = C_{\mathbf{e}}$.
9. $\mathit{rem}(Q, C_{\mathbf{e}}) = \mathit{rem}(P, B_{\mathbf{e}})$.
10. $\mathit{prefix}_m(C_{\mathbf{e}} \cdot N) \in \gamma^{\mathbf{M}}(Q, C_{\mathbf{e}})$. [Def. 4.4.6, 4, and 9]
11. $\exists O : \mathsf{reach}^{\mathbf{e}}(Q, O) \wedge O = \mathit{prefix}_m(C_{\mathbf{e}} \cdot N)$. [Construction of $\mathcal{I}'$]

12. C satisfies Def. 4.4.9.3.

13. $\mathit{rem}(Q, \mathit{prefix}_0(C_{\mathbf{e}})) = Q \wedge \mathit{drop}_0(C_{\mathbf{e}}) = C_{\mathbf{e}}$.

14. $\exists N' : \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, \mathit{prefix}_0(C_{\mathbf{e}})), N')$
    $\wedge\ N' = \mathit{prefix}_m(\mathit{drop}_0(C_{\mathbf{e}}) \cdot N)$.

15. Let $1 \leq l' \leq \|C_{\mathbf{e}}\|$.

16. $\exists N' : \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(P, \mathit{prefix}_{l'-1}(B_{\mathbf{e}})), N')$
    $\wedge\ N' = \mathit{prefix}_m(\mathit{drop}_{l'-1}(B_{\mathbf{e}}) \cdot N)$. [6]

17. $\mathit{rem}(Q, I) = P$. [Premise]

18. $\mathit{rem}(Q, \mathit{prefix}_{l'}(C_{\mathbf{e}})) = \mathit{rem}(P, \mathit{prefix}_{l'-1}(B_{\mathbf{e}}))$. [1 and 17]

19. $\mathit{drop}_{l'}(C_{\mathbf{e}}) = \mathit{drop}_{l'-1}(B_{\mathbf{e}})$. [1]

20. $\exists N' : \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, \mathit{prefix}_{l}(C_{\mathbf{e}})), N')$.
    $\wedge\ N' = \mathit{prefix}_m(\mathit{drop}_{l'}(C_{\mathbf{e}}) \cdot N)$. [Subst. into 16]

21. $\forall l : 0 \leq l \leq \|C_{\mathbf{e}}\|, \exists N' : \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, \mathit{prefix}_{l}(C_{\mathbf{e}})), N')$
    $\wedge\ N' = \mathit{prefix}_m(\mathit{drop}_{l}(C_{\mathbf{e}}) \cdot N)$. [14 and 20]

22. C satisfies Def. 4.4.9.4.

23. $\mathit{trunc}_{m,m}(C)$ is derivable. [Lemma 4.4.8]

Object sensitivity. We must show that $\breve{B_{\mathbf{e}}} \cdot \wideparen{B_{\mathbf{x}}} \cdot \widehat{H}$ is derivable:

1. Let $C \equiv \breve{C_{\mathbf{x}}} \cdot \wideparen{C_{\mathbf{e}}} \equiv \breve{B_{\mathbf{e}}} \cdot \wideparen{B_{\mathbf{x}}} \cdot \widehat{H}$.

2. Inst. N: $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(R, B_{\mathbf{x}}), N)$. [Def. 4.4.9.1]

3. $\forall l : 0 \leq l \leq \|B_{\mathbf{x}}\|, \exists N' : \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(R, \mathit{prefix}_l(B_{\mathbf{x}})))$
   $\wedge\ N' = \mathit{prefix}_m(\mathit{drop}_l(B_{\mathbf{x}}) \cdot N)$. [Def. 4.4.9.2]

4. $\exists N' : \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(P, B_{\mathbf{e}}))$. [Def. 4.4.9.4]

5. C satisfies Def. 4.4.9.1.

6. $\forall l : 0 \leq l \leq \|B_{\mathbf{e}}\|, \exists N' : \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(P, \mathit{prefix}_l(B_{\mathbf{e}})))$
   $\wedge\ N' = \mathit{prefix}_m(\mathit{drop}_l(B_{\mathbf{e}}) \cdot N)$. [Def. 4.4.9.4]

7. C satisfies Def. 4.4.9.2.

8. $\mathit{target}(C) = C_{\mathbf{e}}$.

9. $\mathit{prefix}_m(C_{\mathbf{e}} \cdot N) \in \gamma^{\mathbf{M}}(Q, \mathit{target}(C))$.

10. $\exists O : \mathsf{reach}^{\mathbf{e}}(Q, O) \wedge O = \mathit{prefix}_m(C_{\mathbf{e}} \cdot N)$. [Construction of $\mathcal{I}'$]

11. C satisfies Def. 4.4.9.3.

12. $\mathit{rem}(Q, \mathit{prefix}_0(C_{\mathbf{e}})) = Q \wedge \mathit{drop}_0(C_{\mathbf{e}}) = C_{\mathbf{e}}$.

13. $\exists O : \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, \mathit{prefix}_0(C_{\mathbf{e}})), O) \wedge O = \mathit{prefix}_m(\mathit{drop}_0(C_{\mathbf{e}}) \cdot N)$.

14. Let $1 \leq l' \leq \|C_{\mathbf{e}}\|$.

15. $\mathit{rem}(Q, H) = R$. [Premise]

16. $\mathit{rem}(Q, \mathit{prefix}_l(C_{\mathbf{e}})) = \mathit{rem}(R, \mathit{prefix}_{l'-1}(B_{\mathbf{x}}))$. [1 and 15]

17. $\mathit{drop}_{l'}(C_{\mathbf{e}}) = \mathit{drop}_{l'-1}(B_{\mathbf{x}})$.

18. $\exists N' : \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(R, \mathit{prefix}_{l'-1}(B_{\mathbf{x}})))$
    $\wedge\ N' = \mathit{prefix}_m(\mathit{drop}_{l'-1}(B_{\mathbf{x}}) \cdot N)$. [3]

19. $\exists N' : \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, \mathit{prefix}_{l'}(C_{\mathbf{e}})), N')$
 $\wedge\ N' = \mathit{prefix}_m(\mathit{drop}_{l'}(C_{\mathbf{e}}) \cdot N)$.

20. $\forall l : 0 \le l \le \|C_{\mathbf{e}}\|,$
 $\exists N' : \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, \mathit{prefix}_{l}(C_{\mathbf{e}})), N')$
 $\wedge\ N' = \mathit{prefix}_m(\mathit{drop}_{l}(C_{\mathbf{e}}) \cdot N)$. [13 and 19]

21. C satisfies Def. 4.4.9.4.

22. C is derivable.

$\square$

Lemma 4.4.11. *For all partial method contexts M of P derivable with respect to an interpretation $\mathcal{I}$ and invocations I in P, let $\mathcal{I}'$ be an interpretation defined as follows:*

$$\mathcal{I}' \equiv \mathcal{I} \cup \{\mathsf{reach}^{\mathbf{e}}(Q, O) : O \in \gamma^{\mathbf{M}}(Q, \mathit{target}(\mathit{merge\_s}^{\mathbf{t}\square}(I, M)))\}$$

Then, $\mathit{merge\_s}^{\mathbf{t}\square}(I, M)$ from P to Q is derivable with respect to $\mathcal{I}'$.

Proof.

1. **Call-site sensitivity**. We must show that $\widehat{I}$ is derivable:

   Let $M' \in \gamma^{\mathbf{M}}(M)$ be a reachable context of P. Then $\mathsf{reach}^{\mathbf{e}}(Q, \mathit{prefix}_m(I \cdot M'))$ from the construction of $\mathcal{I}'$. Thus, $\widehat{I}$ is derivable.

2. **Object sensitivity**. We must show that $\widecheck{M} \cdot \widehat{M}$ is derivable:

   By Lemma 4.4.1, M is not empty. Let $R = \mathit{base}(M, P, Q) = \mathit{parent}(\mathit{last}(M))$. M is derivable, thus there exists a reachable context N of R. Then, $\forall l : 0 \le l \le \|M\|, \exists N' : \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(P, \mathit{prefix}_l(M)), N') \wedge N' = \mathit{prefix}_m(\mathit{drop}_l(M) \cdot N)$, because M is derivable. We have $\forall l : 1 \le l \le \|M\|, \mathit{rem}(Q, \mathit{prefix}_l(M)) = \mathit{rem}(P, \mathit{prefix}_l(M))$. From the construction of $\mathcal{I}'$, we have $\mathsf{reach}^{\mathbf{e}}(Q, \mathit{prefix}_m(M \cdot N))$. Thus, $\forall l : 0 \le l \le \|M\|, \exists N' : \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, \mathit{prefix}_l(M)), N') \wedge N' = \mathit{prefix}_m(\mathit{drop}_l(M) \cdot N)$. Thus $\widecheck{M} \cdot \widehat{M}$ is derivable.

□

Theorem 4.4.12. *All transformer strings in the minimum model of a call-site- or object-sensitive transformer string instantiation of the rules in Figure 3.3 are derivable with respect to the minimum model.*

Proof. Follows from Lemmas 4.4.6, 4.4.7, 4.4.8, 4.4.9, 4.4.10, and 4.4.11 by induction on the derivation of transformer strings according to the rules in Figure 3.3. □

4.4.4 Subset Theorem

Let transformer strings be derivable with respect to the minimum model of a call-site- or object-sensitive transformer string instantiation of the rules in Figure 3.3.

We define convenience functions, similarly to Section 4.3.2, for concretizing the last arguments of these predicates:

$$\gamma_{i,k}^{\mathsf{comp}}(A, B, P, R) \equiv \bigcup\{\gamma_{i,k}(C, P, R) \mid \mathsf{comp}^{\mathsf{t}}(A, B, C)\}.$$
$$\gamma_{h,m}^{\mathsf{any}}(A, P, R) \equiv \bigcup\{\gamma_{h,m}(B, P, R) \mid \mathsf{any}^{\mathsf{t}}(R, A, B)\}.$$

The following lemma states that concretization to explicit strings of a particular pair of lengths is invariant with respect to truncation of transformer strings to the same pair of lengths. The lemma is used in later lemmas for properties of operations that use the truncation function:

Lemma 4.4.13. *For all consistent transformer strings $A \equiv \overset{\smile}{A_{\mathbf{x}}} \cdot A_{\mathbf{w}} \cdot \widehat{A_{\mathbf{e}}}$ from P to Q and $i, j \geq 0$, the following equality holds:*

$$\gamma_{i,j}(A, P, Q) = \gamma_{i,j}(trunc_{i,j}(A), P, Q).$$

Proof. If $\|A_{\mathbf{x}}\| < i \wedge \|A_{\mathbf{e}}\| < j$ then the lemma trivially follows. If $\|A_{\mathbf{x}}\| \geq i \wedge \|A_{\mathbf{e}}\| \geq j$, then $\gamma_{i,j}(A, P, Q) = \{(prefix_i(A_{\mathbf{x}}), prefix_j(A_{\mathbf{e}}))\} = \gamma_{i,j}(trunc_{i,j}(A), P, Q)$ and the lemma follows.

Suppose $\|A_{\mathbf{x}}\| \geq i \wedge \|A_{\mathbf{e}}\| < j$. Then,

$$\begin{aligned}\gamma_{i,j}(A, P, Q) &= \{(\mathit{prefix}_i(A_{\mathbf{x}} \cdot N), \mathit{prefix}_j(A_{\mathbf{e}} \cdot N)) \mid \mathsf{reach}^{\mathbf{e}}(\mathit{base}(A, P, Q), N)\} \\ &= \{(\mathit{prefix}_i(A_{\mathbf{x}}), \mathit{prefix}_j(A_{\mathbf{e}} \cdot N)) \mid \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, A_{\mathbf{e}}), N)\}.\end{aligned}$$

We have $\mathit{trunc}_{i,j}(A) = \overbrace{\mathit{prefix}_i(A_{\mathbf{x}})} \cdot * \cdot \widehat{A_{\mathbf{e}}}$. Thus,

$$\gamma_{i,j}(\mathit{trunc}_{i,j}(A), P, Q) = \{(\mathit{prefix}_i(A_{\mathbf{x}}), \mathit{prefix}_j(A_{\mathbf{e}} \cdot N)) \mid \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, A_{\mathbf{e}}), N)\}.$$

The case of $\|A_{\mathbf{x}}\| < i \wedge \|A_{\mathbf{e}}\| \geq j$ follows the same reasoning as above. □

The next five lemmas state that operations on explicit strings and transformer strings preserve the subset inequality.

Lemma 4.4.14. *For all consistent and derivable transformer strings $A \equiv \overbrace{A_{\mathbf{x}}} \cdot A_{\mathbf{w}} \cdot \widehat{A_{\mathbf{e}}} \in$ $\mathbf{CtxtT}^{\mathbf{t}}_{i,j}$ from P to Q, and reachable methods R, the following inequalities hold:*

1. $\gamma^{\mathbf{M}}(\mathtt{main}, \widehat{\mathtt{entry}}) \subseteq \{\mathtt{entry}\}$.
2. $\gamma_{j,i}(\mathit{inv}^{\mathbf{t}}(A), Q, P) \subseteq \{\mathit{inv}^{\mathbf{e}}(A') \mid A' \in \gamma_{i,j}(A, P, Q)\}$.
3. $\gamma^{\mathsf{any}}_{h,m}(A, P, R) \subseteq \{B \mid A' \in \gamma_{h,m}(A, P, Q) \wedge \mathsf{any}^{\mathbf{e}}(R, A', B)\}$.
4. $\gamma^{\mathbf{M}}(Q, \mathit{target}^{\mathbf{t}}(A)) \subseteq \{\mathit{target}^{\mathbf{e}}(A') \mid A' \in \gamma_{h,m}(A, P, Q)\}$.

Proof.

1. Follows trivially from Definition 4.4.1 and 4.4.9.
2. Follows trivially from symmetry of Definition 4.4.9.

3. We must show that

$$\gamma_{h,m}(\overset{\smile}{A_{\mathbf{x}}}\cdot *, P, R) \subseteq \{(X, M) \mid (X, Y) \in \gamma_{h,m}(A, P, Q) \wedge \mathsf{reach}^{\mathbf{e}}(R, M)\}.$$

   Let $(X', Y') \in \gamma_{h,m}(\overset{\smile}{A_{\mathbf{x}}}\cdot *, P, R)$. By Definition 4.4.5, there exists N such that $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(P, A_{\mathbf{x}}), N)$ and $X' = \mathit{prefix}_h(A_{\mathbf{x}}\cdot N)$. Furthermore, $\mathsf{reach}^{\mathbf{e}}(R, Y')$. Thus, $(X, Y'') \in \gamma_{h,m}(A, P, Q)$ for some Y''. Thus, $(X', Y') \in \{(X, M) \mid (X, Y) \in \gamma_{h,m}(A, P, Q), \mathsf{reach}^{\mathbf{e}}(R, M)\}$.

4. We must show that $\gamma^{\mathbf{M}}(Q, A_{\mathbf{e}}) \subseteq \{Y \mid (X, Y) \in \gamma_{h,m}(A, P, Q)\}$:

   Let $Y' \in \gamma^{\mathbf{M}}(Q, A_{\mathbf{e}})$. By Definition 4.4.5, there exists N such that $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, A_{\mathbf{e}}), N)$ and $Y' = \mathit{prefix}_m(A_{\mathbf{e}}\cdot N)$. Thus, $(X', Y') \in \gamma^{\mathbf{M}}(A, P, Q)$ for some X'. Thus, $Y' \in \{Y \mid (X, Y) \in \gamma_{h,m}(A, P, Q)\}$.

□

Lemma 4.4.15. *Let $A \equiv \overset{\smile}{A_{\mathbf{x}}}\cdot A_{\mathbf{w}}\cdot \widehat{A_{\mathbf{e}}} \in \mathbf{CtxtT}^{\mathbf{t}}_{i,j}$ from P to Q and $B \equiv \overset{\smile}{B_{\mathbf{x}}}\cdot B_{\mathbf{w}}\cdot \widehat{B_{\mathbf{e}}} \in \mathbf{CtxtT}^{\mathbf{t}}_{j,k}$ from Q to R be consistent and derivable transformer strings, where $i, j, k \leq m$. Then,*

$$\gamma^{\mathsf{comp}}_{i,k}(A, B, P, R) \subseteq \{C' \mid A' \in \gamma_{i,j}(A, P, Q) \wedge B' \in \gamma_{j,k}(B, Q, R) \wedge \mathsf{comp}^{\mathbf{e}}(A', B', C')\}.$$

Proof. The right-hand-side of the inequality is the set

$$\alpha \equiv \{(X, Z) \mid (X, Y) \in \gamma_{i,j}(A, P, Q) \wedge (Y, Z) \in \gamma_{j,k}(B, Q, R)\}.$$

If $\mathit{match}(A\cdot B) = \perp$, then $\gamma^{\mathsf{comp}}_{i,k}(A, B, P, R) = \emptyset$ and by Lemma 3.5.1, $A_{\mathbf{e}} \not\cong B_{\mathbf{x}}$. Thus, if $(U, V) \in \gamma_{i,j}(A, P, Q)$ and $(X, Y) \in \gamma_{j,k}(B, Q, R)$, V and X have different prefixes. Thus $\alpha = \emptyset$.

Suppose $\mathit{match}(A\cdot B) \neq \perp$. Then, we wish to show the following inequality:

$$\gamma_{i,k}(\mathit{match}(A\cdot B), P, R) \subseteq \alpha.$$

Then, $\gamma_{i,k}(\mathit{trunc}_{i,k}(\mathit{match}(A\cdot B)), P, R) \subseteq \alpha$ follows from Lemma 4.4.13.

1. Let $(X, Z) \in \gamma_{i,k}(\mathit{match}(A \cdot B), P, R)$.

2. Suppose $\|A_{\mathbf{e}}\| \leq \|B_{\mathbf{x}}\|$.

3. $A_{\mathbf{e}} \cong B_{\mathbf{x}}$.

4. Let $B'_{\mathbf{x}} \equiv \mathit{drop}_{\|A_{\mathbf{e}}\|}(B_{\mathbf{x}})$.

5. $B_{\mathbf{x}} = A_{\mathbf{e}} \cdot B'_{\mathbf{x}}$.

6. *Case*: $A_{\mathbf{w}} = B_{\mathbf{w}} = \epsilon$.

   (a) $\mathit{match}(A \cdot B) = \overset{\smile}{A_{\mathbf{x}}} \cdot \overset{\smile}{B'_{\mathbf{x}}} \cdot \widehat{B_{\mathbf{e}}}$.

   (b) Inst. N: $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(R, B_{\mathbf{e}}), N)$
   $\wedge\ X = \mathit{prefix}_i(A_{\mathbf{x}} \cdot B'_{\mathbf{x}} \cdot N)$
   $\wedge\ Z = \mathit{prefix}_k(B_{\mathbf{e}} \cdot N)$. [Def. 4.4.5 and 1]

   (c) $\mathit{rem}(R, B_{\mathbf{e}}) = \mathit{rem}(Q, B_{\mathbf{x}})$. [$B$ is consistent]

   (d) $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, B_{\mathbf{x}}), N)$.

   (e) Inst. M: $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, \mathit{prefix}_{\|A_{\mathbf{e}}\|}(B_{\mathbf{x}})), M)$
   $\wedge\ M = \mathit{prefix}_m(\mathit{drop}_{\|A_{\mathbf{e}}\|}(B_{\mathbf{x}}) \cdot N)$. [Def. 4.4.9.2]

   (f) $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, A_{\mathbf{e}}), M) \wedge M = \mathit{prefix}_m(B'_{\mathbf{x}} \cdot N)$. [4]

   (g) $X = \mathit{prefix}_i(A_{\mathbf{x}} \cdot M)$. [$i \leq m$, 6b, and 6f]

   (h) Let $Y \equiv \mathit{prefix}_j(A_{\mathbf{e}} \cdot M)$.

   (i) $\mathsf{reach}^{\mathbf{e}}(\mathit{base}(A, P, Q), M)$.

   (j) $(X, Y) \in \gamma_{i,j}(\overset{\smile}{A_{\mathbf{x}}} \cdot \widehat{A_{\mathbf{e}}}, P, Q)$. [6g, 6h, and 6i]

   (k) $Y = \mathit{prefix}_j(A_{\mathbf{e}} \cdot B'_{\mathbf{x}} \cdot N)$. [$j \leq m$, 6f, and 6h]

   (l) $Y = \mathit{prefix}_j(B_{\mathbf{x}} \cdot N)$.

   (m) $\mathsf{reach}^{\mathbf{e}}(\mathit{base}(B, Q, R), N)$.

(n) $(Y, Z) \in \gamma_{j,k}(\breve{B_{\mathbf{x}}} \cdot \widehat{B_{\mathbf{e}}}, Q, R)$. [6b, 6l, and 6m]

(o) $(X, Z) \in \alpha$. [6j and 6n]

Case: $A_{\mathbf{w}} = * \wedge B_{\mathbf{w}} = \epsilon$.

(a) $\mathit{match}(A \cdot B) = \breve{A_{\mathbf{x}}} \cdot * \cdot \widehat{B_{\mathbf{e}}}$.

(b) Inst N: $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(R, B_{\mathbf{e}}), N)$
$\wedge\ Z = \mathit{prefix}_k(B_{\mathbf{e}} \cdot N)$. [Def. 4.4.5 and 1]

(c) Let $Y \equiv \mathit{prefix}_j(B_{\mathbf{x}} \cdot N)$.

(d) $\mathsf{reach}^{\mathbf{e}}(\mathit{base}(B, Q, R), N)$.

(e) $(Y, Z) \in \gamma_{j,k}(\breve{B_{\mathbf{x}}} \cdot \widehat{B_{\mathbf{e}}}, Q, R)$. [6b, 6c, and 6d]

(f) $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, B_{\mathbf{x}}), N)$. [$B$ is consistent]

(g) Inst M: $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, \mathit{prefix}_{\|A_{\mathbf{e}}\|}(B_{\mathbf{x}})), M)$
$\wedge\ M = \mathit{prefix}_m(\mathit{drop}_{\|A_{\mathbf{e}}\|}(B_{\mathbf{x}}) \cdot N)$. [Def. 4.4.9.2]

(h) $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, A_{\mathbf{e}}), M) \wedge M = \mathit{prefix}_m(B'_{\mathbf{x}} \cdot N)$. [5]

(i) $Y = \mathit{prefix}_j(A_{\mathbf{e}} \cdot B'_{\mathbf{x}} \cdot N)$.

(j) $Y = \mathit{prefix}_j(A_{\mathbf{e}} \cdot M)$.

(k) $\exists M' : \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(P, A_{\mathbf{x}}), M') \wedge X = \mathit{prefix}_i(A_{\mathbf{x}} \cdot M')$.

(l) $(X, Y) \in \gamma_{i,j}(\breve{A_{\mathbf{x}}} \cdot * \cdot \widehat{A_{\mathbf{e}}}, P, Q)$. [6h, 6j, and 6k]

(m) $(X, Z) \in \alpha$.

Case: $A_{\mathbf{w}} = \epsilon \wedge B_{\mathbf{w}} = *$.

(a) $\mathit{match}(A \cdot B) = \breve{A_{\mathbf{x}}} \cdot \breve{B'_{\mathbf{x}}} \cdot * \cdot \widehat{B_{\mathbf{e}}}$.

(b) Inst. M: $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(P, A_{\mathbf{x}} \cdot B'_{\mathbf{x}}), M)$
$\wedge\ X = \mathit{prefix}_i(A_{\mathbf{x}} \cdot B'_{\mathbf{x}} \cdot M)$. [Def. 4.4.5 and 1]

(c) Inst. N: $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(R, B_{\mathbf{e}}), N)$
$\wedge Z = \mathit{prefix}_k(B_{\mathbf{e}} \cdot N)$. [Def. 4.4.5 and 1]

(d) Inst. M': $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(P, \mathit{prefix}_{\|A_{\mathbf{x}}\|}(A_{\mathbf{x}} \cdot B'_{\mathbf{x}})), M')$
$\wedge M' = \mathit{prefix}_m(\mathit{drop}_{\|A_{\mathbf{x}}\|}(A_{\mathbf{x}} \cdot B'_{\mathbf{x}}) \cdot M)$. [Def. 4.4.9.2]

(e) $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(P, A_{\mathbf{x}}), M') \wedge M' = \mathit{prefix}_m(B'_{\mathbf{x}} \cdot M)$.

(f) $X = \mathit{prefix}_i(A_{\mathbf{x}} \cdot M')$.

(g) Let $Y \equiv \mathit{prefix}_j(A_{\mathbf{e}} \cdot M')$.

(h) $\mathsf{reach}^{\mathbf{e}}(\mathit{base}(A, P, Q), M')$.

(i) $(X, Y) \in \gamma_{i,j}(\breve{A_{\mathbf{x}}} \cdot \widehat{A_{\mathbf{e}}}, P, Q)$. [6f, 6g, and 6h]

(j) $Y = \mathit{prefix}_j(A_{\mathbf{e}} \cdot B'_{\mathbf{x}} \cdot M)$.

(k) $Y = \mathit{prefix}_j(B_{\mathbf{x}} \cdot M)$.

(l) $B'_{\mathbf{x}} = \epsilon \rightarrow \mathit{rem}(P, A_{\mathbf{x}} \cdot B'_{\mathbf{x}}) = \mathit{rem}(Q, A_{\mathbf{e}}) = \mathit{rem}(Q, A_{\mathbf{e}} \cdot B'_{\mathbf{x}})$. [$A$ is consistent]

(m) $B'_{\mathbf{x}} \neq \epsilon \rightarrow \mathit{rem}(P, A_{\mathbf{x}} \cdot B'_{\mathbf{x}}) = \mathit{last}(B'_{\mathbf{x}}) = \mathit{rem}(Q, A_{\mathbf{e}} \cdot B'_{\mathbf{x}})$.

(n) $\mathit{rem}(P, A_{\mathbf{x}} \cdot B'_{\mathbf{x}}) = \mathit{rem}(Q, B_{\mathbf{x}})$.

(o) $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, B_{\mathbf{x}}), M)$.

(p) $(Y, Z) \in \gamma_{j,k}(\breve{B_{\mathbf{x}}} \cdot * \cdot \widehat{B_{\mathbf{e}}}, Q, R)$. [6c, 6k, and 6o]

(q) $(X, Z) \in \alpha$.

Case: $A_{\mathbf{w}} = * \wedge B_{\mathbf{w}} = *$.

(a) $\mathit{match}(A \cdot B) = \breve{A_{\mathbf{x}}} \cdot * \cdot \widehat{B_{\mathbf{e}}}$.

(b) Inst. M: $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(P, A_{\mathbf{x}}), M)$
$\wedge X = \mathit{prefix}_i(A_{\mathbf{x}} \cdot M)$. [Def. 4.4.5 and 1]

(c) Inst. N: $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(R, B_{\mathbf{e}}), N)$
$\wedge Z = \mathit{prefix}_k(B_{\mathbf{e}} \cdot N)$. [Def. 4.4.5 and 1]

(d) Inst. N': $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, B_{\mathbf{x}}), N')$. [Def. 4.4.9.1]

(e) Inst. M': $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, \mathit{prefix}_{\|A_{\mathbf{e}}\|}(B_{\mathbf{x}})), M')$
$\wedge\ M' = \mathit{prefix}_m(\mathit{drop}_{\|A_{\mathbf{e}}\|}(B_{\mathbf{x}}) \cdot N')$. [Def. 4.4.9.2]

(f) $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, A_{\mathbf{e}}), M') \wedge M' = \mathit{prefix}_m(B'_{\mathbf{x}} \cdot N')$.

(g) Let $Y \equiv \mathit{prefix}_j(B_{\mathbf{x}} \cdot N')$.

(h) $(Y, Z) \in \gamma_{j,k}(\overset{\smile}{B_{\mathbf{x}}} \cdot * \cdot \widehat{B_{\mathbf{e}}}, Q, R)$.

(i) $Y = \mathit{prefix}_j(A_{\mathbf{e}} \cdot M')$.

(j) $(X, Y) \in \gamma_{i,j}(\overset{\smile}{A_{\mathbf{x}}} \cdot * \cdot \widehat{A_{\mathbf{e}}}, P, Q)$.

(k) $(X, Z) \in \alpha$.

7. The case of $\|A_{\mathbf{e}}\| \geq \|B_{\mathbf{x}}\|$ follows the same reasoning as above.

□

Lemma 4.4.16. *For all methods P,*

$$\gamma_{h,m}(\mathit{record}^{\mathbf{t}}(\_), P, P) = \{\mathit{record}^{\mathbf{e}}(M') \mid \mathsf{reach}^{\mathbf{e}}(P, M')\}.$$

Proof. Follows trivially from definitions. □

Lemma 4.4.17. *For all consistent and derivable transformer strings $B \equiv \overset{\smile}{B_{\mathbf{x}}} \cdot B_{\mathbf{w}} \cdot \widehat{B_{\mathbf{e}}} \in$ $\mathbf{CtxtT}^{\mathbf{t}}_{h,m}$ from R to P, heap allocation sites H in R, and invocations in I in P,*

$$\gamma_{m,m}(\mathit{merge}^{\mathbf{t}}(H, I, B), P, Q) \subseteq \{\mathit{merge}^{\mathbf{e}}(H, I, B') : B' \in \gamma_{h,m}(B, R, P)\}.$$

Proof.

Call-site sensitivity. We must show that

$$\gamma_{m,m}(\mathit{trunc}_{m,m}(\overset{\smile}{B_{\mathbf{e}}} \cdot \widehat{B_{\mathbf{e}}} \cdot \widehat{I}), P, Q) \subseteq \{(Y', \mathit{prefix}_m(I \cdot Y')) : (X', Y') \in \gamma_{h,m}(\overset{\smile}{B_{\mathbf{x}}} \cdot B_{\mathbf{w}} \cdot \widehat{B_{\mathbf{e}}}, R, P)\}.$$

1. Let $(X,Y) \in \gamma_{m,m}(\mathit{trunc}_{m,m}(\breve{B}_{\mathbf{e}} \cdot \widehat{B}_{\mathbf{e}} \cdot \widehat{I}), P, Q)$.

2. Let $\alpha \equiv \{(Y', \mathit{prefix}_m(I \cdot Y')) : (X', Y') \in \gamma_{h,m}(\breve{B}_{\mathbf{x}} \cdot B_{\mathbf{w}} \cdot \widehat{B}_{\mathbf{e}}, R, P)\}$.

3. $(X,Y) \in \gamma_{m,m}(\breve{B}_{\mathbf{e}} \cdot \widehat{B}_{\mathbf{e}} \cdot \widehat{I}, P, Q)$. [Lemma 4.4.13]

4. Inst. M: $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(P, B_{\mathbf{e}}), M)$
   $\wedge\ X = \mathit{first}_m(B_{\mathbf{e}} \cdot M)$
   $\wedge\ Y = \mathit{first}_m(I \cdot B_{\mathbf{e}} \cdot M)$. [Def. 4.4.5]

5. $(X', \mathit{first}_h(B_{\mathbf{e}} \cdot M)) \in \gamma_{h,m}(\breve{B}_{\mathbf{x}} \cdot B_{\mathbf{w}} \cdot \widehat{B}_{\mathbf{e}}, R, P)$. [Def. 4.4.5 and 4]

6. $(\mathit{first}_m(B_{\mathbf{e}} \cdot M), \mathit{first}_m(I \cdot B_{\mathbf{e}} \cdot M)) \in \alpha$.

7. $(X,Y) \in \alpha$.

Object sensitivity. We must show that

$$\gamma_{m,m}(\breve{B}_{\mathbf{e}} \cdot B_{\mathbf{w}} \cdot \widehat{B}_{\mathbf{x}} \cdot \widehat{H}, P, Q) \subseteq \{(Y', H \cdot X') : (X', Y') \in \gamma_{h,m}(\breve{B}_{\mathbf{x}} \cdot B_{\mathbf{w}} \cdot \widehat{B}_{\mathbf{e}}, R, P)\}.$$

1. Let $(X,Y) \in \gamma_{m,m}(\breve{B}_{\mathbf{e}} \cdot B_{\mathbf{w}} \cdot \widehat{B}_{\mathbf{x}} \cdot \widehat{H}, P, Q)$.

2. Let $\alpha \equiv \{(Y', H \cdot X') : (X', Y') \in \gamma_{h,m}(\breve{B}_{\mathbf{x}} \cdot B_{\mathbf{w}} \cdot \widehat{B}_{\mathbf{e}}, R, P)\}$.

3. *Case:* $B_{\mathbf{w}} = \epsilon$.

   (a) Inst. M: $X = \mathit{first}_m(B_{\mathbf{e}} \cdot M)$
       $\wedge\ Y = \mathit{first}_m(H \cdot B_{\mathbf{x}} \cdot M)$
       $\wedge\ \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(P, B_{\mathbf{e}}), M)$. [Def. 4.4.5]

   (b) $(\mathit{first}_h(B_{\mathbf{x}} \cdot M), \mathit{first}_m(B_{\mathbf{e}} \cdot M)) \in \gamma_{h,m}(\breve{B}_{\mathbf{x}} \cdot \widehat{B}_{\mathbf{e}}, R, P)$. [Def. 4.4.5 and 3a]

   (c) $(\mathit{first}_m(B_{\mathbf{e}} \cdot M), \mathit{first}_m(H \cdot B_{\mathbf{x}} \cdot M)) \in \alpha$.

   (d) $(X,Y) \in \alpha$.

   Case: $B_{\mathbf{w}} = *$.

(a) Inst. M: $X = \mathit{first}_m(B_{\mathbf{e}} \cdot M) \wedge \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(P, B_{\mathbf{e}}), M)$. [Def. 4.4.5]

(b) Inst. N: $Y = \mathit{first}_m(H \cdot B_{\mathbf{x}} \cdot M) \wedge \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(Q, H \cdot B_{\mathbf{x}}), N)$. [Def. 4.4.5]

(c) $\mathit{rem}(Q, H) = R$.

(d) $\mathit{rem}(Q, H \cdot B_{\mathbf{x}}) = \mathit{rem}(R, B_{\mathbf{x}})$.

(e) $\mathsf{reach}^{\mathbf{e}}(\mathit{rem}(R, B_{\mathbf{x}}), N)$.

(f) $(\mathit{first}_h(B_{\mathbf{x}} \cdot M), \mathit{first}_m(B_{\mathbf{e}} \cdot N)) \in \gamma_{h,m}(\overset{\smile}{B_{\mathbf{x}}} \cdot * \cdot \widehat{B_{\mathbf{e}}}, R, P)$. [Def. 4.4.5, 3a, and 3b]

(g) $(\mathit{first}_m(B_{\mathbf{e}} \cdot M), \mathit{first}_m(H \cdot B_{\mathbf{x}} \cdot N)) \in \alpha$.

(h) $(X, Y) \in \alpha$.

□

Lemma 4.4.18. *For all invocations I in P and derivable partial method context M of P,*

$$\{\gamma_{m,m}(\mathit{merge\_s}^{\mathbf{t}}(I, M), P, Q)\} \subseteq \{\mathit{merge\_s}^{\mathbf{e}}(I, M') \mid \mathsf{reach}^{\mathbf{e}}(P, M')\}.$$

Proof.

Call-site sensitivity. We must show that

$$\gamma_{m,m}(\widehat{I}, P, Q) \subseteq \{(M', \mathit{prefix}_m(I \cdot M')) \mid \mathsf{reach}^{\mathbf{e}}(P, M')\}.$$

1. Let $(X, Y) \in \gamma_{m,m}(\widehat{I}, P, Q)$.
2. Let $\alpha \equiv \{(M', \mathit{prefix}_m(I \cdot M')) \mid \mathsf{reach}^{\mathbf{e}}(P, M')\}$.
3. Inst. N: $X = N \wedge Y = \mathit{prefix}_m(I \cdot N) \wedge \mathsf{reach}^{\mathbf{e}}(P, N)$. [Def. 4.4.5 and 1]
4. $(X, Y) \in \alpha$.

Object sensitivity. We must show that

$$\gamma_{m,m}(\overset{\smile}{M} \cdot \widehat{M}, P, Q) \subseteq \{(M', M') \mid \mathsf{reach}^{\mathbf{e}}(P, M')\}.$$

1. Let $(X, Y) \in \gamma_{m,m}(\overset{\smile}{M} \cdot \widehat{M}, P, Q)$.

2. Let $\alpha \equiv \{(M', M') \mid \mathsf{reach}^{\mathbf{e}}(P, M')\}$.

3. Inst. N: $X = Y = \mathit{prefix}_m(M \cdot N) \wedge \mathsf{reach}^{\mathbf{e}}(\mathit{rem}(P, M), N)$. [Def. 4.4.5 and 1]

4. $\mathsf{reach}^{\mathbf{e}}(P, \mathit{prefix}_m(M \cdot N))$. [$M$ is derivable]

5. $(X, Y) \in \alpha$.

□

Theorem 4.4.19. $\mathcal{A}^{\mathbf{t}\Box}_{m,h} \subseteq \mathcal{A}^{\mathbf{e}\Box}_{m,h}$, *for call-site and object sensitivity.*

Proof. $\mathcal{C}^{\mathbf{t}\Box}_{m,h} \subseteq \mathcal{C}^{\mathbf{e}\Box}_{m,h}$ follows from Lemmas 4.4.14, 4.4.15, 4.4.16, 4.4.17, and 4.4.18, by induction on the derivation of explicit and transformer strings according to the rules in Figure 3.3. Thus $\mathcal{A}^{\mathbf{t}\Box}_{m,h} \subseteq \mathcal{A}^{\mathbf{e}\Box}_{m,h}$. □

Chapter 5

Implementation

This section describes DLE, a Datalog engine designed for concisely expressing and solving static analyses. DLE supports extensions to Datalog such as complex terms (Section 5.2) and stratified negation (Section 5.4), supports a rich language for specifying data structures of relations (Section 5.6), and a mechanism of efficiently evaluating operations used by static analyses (Section 5.9). DLE compiles Datalog programs into native code using the LLVM Compiler Infrastructure [17].

5.1 Introduction to Datalog

Variables range over a *universe of discourse (universe). Terms* are either *variables* or *constant symbols (constants)*. An *atomic formula (atom)* is the smallest syntactic element that can be assigned a truth value. A *literal* is an atom of the form "$P(t_1, \ldots, t_n)$", where P is a *predicate* of *arity* n, and $t_1, \ldots, t_n$ are terms that are *arguments* to the predicate. A Datalog program consists of *clauses* of the from "$P(\ldots) \mathrel{:-} Q_1(\ldots), \ldots, Q_n(\ldots)$.", where the literal "$P(\ldots)$" is called the *head* of the rule and the sequence of atoms "$Q_1(\ldots), \ldots, Q_n(\ldots)$" is its *body*. (An atom may also be a *negated literal* of the form "$\neg Q(\ldots)$". Negated literals are discussed in the next section. For now, all literals are *positive*.) A clause with an empty body is called a *fact* and is written without the ":-" connective; otherwise a clause is a rule. All variables that appear in the head of a clause must appear in the body. This requirement ensures that all facts are *ground literals*, meaning only constants appear as terms inside facts. A set of facts of a particular predicate is called a *relation*.

Intensional database (IDB) predicates represent relations defined by rules, and *extensional database (EDB) predicates* represent relations provided externally to the Datalog engine. In the context of Datalog's application to static analysis, the "Datalog program" consists of rules that define the analysis, and "the input" is a set of facts derived from an intermediate representation of a program that is to be analyzed.

The *Herbrand universe* of a Datalog program is the set of all constants that appear in the program (including constants in EDB relations). Its *Herbrand base* is the set of all ground atoms that can be formed by predicate symbols that appear in the program. An *Herbrand interpretation* of the program is a total function that maps elements of the Herbrand base to true or false values. If there exists an interpretation in any universe of discourse that satisfies a set of clauses T, then there is a Herbrand interpretation that satisfies T. Notation-wise, an interpretation is written as a set consisting only of facts that are true with respect to the interpretation.

A *ground substitution* (or just substitution) is a function from variables to constants. Given a clause C and a substitution θ, the *application* of θ to C, written $C\theta$, is a ground clause obtained by replacing variables with constants using θ. $C\theta$ is called a *ground instance* of C.

We describe the *model-theoretic semantics* of a Datalog program. The body of a ground instance of a Datalog clause is true under an interpretation $\mathcal{I}$ iff all literals in the body are in $\mathcal{I}$. A clause "L :- ϕ." is true under an interpretation $\mathcal{I}$ iff for all ground substitutions θ, if $\phi\theta$ is true under $\mathcal{I}$, then $L\theta$ must be in $\mathcal{I}$. An interpretation is a *model* of a Datalog program if all clauses of the program are true. The intersection of all Herbrand models of a program defines the unique *minimum model* of the program [1].

5.2 Complex Terms

Pure Datalog has the desirable property that the Herbrand universe, base, and interpretation are all finite. When extended with *complex terms* (uninterpreted functions in logic), this property is lost, but the extension offers several benefits discussed later in this section.

The following definitions from the previous section are modified to support complex terms:

- Terms are either variables or complex terms. A complex term is of the form $f(t_1, \ldots, t_n)$, where f is a n-ary *function symbol* and $t_1, \ldots, t_n$ are terms that are arguments of the function. Constants are special cases of complex terms and are 0-ary functions.
- The Herbrand universe of a program T is defined as the smallest set $\mathcal{U}$ that satisfies:
    1. $\mathcal{U}$ contains all constants in T.
    2. If $t_1, \ldots, t_n \in \mathcal{U}$ and f is a n-ary function symbol in T, then $f(t_1, \ldots, t_n) \in \mathcal{U}$.

Under the new definition, the Herbrand universe, base, and interpretation are infinite if the program contains a function symbol of arity one or greater. If the program does have a finite minimal model, then DLE computes it, but if it does not, DLE does not terminate until it runs out of memory. Determining whether a Datalog program with complex terms has a finite minimal model is an undecidable problem [7]. The onus that a program does have a finite minimal model is on the writer of the program.

Complex terms offer a notational convenience of forming a new value out of several components. For example, consider the following example program without complex terms that computes a *lives-together* relation, given a *lives-at* relation:

```
lives_together(Person1,Person2) :-
    lives_at(Person1,StreetNo,Street,City),
    lives_at(Person2,StreetNo,Street,City).
```

The program can be improved with better style by identifying that the street number, street name, and city are components of a higher-level concept of an *address*, and that rules that manipulates addresses, but not components of addresses, can be expressed more concisely using complex terms:

```
lives_at_address(Person,address(StreetNo,Street,City)) :-
    lives_at(Person1,StreetNo,Street,City).
lives_together(Person1,Person2) :-
    lives_at_address(Person1,Addr),
    lives_at_address(Person2,Addr).
```

The second use of complex terms is to define data structures. For example, a polymorphic *cons-list* data type can be declared as follows:

```
data list a = cons a (list a) | nil.
```

`a` is a type variable. The above declaration declares two *data constructors* `cons` and `nil` that have types `a` $\times$ `list a` $\rightarrow$ `list a` and `list a`, respectively. Data constructors are the function symbols of complex terms. Note that arguments to functions do not curry in DLE.

Within terms, constants of primitive types (integers and strings) do not require an empty pair of parenthesis, but 0-ary data constructors of user-defined types require an empty pair of parenthesis (e.g. `nil()`).

DLE has a builtin syntax for expressing lists: the term `[1,2,3]` is equivalent to `cons(1,cons(2,cons(3,nil())))`.

5.3 Evaluation

DLE uses semi-naive evaluation, a bottom-up evaluation technique for logic programs, to compute the minimal model of a program. We first describe *naive evaluation*: Let M_0 be the interpretation of EDB predicates, represented as a set of facts. Given a set of facts M, let $\overline{P}(M)$ be the facts the can be derived from M using a clause from P. Naive evaluation computes the minimum model, which is the least fixed point of $\overline{P}$, by iteration starting from M_0.

The disadvantage of naive evaluation is that it repeatedly derives the same facts: all facts derived in the n^{th} iteration are re-derived in all iterations after the n^{th} iteration. *Semi-naive evaluation* improves upon naive evaluation by ensuring that subsequent derivations of facts use at least one fact derived from the previous iteration. This approach is accomplished by recording newly derived facts in *delta relations*. For every IDB relation Q, let Q^{Δ} record facts newly derived in the previous iteration and let Q^{new} record facts derived in the current iteration.

A program is transformed into a *delta-transformed program* as follows: Every rule in the original program is duplicated in the transformed program for every IDB predicate in its body, where each duplicated rule has an IDB predicate replaced by its corresponding delta predicate[1]. For example, a rule "$P(\ldots) \mathrel{:\!-} Q_1(\ldots), \ldots, Q_m(\ldots), R_1(\ldots), \ldots, R_n(\ldots).$", where $Q_1, \ldots, Q_m$ are IDB predicates and $R_1, \ldots, R_n$ are EDB predicates, is transformed into the following set of *delta rules*:

$$\begin{gathered}
P(\ldots) \mathrel{:\!-} Q_1^{\Delta}(\ldots), Q_2(\ldots), \ldots, Q_m(\ldots), R_1(\ldots), \ldots, R_n(\ldots).\\
P(\ldots) \mathrel{:\!-} Q_1(\ldots), Q_2^{\Delta}(\ldots), \ldots, Q_m(\ldots), R_1(\ldots), \ldots, R_n(\ldots).\\
\vdots\\
P(\ldots) \mathrel{:\!-} Q_1(\ldots), Q_2(\ldots), \ldots, Q_m^{\Delta}(\ldots), R_1(\ldots), \ldots, R_n(\ldots).
\end{gathered}$$

[1]For simplicity, we ignore a complication that arises when a relation appears more than once in a body [1].

```
procedure COMPUTE($P$)
    Let $P'$ be the delta-transformed program of $P$.
    for all EDB clauses $R$ in $P'$ do
        EVAL($R$).
    SWAPDELTAS().
    while there exists a non-empty relation $Q^{last}$ do
        for all delta rules $R$ in $P'$ do
            EVAL($R$).
        SWAPDELTAS().

    subprocedure SWAPDELTAS()
        for all IDB relations $Q$ do
            $Q^{\Delta} \leftarrow Q^{new}$.
            $Q^{new} \leftarrow \emptyset$.
```

Figure 5.1: Semi-naive evaluation.

Clauses in the original program without IDB predicates in their bodies are *EDB clauses*, and are present in the transformed program intact. All facts are EDB clauses.

Figure 5.1 contains pseudo-code of a semi-naive evaluation algorithm. Relations corresponding to predicate symbols in a program, including delta relations, form the global mutable state of the pseudo-code algorithm. Given a rule R with a head predicate Q, EVAL(R) derives a set of facts that can be derived using R, and inserts the facts into Q^{new}. The details of this procedure are given in Section 5.8.

5.4 Stratification

Stratified Datalog extends Datalog by allowing negated literals (e.g. "$\neg P(\ldots)$") as atoms.

A *stratification* of a Datalog program (represented as a set of clauses P) is a partition $P_1, \ldots, P_n$ of P and a *stratification mapping* τ from predicates in P to $1 \ldots n$ such that

1. $\tau(Q) = 0$ for all EDB predicates Q,
2. all clauses that derive a predicate Q are in $P_{\tau(Q)}$,
3. if $Q(\ldots) \text{:-} \ldots S(\ldots) \ldots$ is in P then $\tau(S) \leq \tau(Q)$,
4. and if $Q(\ldots) \text{:-} \ldots \neg S(\ldots) \ldots$ is in P then $\tau(S) < \tau(Q)$.

Each P_i is called a *stratum* of P. A Datalog program with negation is *stratifiable* if there exists a stratification of it [1].

The interpretation of a ground body is redefined to support negated literals: a ground body of a stratified Datalog clause is true under an interpretation $\mathcal{I}$ iff all positive literals are in $\mathcal{I}$ and all negative literals are not in $\mathcal{I}$.

The semantics of a stratifiable Datalog program is obtained by defining a sequence of interpretations for each of its strata. Each stratum P_i can be thought of as a separate and complete Datalog program, where all predicates Q such that $\tau(Q) < i$ are EDB predicates of P_i. Let $P_i(\mathcal{I})$ be the minimum model of P_i given an interpretation $\mathcal{I}$ of EDB predicates of P_i. Given a stratification of a Datalog program P, define $\mathcal{I}_i = \mathcal{I}_{i-1} \cup P_i(\mathcal{I}_{i-1})$ for all $0 < i < n$ and let $\mathcal{I}_0$ be the interpretations of all EDB predicates of P. Then, the model-theoretic semantic of P is $\mathcal{I}_n$. All stratifications of P result in the same minimum model [1].

Giving semantics to negation is not the only purpose of stratification. Stratification also improves the efficiency of semi-naive evaluation by reducing the number of delta rules in a program. The reduction occurs because an IDB predicate in stratum i becomes an EDB predicate in all strata larger than i.

For example, consider the following program that computes transitive closure:

```
edge(X,Y) :- xedge(X,Y).
path(X,Y) :- edge(X,Y).
path(X,Z) :- path(X,Y), edge(Y,Z).
```

The program has a single EDB predicate, `xedge`. Without stratification, the delta-transformed program is as follows:

```
edge(X,Y) :- xedge(X,Y).
path(X,Y) :- edge^Δ(X,Y).
path(X,Z) :- path^Δ(X,Y), edge(Y,Z).
path(X,Z) :- path(X,Y), edge^Δ(Y,Z).
```

The program can be stratified into two strata P_1 and P_2 where P_1 contains the first rule and P_2 contains the remaining two. In P_2, `edge` is a EDB predicate, and thus the delta-transformed program of P_1 and P_2 contains three rules in total:

```
// P1:
edge(X,Y) :- xedge(X,Y).
// P2:
path(X,Y) :- edge(X,Y).
path(X,Z) :- path^Δ(X,Y), edge(Y,Z).
```

In general, the reduction in the number of delta rules results in fewer relational joins to compute the minimal model. Thus, using the most *fine-grained* stratification is advantageous in terms of efficiency. Such a stratification is obtained by building a *predicate-dependency graph* of a program, which consists of predicates as nodes and a directed edge from p to q if the program has a rule where p appears in its body, and q appears in its head. The condensation graph (strongly-connected components are merged into a single node) of the predicate-dependency graph is computed using a strongly-connected-component algorithm. A post-order traversal of the condensation graph gives a sequence of sets of predicates $R_1, \ldots, R_n$. Strata $P_1, \ldots, P_n$ are formed by placing a rule that derives a predicate R_i into P_i.

5.5 Syntax and Type System

Type declaration syntax in DLE is based on Haskell's syntax. For example, the statement

```
data t = d1 int | d2 char int.
```

declares a type `t` with two data constructors: `d1` : `int` $\rightarrow$ `t` and `d2` : `char` $\times$ `int` $\rightarrow$ `t`.

The following is an example predicate declaration:

```
predicate p(int,int,t).
```

The above statement declares that the interpretation of `p` is a ternary relation that is a subset of `int` $\times$ `int` $\times$ `t`.

The following is the grammar for rules and facts:

term $\rightarrow$ *variable.*

term $\rightarrow$ *constant.*

term $\rightarrow$ *identifier* '`(`' *term_list* '`)`'.

term_list $\rightarrow$ ϵ | *term* ('`,`' *term*)*.

literal $\rightarrow$ *identifier* '`(`' *term_list* '`)`'.

atom $\rightarrow$ *literal* | '$\neg$' *literal* | '`(`' *term* '`)`'.

rule $\rightarrow$ *atom* '`.`'.

rule $\rightarrow$ *atom* '`:-`' *atom* ('`,`' *atom*)* '`.`'.

variable, *constant*, and *identifier* are lexical elements, where variable identifiers start with an uppercase letter and non-variable identifiers start with a lowercase letter. Constants are either numbers or strings within double quotes.

In addition to uninterpreted functions, DLE supports builtin and user-defined functions. Builtin functions include arithmetic and relational operations on integer-valued terms. User-defined functions can be defined in a limited functional-programming-language dialect:

function_term → *term*.
function_term → '`if`' *term* '`then`' *term* '`else`' *term*.
function_defn → *identifier* '`(`' *term_list* '`)`' '`=`' *function_term*.

Functions are declared using the `function` keyword. For example the following statement declares a function that takes a value of type "`list a`" (`a` is a type variable) and returns an integer:

```
function count_zero :: list a -> int.
```

A function that counts the number of occurrences of the number '0' in a cons-list can be defined through pattern matching as follows:

```
count_zero(nil()) = 0.
count_zero(cons(x,y)) = if x == 0 then
   1+count_zero(y) else count_zero(y).
```

5.6 Index Declarations and Cost Estimates

Similar to the LogicBlox engine used by DOOP [12, 5], our Datalog engine exposes indexing decisions at the language level. *Index declarations* describe how predicates are materialized as in-memory data structures by the engine. A single *logical relation* may be materialized as multiple *physical relations* (or *indices*), all representing the same relation, but different indices may have different join costs depending on the attributes over which the join is performed. The following grammar describes index declarations:

> *number_list* → ‘`[`’ *number* (‘`,`’ *number*)* ‘`]`’.
>
> *index_decl* → `index` *identifier* *index* ‘`secondary`’ (‘`.`’)?.
>
> *index* → ‘`value_list`’ ‘`(`’ *number* ‘`)`’.
>
> *index* → ‘`hash_set`’ ‘`(`’ *number* ‘`)`’.
>
> *index* → ‘`array_map`’ ‘`(`’ *number* ‘`,`’ *number* ‘`,`’ *index* ‘`)`’.
>
> *index* → ‘`hash_map`’ ‘`(`’ *number_list* ‘`,`’ *number* ‘`,`’ *index* ‘`)`’.
>
> *index* → ‘`value`’ ‘`(`’ ‘`)`’.

Two operations on indices are *scans* and *inserts*. Given a literal, a scan operation enumerates all tuples in an index that unify with the literal, and an insert operation inserts a fact into an index. When a DLE rule is *evaluated*, all relations in the body of the rule are *joined* through scan operations, and newly derived tuples are inserted into the relation at the head of the rule. This process is described in more detail in Section 5.8.

An index declaration describes a data structure by the composition of *index elements*. `value_set`, `hash_set`, and `value` are *leaf elements*, while `array_map` and `hash_map` are *outer elements*. Index elements nested within an outer element O are *sub-elements* of O and O is a *super-element* to all its sub-elements. Scan and insert operations on indices are implemented as a composition of operations on the index elements that compose the index.

Scans of indices are performed with respect to a literal of the predicate (the *join literal*) and a set of *bound variables*. The join literal and set of bound variables determine the cost of scanning the index. Index elements may specify a set of *key attributes* (a set of ordinals that specifies attributes of the predicate) that determines whether scans of the index element are *lookup scans* or *full scans*: if all arguments of the join literal specified as a key attribute in an index element are either bound or are specified as a key attribute in a super-element of the index element, then a lookup scan is performed; otherwise a full scan is performed.

The declaration of an index element specifies a *cost estimate*, which is a user-specified and unit-less number that describes the cost of performing a full scan of the index element relative to other indices. The cost of a lookup scan is always 1. The cost estimates determine a *join order*, which is a process described in Section 5.7.

Each of the index elements are described below:

- `value_list`(N) specifies a *value-list element* with a cost estimate of N to perform a full scan of the index element. Lookup scans are not possible (all scans are full scans). The value-list element is represented in memory as an array of tuples.

  For example, consider the following index declaration for a predicate p:

  ```
  index p value_list( 100 ).
  ```

  The time to perform a scan of this index is always linearly proportional to the size of the relation `p`. The number 100 is the cost estimate of scanning the index.

- `hash_set`(N) specifies a *hash-set element* with a cost estimate of N to perform a full scan of the index. All attributes that are not specified as key attributes in a hash-set element's super-elements are key attributes of the hash-set element. The hash-set element is represented in memory as a hash table.

  For example, consider the following index declaration:

  ```
  index p hash_list( 100 ).
  ```

  Scans of this index takes constant time if the join literal is fully bound (all variables in the literal appear in the set of bound variables), and have a cost estimate of 1. Otherwise, scans take time proportional to the size of `p` with a cost estimate of 100.

- `array_map`(A, N, I) specifies an *array-map element*: the A^{th} attribute is the element's key attribute, N is its cost estimate, and I is its direct sub-element. The array-map element is represented in memory as an array of its sub-element.

  For example, consider the following index declaration:

  ```
  index p array_map( 1, 10, value_list( 100 ) ).
  ```

The number `1` specifies that the first attribute is used as the key to an associative map, implemented as an array, that maps to `value_list` elements. If the first attribute is bound, for example if the join literal is `p`$(5, Y, Z)$ and the set of bound variables is empty, then the scan operation on the index takes time proportional to the number of tuples in `p` whose first attribute has the value 5. If the first attribute is not bound, then the scan operation takes time proportional to the size of `p`. The cost estimate of scanning an outer index element is formed by taking the product of the specified cost estimate of the outer index and the computed cost of scanning the outer index's sub-index (i.e., the `value_list`). Thus, if the first attribute is bound, the cost estimate of scanning the index is $1 \cdot 100 = 100$; if the first attribute is not bound, the cost estimate is $10 \cdot 100 = 1000$.

- `hash_map`(A, N, I) specifies a *hash-map element*: A is a list of ordinals that specifies the element's key attributes, N is its cost estimate, and I is its direct sub-element. The array-map element is represented in memory as a hash table that maps to its sub-element.

  For example, consider the following index declaration:

  ```
  index p hash_map( [1,2], 5, hash_set( 100 ) ).
  ```

  If the join literal is fully bound, then scanning the index takes constant time, and the cost estimate is 1. If only the first and second attributes of the join literal are bound, then the cost estimate is 100. If only the third attribute is bound, then the cost estimate is 5. Otherwise, the cost estimate is $5 \cdot 100 = 500$.

- `value`() specifies a *value element*: a data structure that holds only one tuple. An index with a value element implies a functional dependency from the set of all key attributes of super-elements of the value element to the remaining non-key attributes. For example, an index declaration "`index p value().`" implies that the size of the relation `p` is at most one; an index declaration "`index q hash_map( [1,2], 5,` `value() ).`" of a 4-ary relation `q` implies that there is a functional dependency from the first and second attributes to the third and fourth attributes. Violating a function dependency produces an assertion error during evaluation of a program.

Predicates may be associated with multiple indices through multiple index declarations. An index may either be a *primary* or a *secondary* index. A relation may have zero or one primary index and any number of secondary indices. The differentiation between primary and secondary is irrelevant for joins (scanning) but affects insert operations. Insert operations on primary indices are *unique inserts*, meaning that duplicate tuples are not inserted. *Non-unique inserts* are performed on secondary indices, meaning that the index may contain duplicate tuples. If a predicate is associated with a primary index, then inserts into secondary indices are performed after inserting into the primary index, and performed only if the tuple being inserted is not a duplicate. If a predicate is not associated with a primary index, then tuples are inserted into secondary indices unconditionally, meaning that they may contain duplicates. Unique inserts and non-unique inserts are algorithmically identical for all index elements except for `value_list`s, and take constant time. For `value_list`s, unique inserts take linear time and non-unique inserts take constant time.

The differentiation between primary and secondary indices enables fine-tuning of the choice of optimal data structures: full scan and non-unique-insert operations on `value_list` index elements are more efficient than their corresponding operations on `hash_set` index elements; thus `value_list` index elements are suitable for secondary indices but not for primary indices due to their linear time unique-insert time complexity. If no duplicate tuples of a predicate are expected to be derived, then the predicate may be associated only with secondary indices using `value_list` index elements.

5.7 Join Order

A *join order* of a rule is a total ordering of literals that appear in the body of the rule, where logical relations are replaced by physical relations (indices). Join orders are determined on the rules of a delta-transformed program. The join order determines the application order of binary relational-join operations performed during evaluation: specifically, the joins are performed in a linear chain of applications, a left fold on the total ordering of indices. For example, if a join ordering is $[p(A, B), q(B, C), r(C, D)]$, then relational joins (notated with $\bowtie_{x=y}$ where x and y denote the attributes over which the join is performed) are performed as $((p \bowtie_{2=1} q) \bowtie_{2=1} r)$.

Join orders are determined by a greedy algorithm. Let L be the set of literals that appear in a body of a rule. The algorithm to determine the join order of the rule starts with an initially empty list of literals I, an empty set of joined literals J, and an empty set of bound variables B.

- If L contains a literal of a delta relation, then the literal is added to I and J, and the variables that appear in the literal are added to B.
- For each index P' of a literal $P(\bar{x})$ in $L \setminus J$,
  - a cost estimate of scanning P' with respect to the set of bound variables B is tallied;
  - the index Q' with the lowest-tallied cost estimate is determined;
  - $Q'(\bar{x})$ is appended to I;
  - $Q(\bar{x})$ is added to J;
  - and the variables that appear in $\bar{x}$ are added to B.
- This process repeats until $L = J$.

Variables that appear inside a builtin or user-defined function must be bound before the literal containing the function can be appended to a join order.

The rationale for choosing the delta relation as the first relation in the join order is based on an assumption that the delta relation is the smallest relation in the join. This assumption may not be true in all programs. For example, if a program contains a rule "$p(\dots)$:- $p(\dots), q(\dots), r(\dots)$.", where q and r are EDB relations in the stratum containing the rule, then the only two join orders admitted by DLE are $((p^{\Delta} \bowtie q) \bowtie r)$ and $((p^{\Delta} \bowtie r) \bowtie q)$. If the join order $(p^{\Delta} \bowtie (q \bowtie r))$ is desired, then the program must be refactored into two rules using a new relation:

$$s(\dots) \text{ :- } q(\dots), r(\dots).$$
$$p(\dots) \text{ :- } p(\dots), s(\dots).$$

5.8 Rule Evaluation and Code Generation

Rule evaluation is performed by nested scanning of indices in a join order. Figure 5.2 presents the pseudo-code of the algorithm. Eval(R) evaluates a clause R. If R is a fact, then it has an empty join order ($n = 0$). ϕ is a mapping from variables in a rule to ground terms. As indices are scanned, ϕ is updated with new variable mappings, which may overwrite previous mappings. EvalJoin(k) performs a join of the k^{th} literal in the join order of R. The function *subst* applies a variable mapping to a literal and returns a ground literal (a fact).

DLE uses the LLVM Compiler Infrastructure [17] to generate code that evaluates each rule. Specifically, for each clause, DLE generates a function that implements the Eval procedure where the EvalJoin subprocedure is completely unrolled into Eval. Since the join order, and thus the indices accessed, are known at compile-time when generating the function for a particular rule, the code for scanning the indices is generated directly in the function that evaluates the rule. This form of per-rule code generation, as opposed to an

```
procedure $\textsc{Eval}(R)$
    Let $I_1, \ldots, I_n$ be the join order of clause $R$.
    Let $H$ be the head literal of $R$.
    $\phi \leftarrow \lambda x.\ \bot$.
    $\textsc{EvalJoin}(1)$.

    subprocedure $\textsc{EvalJoin}(k)$
        if $k > n$ then
            Let $Q(\vec{t}) \equiv \mathit{subst}(H, \phi)$.
            if $Q(\vec{t})$ is not in the primary index of $Q$
                    or if $Q$ does not have a primary index then
                Insert $Q(\vec{t})$ into $Q^{new}$.
                Insert $Q(\vec{t})$ into the primary index and all secondary indices of $Q$.
        else
            for fact $f \in \textsc{Scan}(\mathit{subst}(I_k, \phi))$ do
                Let $\theta$ be a substitution of variables in $I_k$ such that
                        the arguments of $f$ and $\mathit{subst}(I_k, \theta)$ unify.
                $\phi \leftarrow \lambda x. \begin{cases} \theta(x) & \text{if } \theta(x) \neq \bot \\ \phi(x) & \text{otherwise.} \end{cases}$
                $\textsc{EvalJoin}(k + 1)$.
```

Figure 5.2: Nested index iteration.

interpreter design, is beneficial performance-wise because accessing certain indices requires very few machine instructions: for example scanning a `value` index element is simply a load from memory.

5.9 Inline Predicates

This section gives a detailed description of a feature of DLE called *inline predicates*, which allows efficient evaluation of rules that use complex structured objects, such as the transformer string abstraction.

The following is a contrived example that computes a relation of pairs of people with the same last name, that illustrates the use of inline predicates:

```
data person = person string string.

predicate same_last_name(person,person).
index same_last_name hash_set(100).

predicate people(person).
index people hash_set(100).

same_last_name(person(A,X),person(B,X))
    :- people(person(A,X)),people(person(B,X)).
```

The evaluation of the `same_last_name` performs a Cartesian product of all tuples in the `people` relation: regardless of whether `person(A,X)` or `person(B,X)` is scanned first, the complex term is only partially bound (only `X` is bound) for the scan of the second relation.

A simple solution to obtain a more efficient join is to change the `people` relation to be a binary relation of strings:

```
predicate people_s(string,string).
index people hash_map([2],100,hash_set(100)).
```

```
same_last_name(person(A,X),person(B,X))
    :- people_s(A,X),people_s(B,X).
```

An efficient join is possible using the second attributes of the `people_s` literals. Inline predicates are a feature that enables DLE to perform this transformation implicitly. An inline predicate is declared by appending the keyword `inline` to a predicate declaration. Rules that derive an inline predicate are called *inline rules*, and the body of an inline rule is substituted into every occurrence of the head of the inline rule in a body of a rule, including, recursively in other inline rules.

For example, the following program has the same join of the `person` relations as the program above:

```
predicate people(person) inline.

predicate people_s(string,string).
index people_s hash_map([2],100,hash_set(100)).
person(person(A,X)) :- person_s(A,X).

same_last_name(person(A,X),person(B,X))
    :- people(person(A,X)),people(person(B,X)).
```

The next subsection contains a larger example that presents a more motivating use of the feature.

5.9.1 Field-sensitive C Analysis

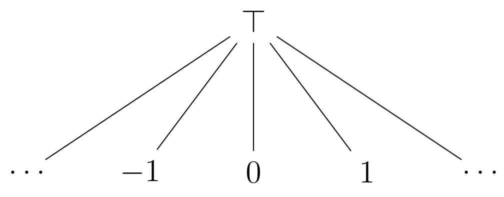
Consider a static analysis of an intermediate representation of C code. In Java, values stored in an instance field through a specific field signature can only be loaded through the same field signature. Field-sensitive analysis of C is more difficult in comparison, because fields may be assigned to, not just through the member access operator and a field name (e.g., "`x.f`"), but also through pointer arithmetic (e.g., "`&x+offsetof(struct X,f)`"). Pearce *et al.* detailed a field-sensitive analysis for C that used type information [25]. In contrast, we assume no type information is present in our analysis, which makes it more suitable for analysis of a lower-level representation of code.

Statements are canonicalized into one of three forms: address-of statements "`x = &y;`", store statements "`*x = y+z;`", and load statements "`x = *y;`". The following program computes the points-to relation of a program described by the EDB predicates `addrof`, `store`, and `load`:

```
// EDB
predicate addrof(var,var).     // addrof(X,Y): X = &Y;
predicate store(var,var,int). // store(X,Y,Z): *X = Y+Z;
predicate load(var,var).       // load(X,Y): X = *Y;
index addrof value_list(100) secondary.
index store array_map(1,100,value_list(5)) secondary.
index store array_map(2,100,value_list(5)) secondary.
index load array_map(1,100,value_list(5)) secondary.
index load array_map(2,100,value_list(5)) secondary.
```

We identify specific indices by superscripts that specify the order in which indices are declared: for example $\texttt{load}^1$ denotes the `array_map(1,100,value_list(5))` index and $\texttt{load}^2$ denotes the `array_map(2,100,value_list(5))` index.

Values are stored at memory locations that are described by an integer *offset* within an object. Theoretically, there is an infinite number of offsets. An important concept in static analysis is the formulation of lattice-based *abstractions* that model a problem domain through approximations [9]. An abstraction is defined by a join lattice (the *abstraction domain*) and a *concretization* function that defines which elements in the *concrete (problem) domain* are abstracted by elements in the abstraction domain. Our concrete domain is $\mathbb{Z}$, the conceptually infinite number of possible offsets within objects that a C program may store to and load from. Let our abstraction domain be $\mathbb{Z} \cup \{\top\}$ with a partial order defined by the following Hasse diagram (this is the same abstraction domain used by the *constant propagation* analysis [43]):



The concretization function $\gamma : \mathbb{Z} \cup \{\top\} \to \mathcal{P}(\mathbb{Z})$ is defined as follows:

$$\gamma(x) = \begin{cases} \mathbb{Z} & \text{if } x = \top \\ \{x\} & \text{if } x \in \mathbb{Z} \end{cases}$$

The abstraction domain is expressed as a type called `aoff` for *abstract offset*. The abstract offset type and the points-to relation of the analysis is declared as follows:

```
type aoff = top | const int.

predicate pts(var,aoff,var,aoff).
index pts array_map(1,100,array_map(2,10,hash_set(100))).
index pts array_map(3,100,array_map(4,10,value_list(100))) secondary.
```

A fact `pts`(x, f, h, g) indicates that the memory location with address `&`$x + f'$ may point to an address `&`$h + g'$ where $f' \in \gamma(f)$ and $g' \in \gamma(g)$.

The rule that handles address-of statements is trivial:

```
pts(X,0,Y,0)      :- addrof(X,Y).
```

The "store" statement performs arithmetic on offsets. Addition in the abstraction domain, denoted $\oplus$, can be defined as a homomorphism of addition in the concrete domain.

However, in order to ensure termination of the analysis, we use a *widened* operation that models the concrete domain more loosely:

$$\begin{aligned}
&\top \oplus y = \top. \\
&x \oplus \top = \top. \\
&x \oplus y = \top && \text{if } x, y \neq \top \wedge x \neq 0 \wedge y \neq 0. \\
&x \oplus y = x + y && \text{if } x, y \neq \top \wedge (x = 0 \vee y = 0).
\end{aligned}$$

Widened addition is defined as a function in DLE using pattern matching:

```
function add : aoff, aoff -> aoff.
add(top,_) = top.
add(_,top) = top.
add(const(X),const(Y)) = if X == 0 then Y
                         else if Y == 0 then X
                         else top.
```

A trivial property of $\oplus$ is that its result is either one of its operands, or $\top$. Thus, an analysis that uses $\oplus$ only derives facts containing either $\top$ or a `const`(X) term where X appears in a fact in an EDB relation, and thus termination is guaranteed.

Using the widened addition operator, the rule that models store statements is as follows:

```
pts(G,Gf,H,add(Hf,Z)) :- store(X,Y,Z), pts(X,0,G,Gf), pts(Y,0,H,Hf).
```

The rule to handle a load statement "x `= *`y`;`" must derive facts `pts`$(x, 0, z, g)$ if y points to an offset f within an object b, and the offset f within b points to offset g within object z. Reasoning with abstract offsets, if y points to an abstract offset f within b, then it may load an abstract offset f' within b if $\gamma(f) \cap \gamma(f') \neq \emptyset$. We define a new function to model this behaviour: Let `overlap`(X, Y) be true if $\gamma(X) \cap \gamma(Y) \neq \emptyset$. `overlap` can be defined as a Boolean-valued function in DLE:

```
function overlap : aoff, aoff -> bool.
overlap(top,_) = true.
overlap(_,top) = true.
overlap(const(A),const(B)) = A == B.
```

Boolean-valued terms are permitted in place of atoms, and thus loads from memory can be modelled soundly by a single rule:

```
pts(X,0,H,Hf) :- load(X,Y), pts(Y,0,G,Gf), pts(G,Gf2,H,Hf),
                 overlap(Gf,Gf2).
```

User-defined functions that appear in atoms can only be evaluated after all variables that appear in them are bound. Consider the first delta rule of the rule. The following joins are performed before evaluating the function `overlap`:

$$[\mathtt{pts}^{\Delta}(\mathtt{Y},\mathtt{0},\mathtt{G},\mathtt{Gf}),\mathtt{load}^{2}(\mathtt{X},\mathtt{Y}),\mathtt{pts}^{1}(\mathtt{G},\mathtt{Gf2},\mathtt{H},\mathtt{Hf})]$$

Note that the join of $\mathtt{pts}^1$ is performed over one attribute because `G` is the only bound term when evaluating the join.

A more efficient program replaces the function `overlap` with an inline predicate:

```
predicate overlap(aoff,aoff) inline.
overlap(top,_).
overlap(_,top).
overlap(const(A),const(A)).
```

Note that these clauses are not permitted in pure Datalog because variables (`A` and the anonymous variables) are not bound by the (empty) body of the clauses. Since `overlap` is an inline predicate, DLE permits such clauses if all variables are bound after inline predicates have been inlined.

Before a join order is determined, DLE inlines the `overlap` predicate and performs first-order term unification. Then, a join order is determined for the following expanded rules:

```
pts(X,O,H,Hf) :- load(X,Y), pts(Y,O,G,top()), pts(G,_,H,Hf).
pts(X,O,H,Hf) :- load(X,Y), pts(Y,O,G,_), pts(G,top(),H,Hf).
pts(X,O,H,Hf) :- load(X,Y), pts(Y,O,G,const(X)), pts(G,const(X),H,Hf).
```

The join between the `pts` relations in the third expanded rule is performed over two attributes, which is more efficient than performing the join over one attribute.

In this simplistic example, the inlining of the `overlap` predicate resulted in one rule becoming three rules. The increase in the number of rules is modest because `overlap` is a Boolean-valued function with a trivial declarative definition. The next section applies the technique above to pointer analysis using the explicit string and transformer string abstractions, where the number of rules increases by an order of magnitude.

5.10 Context Transformation Analysis

This section describes the transformation of the deduction rules presented in Figure 3.3 into plain Datalog rules under different instantiations. For explanatory purposes, the transformation of the Store and Field rules in particular are described in detail because these rules are the most expensive to evaluate in a pointer analysis.

Functions $\mathit{merge}^{\square\square}$, and $\mathit{merge\_s}^{\square\square}$ are transformed into *function-style predicates* `merge` and `merge_s`. Given a n-ary function f, its expression as a function-style predicate P is defined as $P(t_1, \ldots, t_n, r) \iff f(t_1, \ldots, t_n) = r$.

The symbols $\mathsf{comp}^{\square}$, $\mathit{inv}^{\square}$, $\mathsf{any}^{\square}$, $\mathit{merge}^{\square\square}$, and $\mathit{merge\_s}^{\square\square}$ are declared as inline predicates `compose`, `compose_inv`, `any`, `merge`, `merge_s`. Since the function $\mathit{inv}^{\square}$ always appears together with predicate $\mathsf{comp}^{\square}$, they are folded into one predicate: let $\mathtt{compose\_inv}(A, B, C)$ have the same meaning as $\mathsf{comp}^{\square}(A, \mathit{inv}(B), C)$. The predicates $\mathsf{pts}^{\square}$, $\mathsf{hpts}^{\square}$, $\mathsf{call}^{\square}$, and $\mathsf{reach}^{\square}$ are also declared as inline predicates. Different instantiations are obtained by providing different sets of inline rules for these predicates. The inline rules are presented in the following subsections.

The domains of context abstractions have different definitions depending on the instantiation, and are given definitions in the following subsections: let `context_hm` and `context_hh` represent the types of context transformation domains $\mathbf{Ctxt}^{\square}_{h,m}$ and $\mathbf{Ctxt}^{\square}_{h,h}$, respectively.

Although the semantics of DLE is purely declarative, rules require refactoring for efficient evaluation. There are numerous data structure designs for pointer analysis [18], and the scheme that we use is the same as the one used by Doop and Soot, which is to refactor the Store and Field rules in Figure 3.3 using an additional predicate, "`hload`".

The following is a partial listing of our pointer analysis implemented in DLE that is relevant to the explanations in the subsections that follow:

```
predicate pts(variable,heap,context_hm) inline.
predicate hpts(heap,field_sig,heap,context_hh) inline.
predicate hload(heap,field_sig,variable,context_hm) inline.
...
predicate compose(context_hh,context_hm,context_hh) inline.
predicate compose_inv(context_hm,context_hm,context_hh) inline.
...
hpts(G,F,H,A) :- // STORE
    pts(W,H,B),
    store(W,F,Z,_),
    pts(Z,G,C),
    compose_inv(B,C,A).

pts(Y,H,A) :- // FIELD
    hpts(W,F,H,B),
    hload(W,F,Y,C),
    compose(B,C,A).

hload(G,F,Y,A) :- // LOAD
    pts(W,G,A),
    load(W,F,Y,_).
```

5.10.1 Explicit String Instantiation

Under an explicit string instantiation, the context transformation attribute within predicates is flattened into two attributes. The two attributes represent input-output value pairs of context transformations. Let type `ctxt` be the type of elemental contexts in a particular flavour of context sensitivity: `invoke` for call-site sensitivity, `heap` for object sensitivity, and `class` for type sensitivity.

The data type declarations of `context_hm` and `context_hh` are presented below. Values of type `context_hm` consist of pairs consisting of a string of length h and a string of length m, and values of type `context_hh` consist of pairs consisting of strings of length h. Strings shorter than the truncation lengths are padded with dummy elements.

```
data hctxt = hctxt ctxt... (h times).
data mctxt = mctxt ctxt... (m times).

data context_hm = context_hm hctxt mctxt.
data context_hh = context hh hctxt hctxt.
```

The following inline rules for `pts`, `hpts`, and `hload` unwrap complex terms `context_hm` and `context_hh` into two separate terms and replace the predicates with `pts_c`, `hpts_c`, and `hload_c`, respectively.

```
predicate pts_c(variable, heap, hctxt, mctxt).
predicate hpts_c(heap, field_sig, heap, hctxt, hctxt).
predicate hload_c(invoke, method, mctxt, mctxt).
...
pts(Y,H,context_hm(A,B)) :- pts_c(Y,H,A,B).
hpts(G,F,H,context_hh(A,B)) :- hpts_c(G,F,H,A,B).
hload(G,F,Y,context_hm(A,B)) :- hload_c(G,F,Y,A,B).
```

The order of attributes in `pts_c`, `hpts_c`, and `hload_c` 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 `pts_c`(Y, H, U, V), V is a method context of Y, and U is a heap context for H.

The explicit string representation enumerates the input-output pairs of context transformations, and thus function composition is defined simply as follows:

```
compose(context_hh(H1,H2),context_hm(H2,M1),context_hm(H1,M1)).
compose_inv(context_hm(H1,M1),context_hm(H2,M1),context_hh(H1,H2)).
```

Consider the STORE, FIELD, and LOAD rules from the previous section after the `compose` and `compose_inv` predicates are inlined into their bodies and term unification is performed:

```
hpts(G,F,H,context_hh(A1,A2)) :- // STORE
    pts(W,H,context_hm(A1,B)),
    store(W,F,Z,_),
    pts(Z,G,context_hm(A2,B)).

pts(Y,H,A,B) :- // FIELD
    hpts(W,F,H,context_hh(A,AB)),
    hload(W,F,Y,context_hm(AB,B)).

hload_c(G,F,Y,A,B) :- // LOAD
    pts(W,G,context_hm(A,B)),
    load(W,F,Y,_).
```

Inlining the rules for `pts`, `hpts`, and `hload` results in the following rules:

```
hpts_c(G,F,H,A1,A2) :- // STORE
    pts_c(W,H,A1,B),
    store(W,F,Z,_),
    pts_c(Z,G,A2,B).

pts_c(Y,H,A,B) :- // FIELD
    hpts_c(W,F,H,A,AB),
    hload_c(W,F,Y,AB,B).

hload_c(G,F,Y,A,B) :- // LOAD
    pts_c(W,G,A,B),
    load(W,F,Y,_).
```

In the FIELD rule, an obvious indexing scheme for efficiently evaluating the join between the `hpts_c` and `hload_c` is to build indices on their shared variables: the first, second, and fifth attributes of `hpts_c` and the first, second, and fourth attributes of `hload_c`.

5.10.2 Transformer string Instantiation

We represent transformer strings in DLE by classifying them into *configurations*: 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. Relations specialized to a particular configuration are tagged with subscripts that characterize the 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 presence 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 following is the data type corresponding to a $\mathbf{CtxtT}^{\mathbf{t}}_{h,m}$ domain in a 1-method-1-heap analysis. The definition of type `ctxt` determines the flavour of context sensitivity:

```
data context_hm =
    context_ // epsilon
    | context_e ctxt
    | context_x ctxt
    | context_xe ctxt ctxt
    | context_w // wildcard
    | context_we ctxt
    | context_xw ctxt
    | context_xwe ctxt ctxt.
```

A naive method of implementing a transformer string instantiation is to implement the two formulas "$\mathit{match}(A \cdot B) \neq \bot$" and "$\mathit{trunc}_{i,k}(\mathit{match}(A \cdot B))$" of $\mathsf{comp}^{\mathbf{t}}$ as a function `check_match` that takes two values A and B as input, checks if $\mathit{match}(A \cdot B) \neq \bot$, and returns $trunc_{i,k}(match(A \cdot B))$. The performance of such an implementation is significantly slower than a explicit string instantiation. The reason for the lower performance can be understood by inspecting the joins performed when evaluating the FIELD rule:

```
pts_c(Y,H,match_and_trunc(A,B)) :- // FIELD
    hpts_c(W,F,H,A),
    hload_c(W,F,Y,B),
    check_match(A,B).
```

The term `check_match(A,B)` cannot be evaluated until the variables `A` and `B` are bound, thus the join between `hpts_c` and `hload_c` must be performed over two attributes instead of over three attributes in the explicit string instantiations.

A more efficient indexing scheme can be obtained by *specializing* the derived relations to every transformer string configuration. 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, `ppts_xxwe` is a subset of **Var** $\times$ **Heap** $\times$ **Ctxt** $\times$ **Ctxt** $\times$ **Ctxt**, and a fact $\mathsf{pts}(Y, H, \check{X}_1 \cdot \check{X}_2 \cdot * \cdot \widehat{E_1})$, becomes `ppts_xxwe`$(Y, H, X_1, X_2, E_1)$.

The $\mathsf{comp^t}$ 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, Figure 5.3 contains all the true clauses (all variables are universally quantified) of the $\mathsf{comp^t}$ predicate in an $m = 1$ and $h = 1$ instantiation.

Each rule is duplicated for every possible replacement of inline predicates with specialized relations. Figure 5.4 contains the declaration and inline rules for the specialized relations of pts, and the first four clauses of the $\mathsf{comp^t}$ predicate from Figure 5.3. After all inline predicates have been replaced by their specializations, complex terms do not appear in the resulting rules. For example, one such transformed rule is as follows:

```
pts_xe(Y,H,X,E) :- // FIELD
    hpts_xe(W,F,H,X,M),
    hload_xe(W,F,Y,M,E).
```

$\mathsf{comp}^{\mathsf{t}}(\epsilon, \epsilon, \epsilon).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}, \epsilon, \check{X}).$	$\mathsf{comp}^{\mathsf{t}}(*, \epsilon, *).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}*, \epsilon, \check{X}*).$
$\mathsf{comp}^{\mathsf{t}}(\epsilon, \widehat{E}, \widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}, \widehat{E}, \check{X}\widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(*, \widehat{E}, *\widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}*, \widehat{E}, \check{X}*\widehat{E}).$
$\mathsf{comp}^{\mathsf{t}}(\epsilon, \check{X}, \check{X}).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}, \check{Z}, \check{X}*).$	$\mathsf{comp}^{\mathsf{t}}(*, \check{X}, *).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}*, \check{Z}, \check{X}*).$
$\mathsf{comp}^{\mathsf{t}}(\epsilon, \check{X}\widehat{E}, \check{X}\widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}, \check{Z}\widehat{E}, \check{X}*\widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(*, \check{X}\widehat{E}, *\widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}*, \check{Z}\widehat{E}, \check{X}*\widehat{E}).$
$\mathsf{comp}^{\mathsf{t}}(\widehat{E}, \epsilon, \widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}\widehat{E}, \epsilon, \check{X}\widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(*\widehat{E}, \epsilon, *\widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}*\widehat{E}, \epsilon, \check{X}*\widehat{E}).$
$\mathsf{comp}^{\mathsf{t}}(\widehat{Z}, \widehat{E}, *\widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}\widehat{Z}, \widehat{E}, \check{X}*\widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(*\widehat{Z}, \widehat{E}, *\widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}*\widehat{Z}, \widehat{E}, \check{X}*\widehat{E}).$
$\mathsf{comp}^{\mathsf{t}}(\widehat{M}, \widecheck{M}, \epsilon).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}\widehat{M}, \widecheck{M}, \check{X}).$	$\mathsf{comp}^{\mathsf{t}}(*\widehat{M}, \widecheck{M}, *).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}*\widehat{M}, \widecheck{M}, \check{X}*).$
$\mathsf{comp}^{\mathsf{t}}(\widehat{M}, \widecheck{M}\widehat{E}, \widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}\widehat{M}, \widecheck{M}\widehat{E}, \check{X}\widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(*\widehat{M}, \widecheck{M}\widehat{E}, *\widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}*\widehat{M}, \widecheck{M}\widehat{E}, \check{X}*\widehat{E}).$

$\mathsf{comp}^{\mathsf{t}}(\epsilon, *, *).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}, *, \check{X}*).$	$\mathsf{comp}^{\mathsf{t}}(*, *, *).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}*, *, \check{X}*).$
$\mathsf{comp}^{\mathsf{t}}(\epsilon, *\widehat{E}, *\widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}, *\widehat{E}, \check{X}*\widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(*, *\widehat{E}, *\widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}*, *\widehat{E}, \check{X}*\widehat{E}).$
$\mathsf{comp}^{\mathsf{t}}(\epsilon, *\check{X}, *\check{X}).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}, *\check{Z}, \check{X}*).$	$\mathsf{comp}^{\mathsf{t}}(*, *\check{X}, *).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}*, *\check{Z}, \check{X}*).$
$\mathsf{comp}^{\mathsf{t}}(\epsilon, \check{X}*\widehat{E}, \check{X}*\widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}, \check{Z}*\widehat{E}, \check{X}*\widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(*, \check{X}*\widehat{E}, *\widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}*, \check{Z}*\widehat{E}, \check{X}*\widehat{E}).$
$\mathsf{comp}^{\mathsf{t}}(\widehat{E}, *, *).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}\widehat{E}, *, \check{X}*).$	$\mathsf{comp}^{\mathsf{t}}(*\widehat{E}, *, *).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}*\widehat{E}, *, \check{X}*).$
$\mathsf{comp}^{\mathsf{t}}(\widehat{Z}, *\widehat{E}, *\widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}\widehat{Z}, *\widehat{E}, \check{X}*\widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(*\widehat{Z}, *\widehat{E}, *\widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}*\widehat{Z}, *\widehat{E}, \check{X}*\widehat{E}).$
$\mathsf{comp}^{\mathsf{t}}(\widehat{M}, \widecheck{M}*, *).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}\widehat{M}, \widecheck{M}*, \check{X}*).$	$\mathsf{comp}^{\mathsf{t}}(*\widehat{M}, \widecheck{M}*, *).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}*\widehat{M}, \widecheck{M}*, \check{X}*).$
$\mathsf{comp}^{\mathsf{t}}(\widehat{M}, \widecheck{M}*\widehat{E}, *\widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}\widehat{M}, \widecheck{M}*\widehat{E}, \check{X}*\widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(*\widehat{M}, \widecheck{M}*\widehat{E}, *\widehat{E}).$	$\mathsf{comp}^{\mathsf{t}}(\check{X}*\widehat{M}, \widecheck{M}*\widehat{E}, \check{X}*\widehat{E}).$

Figure 5.3: Declarative definition of the $\mathsf{comp}^{\mathsf{t}}$ predicate with $m = 1$ and $h = 1$ levels of context sensitivity.

```
predicate pts_(variable,heap).
predicate pts_e(variable,heap,ctxt).
predicate pts_x(variable,heap,ctxt).
predicate pts_xe(variable,heap,ctxt,ctxt).
predicate pts_w(variable,heap).
predicate pts_we(variable,heap,ctxt).
predicate pts_xw(variable,heap,ctxt).
predicate pts_xwe(variable,heap,ctxt,ctxt).
...
pts(Y,H,context_()) :- pts_(Y,H).
pts(Y,H,context_e(E)) :- pts_e(Y,H,E).
pts(Y,H,context_x(X)) :- pts_x(Y,H,X).
pts(Y,H,context_xe(X,E)) :- pts_xe(Y,H,X,E).
pts(Y,H,context_w()) :- pts_w(Y,H).
pts(Y,H,context_we(E)) :- pts_we(Y,H,E).
pts(Y,H,context_xw(X)) :- pts_xw(Y,H,X).
pts(Y,H,context_xwe(X,E)) :- pts_xwe(Y,H,X,E).
...
compose(context_(),context_(),context_()).
compose(context_(),context_e(E),context_e(E)).
compose(context_(),context_x(X),context_x(X)).
compose(context_(),context_xe(X,E),context_xe(X,E)).
...
```

Figure 5.4: Type declaration, specialized pts predicates, and the first four clauses of the $\mathsf{comp}^{\mathsf{t}}$ predicate from Figure 5.3.

The join of `hpts_xe` and `hload_xe` is performed over three common attributes (`W`, `F`, and `M`), attaining the same indexing efficiency as the explicit string instantiation. Section 6.4 evaluates the difference in efficiency between an analysis that uses the technique described above, and an analysis that implements the comp predicate as a function and does not specialize relations to every transformer string configuration.

The functions $\mathit{any}^{\mathbf{t}}$, $\mathit{record}^{\mathbf{t}}$, $\mathit{merge}^{\mathbf{t}\Box}$, and $\mathit{merge\_s}^{\mathbf{t}\Box}$ are inlined into rules using the same method as the inlining of the "$\mathsf{comp}^{\mathbf{t}}$" predicate. The inline rules (e.g., the $\mathsf{comp}^{\mathbf{t}}$ clauses in Figure 5.3) for a particular instantiation are generated by an external tool that outputs a DLE program fragment containing the rules. This program fragment is concatenated with the DLE program that implements the deduction rules in Figure 3.3 to form a complete program for a particular instantiation.

5.10.3 Configuration Reduction

DLE uses LLVM to generate a custom piece of code for every rule, which is problematic because the transformer string instantiation generates a sizable number of rules. For a 2-method-1-heap transformer string instantiation, code generation with code optimizations takes 10 minutes, but with the following *configuration reduction* scheme, code generation time is reduced to 2 minutes.

The number of configurations of transformer strings can be reduced while still preserving the precision property established in Chapter 4. Given a transformer string $A \equiv \overset{\smile}{A_{\mathbf{x}}} \cdot A_{\mathbf{w}} \cdot \widehat{A_{\mathbf{e}}}$ in $\mathbf{CtxtT}^{\mathbf{t}}_{i,j}$, we say A is *bottomed-out* if $\|A_{\mathbf{x}}\| = i$ or $\|A_{\mathbf{e}}\| = j$. We can observe that the concretization of transformer strings remains the same if the wildcard letter is always added to bottomed-out transformer strings:

$$\mathit{simplify}(\overset{\smile}{A_{\mathbf{x}}} \cdot A_{\mathbf{w}} \cdot \widehat{A_{\mathbf{e}}}) \equiv \begin{cases} \overset{\smile}{A_{\mathbf{x}}} \cdot A_{\mathbf{w}} \cdot \widehat{A_{\mathbf{e}}} & \text{if } \|A_{\mathbf{x}}\| < i \wedge \|A_{\mathbf{e}}\| < j \\ \overset{\smile}{A_{\mathbf{x}}} \cdot * \cdot \widehat{A_{\mathbf{e}}} & \text{otherwise} \end{cases}$$

For all A, $\gamma_{i,j}(A) = \gamma_{i,j}(\mathit{simplify}(A))$ ($\gamma_{i,j}$ is defined in Section 4.4.2). Thus, the theorems about the relative precision of the transformer string and the explicit string abstractions still hold even when *simplify* is applied to the output of all transformer string operations: that is, the transformer string abstraction is as precise as the explicit string abstraction when truncated to the same levels of context sensitivity. In theory, applying the *simplify* function may reduce the precision of the transformer string abstraction, but even with this reduced precision, the simplified transformer string abstraction is still more precise than the explicit string abstraction. Performing the reduction has no impact on the precision in any of the evaluated instantiations and analyzed programs in Chapter 6.

Although the reduction in the number of transformer string configurations is modest (from 12 to 8 for $\mathbf{CtxtT}^{\mathbf{t}}_{2,1}$), the reduction in the number of instantiated rules is substantial (from 4031 to 1669 in an instantiation with $m = 2$ and $h = 1$ levels of context sensitivity).

Chapter 6

Evaluation

The experimental evaluation compares the transformer string instantiation of the pointer analysis described so far with the traditional explicit string instantiation. The analyses are compared in call-site-, object-, and type-sensitive configurations.

6.1 Experimental Setup

The analyzed programs are from the DaCapo benchmark suite (v.2006-10-MR2) under JDK 1.6.0_43 [4]. We use the same *fact generator* as Doop [5], which transforms Java bytecode to a set of relations using the Soot [42] framework. Table 6.1 presents size metrics of the benchmark programs that are relevant to pointer analysis. The first column contains the names of analyzed programs. The next three columns contain the number of initialized classes, reachable methods, and call-graph edges, computed by a context-insensitive analysis. The next two columns contain the number of variables and allocation sites in reachable methods. The last two columns contain the number of loads and stores (accesses of instance fields, array indices, and static fields) in reachable methods. `jython` and `hsqldb` are not evaluated because context-sensitive analyses of the two programs do not scale due to overly

Name	Initialized	Methods	CG edges	Variables	Allocations	Loads	Stores
antlr	1391	8605	54466	77260	17728	13985	6549
bloat	1582	10149	72842	89923	18197	16037	7066
chart	2463	15878	87126	134195	31434	24523	15971
eclipse	1570	9425	56011	80452	17451	13862	6833
luindex	1356	7882	43391	64655	14613	11292	6131
pmd	1551	9317	50543	75699	16232	12998	6970
xalan	1566	8992	49607	73084	16267	12326	6886

Table 6.1: Benchmark metrics collected by a context-insensitive analysis.

conservative handling of Java reflections. `lusearch` is not evaluated because it is too similar to `luindex`.

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).

The experiments were performed on an Intel i7-2600K processor with 16GiB of RAM. DLE is single-threaded.

6.2 Analysis Precision

Table 6.2 presents relation sizes of *context-insensitive* projections of different flavours and levels of context sensitivity. These numbers highlight the precision differences of the different instantiations.

Although transformer strings are theoretically more precise than explicit 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.

		1-call	1-call+H	1-object	2-object+H	2-type+H (**c**)	2-type+H (**t**)
antlr	reach	8436	8436	8391	8161	8189	8189
	pts	2.04M	2.03M	1.69M	0.441M	1.031M	**1.035M**
	hpts	258k	257k	153k	67.4k	87.3k	**88.6k**
	call	52223	52223	51336	46655	47023	47023
bloat	reach	9964	9964	9919	9646	9714	9714
	pts	4.08M	4.08M	3.82M	1.15M	1.43M	**1.43M**
	hpts	457k	457k	413k	207k	261k	**264k**
	call	69646	69646	68520	59752	61203	61203
chart	reach	15474	15474	15705	12222	13339	**13344**
	pts	6.11M	6.11M	5.88M	0.539M	0.882M	**0.886M**
	hpts	357k	356k	282k	76.5k	143k	**147k**
	call	81719	81719	82930	58073	62843	**62856**
eclipse	reach	9125	9125	9073	8769	8807	8807
	pts	1.86M	1.85M	1.60M	0.439M	0.625M	**0.633M**
	hpts	182k	181k	116k	68.8k	99.2k	**102k**
	call	52162	52157	50905	43515	44074	44074
luindex	reach	7713	7713	7666	7434	7462	7462
	pts	1.20M	1.19M	1.01M	0.285M	0.383M	**0.386M**
	hpts	113k	113k	75.4k	49.5k	63.9k	**65.2k**
	call	41164	41164	40193	35615	36012	**36012**
pmd	reach	9147	9147	9095	8834	8865	8865
	pts	1.56M	1.55M	1.35M	0.341M	0.460M	**0.463M**
	hpts	138k	138k	96.1k	63.7k	80.1k	**81.5k**
	call	48285	48285	47353	42117	42536	42536
xalan	reach	8810	8810	8770	8553	8566	8566
	pts	1.81M	1.80M	1.56M	0.392M	0.530M	**0.533M**
	hpts	227k	226k	168k	115k	156k	**158k**
	call	47003	47003	46044	41035	41536	41536

Table 6.2: Sizes of context-insensitive relations of varying flavours and levels of context sensitivity.

Under type-sensitive analysis (column 2-type+H), the transformer string abstraction is less precise, and larger relations are highlighted in bold typeface. The decrease in precision when type-sensitive analysis is performed using the transformer string abstraction is marginal: an average 1% and 2% increase 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-graph edges.

6.3 Analysis Efficiency

Table 6.3 presents the efficiency difference between the transformer string and explicit string instantiations in terms of the analysis time and sizes of relations. The first numbers in each column state the sizes of the context-sensitive $\mathsf{pts}^{\mathsf{e}}$, $\mathsf{hpts}^{\mathsf{e}}$, and $\mathsf{call}^{\mathsf{e}}$ relations, the sum of the sizes of the three relations, and the analysis time, using the explicit 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, as compared to the explicit string abstraction.

No reduction in the size of the hpts relation is present under 1-call and 1-object configurations because the relation is context-insensitive (no heap contexts) and the two abstractions empirically have the same precision.

In the instantiations where transformer strings are as precise as explicit 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

		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%
	hpts	279k	—	2349k	32.0%	170k	—	368k	18.9%	206k	5.4%
	call	377k	15.6%	377k	15.5%	1885k	9.2%	4402k	25.4%	542k	27.8%
	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%
	hpts	475k	—	11802k	13.4%	429k	—	4028k	1.8%	526k	-43.9%
	call	559k	16.5%	559k	16.5%	2791k	6.0%	39212k	3.7%	1078k	7.4%
	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%
	hpts	419k	—	4235k	44.4%	345k	—	721k	42.3%	431k	4.0%
	call	541k	17.4%	541k	17.4%	5094k	7.9%	15520k	49.5%	1379k	35.5%
	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%
	hpts	205k	—	3722k	38.3%	136k	—	806k	28.3%	731k	5.3%
	call	433k	16.7%	433k	16.7%	1579k	9.2%	9757k	27.0%	2564k	14.3%
	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%
	hpts	125k	—	1219k	34.8%	86k	—	248k	26.0%	179k	8.0%
	call	330k	14.4%	330k	14.4%	880k	10.7%	2711k	26.1%	527k	29.2%
	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%
	hpts	151k	—	1499k	33.5%	108k	—	443k	15.9%	298k	5.1%
	call	363k	14.4%	363k	14.4%	1117k	8.7%	3309k	23.6%	580k	27.5%
	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%
	hpts	243k	—	2176k	36.2%	183k	—	6053k	4.7%	336k	5.9%
	call	364k	14.3%	364k	14.3%	1866k	8.1%	49297k	30.4%	816k	30.3%
	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%

Table 6.3: Number of context-sensitive facts and percentage decrease from using the transformer string abstraction, as compared to the explicit string abstraction.

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 explicit strings) of one is a superset of the other. An example are facts $\mathsf{pts}(X, H, *)$, $\mathsf{pts}(X, H, \breve{M_1} \cdot *)$, $\mathsf{pts}(X, H, * \cdot \breve{M_2})$, and $\mathsf{pts}(X, H, \breve{M_1} \cdot * \cdot \widehat{M_2})$. Fact $\mathsf{pts}(X, H, *)$ subsumes facts $\mathsf{pts}(X, H, A)$ for all A. Facts $\mathsf{pts}(X, H, \breve{M_1} \cdot *)$ and $\mathsf{pts}(X, H, * \cdot \widehat{M_2})$ subsume $\mathsf{pts}(X, H, \breve{M_1} \cdot * \cdot \widehat{M_2})$.

Figure 6.1 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 $\breve{\texttt{c1}} \cdot \widehat{\texttt{c1}}$ transformer string. Since all invocations of `m` have a receiver object, $\mathsf{pts}(\texttt{v}, \texttt{h1}, \breve{C} \cdot \widehat{C})$ will be inferred for all method contexts C of `m`, resulting in the same explicit enumeration of contexts as the explicit string representation. Although $\mathsf{pts}(\texttt{t}, \texttt{h2}, \epsilon)$ is just one additional fact in the transformer string representation compared to the explicit string representation, all facts that can be derived using $\mathsf{pts}(\texttt{v}, \texttt{h1}, \breve{C} \cdot \widehat{C})$ for some C can also be derived using $\mathsf{pts}(\texttt{t}, \texttt{h2}, \epsilon)$ 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 an "**xwe**" configuration. Variables pointing to n do so through data-flow paths (arising partly due to imprecision

```
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
$\mathsf{reach}(\texttt{main}, \texttt{entry})$	Entry
$\mathsf{pts}(\texttt{t}, \texttt{h2}, \epsilon)$	New
$\mathsf{call}(\texttt{c1}, \texttt{m}, \widehat{\texttt{c1}})$	Virt
$\mathsf{pts}(\texttt{this}_{\texttt{m}}, \texttt{h2}, \widehat{\texttt{c1}})$	Virt
$\mathsf{reach}(\texttt{m}, \texttt{c1})$	Reach
$\mathsf{pts}(\texttt{v}, \texttt{h1}, \epsilon)$	New
$\mathsf{hpts}(\texttt{h2}, \texttt{f}, \texttt{h1}, \widecheck{\texttt{c1}})$	Store
$\mathsf{hload}(\texttt{h2}, \texttt{f}, \texttt{v}, \widehat{\texttt{c1}})$	Load
$\mathsf{pts}(\texttt{v}, \texttt{h1}, \widecheck{\texttt{c1}} \cdot \widehat{\texttt{c1}})$	Ind

Figure 6.1: Points-to relationships from multiple data-flow paths.

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`.

6.4 Indexing Efficiency

	1-call		1-call+H		1-object		2-object+H		2-type+H	
antlr	7.3s	127.7s	33.0s	945.1s	11.1s	215.3s	12.2s	380.0s	3.3s	17.6s
bloat	18.9s	241.7s	204.0s	N/A	37.9s	1111.3s	942.2s	N/A	17.1s	172.0s
chart	25.2s	398.1s	80.9s	2587.8s	142.4s	N/A	33.1s	2210.8s	8.0s	96.2s
eclipse	6.9s	86.2s	51.3s	1949.5s	12.3s	300.9s	34.7s	1033.8s	17.8s	224.6s
luindex	4.5s	56.9s	17.6s	391.5s	6.1s	113.4s	7.5s	131.9s	2.9s	16.0s
pmd	5.9s	72.4s	22.7s	561.2s	10.5s	227.7s	9.6s	446.3s	3.3s	21.0s
xalan	6.3s	78.1s	30.4s	1139.6s	15.0s	425.0s	876.2s	N/A	4.2s	33.7s

Table 6.4: Analysis times with transformer configuration specialization of relations and without specialization.

Inlining of the declarative definition of the $\mathsf{comp}^{\mathsf{t}}$ predicate greatly increases the number of Datalog rules: the explicit string instantiations have 59 rules for all flavours and levels of context sensitivity; using the transformer string abstraction, the 1-call and 1-object instantiations have 162 rules, the 1-call+H instantiation has 566 rules, and the 2-object+H and 2-type+H instantiations have 1669 rules.

However, the analysis becomes significantly slower if inlining is not performed. Table 6.4 records the analysis times of analyses that use the predicate inlining technique described in Section 5.10.2 (first columns), and analysis times of analyses that implement the $\mathsf{comp}^{\mathsf{t}}$ predicate as a function (second columns). Entries marked as N/A are analyses that did not terminate within one hour.

6.5 Summary

The new transformer string abstraction represents the same context information as the explicit string abstraction using fewer facts: the geometric mean reductions in the numbers of facts over the seven benchmarks are 6.3%, 17.5%, 9.8%, 28.9%, and 20.1% for 1-call, 1-call+H, 1-object, 2-object+H and 2-type+H configurations, respectively. Using the techniques described in Section 5.10.2, the more efficient data representation translates to improved analysis times in general: the geometric mean reductions in analysis times are 8.7%, -0.7%, 8.8%, 27.1%, and 14.9%, respectively, for the configurations listed above. The 2-object+H configuration, which is the most precise configuration that scales to large programs, has the greatest improvement in analysis efficiency.

Chapter 7

Related Work

Our deduction rules are adapted from the rules in the Doop Framework for Java Pointer Analysis [5]. Doop supports various flavours of context sensitivity, including call-site, object, type sensitivity, and combinations thereof [16]. Doop uses the proprietary Datalog engine LogicBlox [12]. 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.

There are several cost/precision trade-offs in pointer analysis [15]: A *unification-based* (also called Steensgaard's or bi-directional analysis) models assignments with equivalence constraints (a statement "`p=q;`" also has the effects of "`q=p;`"). Points-to relations can be expressed as equivalence classes on variables, and pointer analysis can be performed in near-linear time [38]. An *inclusion-based* analysis (also called Andersen's or uni-directional analysis) models assignments using inclusion constraints [2, 11], which is the type of analysis presented in this dissertation.

Our analysis is *flow-insensitive* meaning that the analysis conservatively assumes that statements in a program may be executed in any order. This assumption allows computation of points-to facts that are conservative for all program points in a program. A simple approach to adding flow-sensitivity to a flow-insensitive pointer analysis is to add a component to the points-to relation that indicates the program point of the points-to relationship. This is described as a *dense* data-flow analysis because data-flow facts that are not affected by a particular statement are still replicated before and after every statement. A *sparse* analysis strives to represent information only at program points where it changes. Wegman and Zadeck formulated a sparse constant propagation algorithm in [43], using the Static Single Assignment (SSA) form [10] of a program. The SSA form of a program has the property that all variables are assigned exactly once. Hardekopf and Lin presented a *semi-sparse* pointer analysis for C/C++ where points-to facts of *top-level* variables (objects that are not indirectly referenced) are propagated sparsely but points-to facts of other objects (*address-taken* stack objects and heap objects that may be referenced through pointers) are propagated densely [13]. Yu *et al.* partition objects into levels, where pointers that may reference an object are at the same or higher level as the object, which allows sparse analysis of more objects than just the top-level objects. Some of the benefits of flow-sensitivity can be obtained by transforming the input program into SSA form. Kastrinis and Smaragdakis noted only a marginal improvement in precision from transforming Java programs into SSA form when analyzed using Doop [16]. We do not apply the SSA transformation to input programs in our analysis.

The two primary approaches to context-sensitive data-flow analysis are the *call-string approach* and the *functional* (or *summary-based*) approach [31]. Both aim to improve the precision of an analysis by preventing data-flow facts from one call site from propagating to another. A functional approach summarizes the effect of a procedure as a mapping from data-flow facts before a procedure invocation to data-flow facts after the invocation. Our analysis uses the call-string approach. Originally, the call-string approach was formulated to use call sites as contexts, but analysis of object-oriented languages have the benefit of using better choices of contexts: object- and type-sensitive analyses have a better performance-to-precision trade-off than call-site sensitivity [23, 19, 34].

If a data-flow analysis has *distributive* transfer functions, then the functional approach has the precision of a call-string approach with unbounded call-strings [31]. Pointer analysis (and its simpler cousin: constant-propagation analysis) has a non-distributive transfer function. The functional approach to pointer analysis is complicated by the need to abstract the unbounded number of heap objects at the start of an invocation of a procedure (the heap objects reachable by dereferencing parameters), and the unbounded number of heap objects the procedure may allocate itself. One approach to the first problem is to identify heap objects by abstract *access paths* starting from a parameter: for example, all objects pointed-to by field `f` of a parameter `p` are abstracted by an abstract object identified by the access path "`p.f`". A complication is that objects may be reachable through multiple access paths and thus potentially abstracted by multiple abstract objects. Chatterjee *et al.* address this issue in their *Relevant Context Inference* analysis by predicating points-to tuples with propositional formulas that describe the aliasing relationships of parameters [8]. Sui *et al.* use Steensgaard's analysis as a pre-analysis to determine potentially aliasing access-paths [39], and merge them into a single abstract object. Wilson and Lam detect aliasing access paths and merge abstract objects during the analysis [45]. The three analyses described above are for C/C++.

Sălcianu and Rinard presented a summary-based pointer analysis for Java that identifies *pure methods* (a method is pure if it does not mutate any object that exists before the method's invocation) and various properties of parameters (whether any objects accessible through a parameter are mutated by a method) [29]. They measure the precision of their analysis by the percentage of methods that were identified as being pure.

Sridharan and Bodík proposed a CFL-reachability-based demand-driven context-sensitive analysis for Java [36]. 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 [46]. 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 $\mathsf{pts}(Z, H, \widecheck{A_{\mathbf{x}}} \cdot \widehat{A_{\mathbf{e}}})$ of a call-site-sensitive instantiation, the base method of $\widecheck{A_{\mathbf{x}}} \cdot \widehat{A_{\mathbf{e}}}$ (defined in Section 4.4.1) 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. Our contribution is that we formally define an algebraic structure of context transformations that does not enumerate redundant context information, and show that a common set of parameterized deduction rules can be instantiated, using either the explicit string or transformer string abstractions, into efficient Datalog programs. Comparing the precision of Xu and Rountev's analysis to our analysis is difficult: they analyze the benchmark programs under JDK version 1.3, which is significantly smaller than the JDK analyzed by DOOP and our analysis. For example, they report 4451 virtual call sites in `antlr` out of which 3611 resolve to a single target using an 1-obj+H analysis. In comparison, our analysis reports 26744 reachable virtual call sites out of which 24895 resolve to a single target under context-insensitive analysis. The magnitude of the size difference makes comparative analysis between the two algorithms difficult.

Binary decision diagrams (BDDs) have been extensively studied as a technique for improving the scalability of context-sensitive pointer analysis [44, 49, 19]. 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 [5]. 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.

Analysis performance can be improved by varying the level of context sensitivity of program elements. For example, allocation sites of containers (e.g. Java's `ArrayList` class) are quintessential targets of higher levels of heap contexts, because heap contexts allow finer static differentiation of elements stored in different containers during run-time. In contrast, heap contexts are useless to allocation sites of Java's `String` class, because the class has no instance fields. Our analysis implementation is hard-coded to treat allocation sites of `String`s context-insensitively, which is standard practice in Java pointer analysis [19, 5]. Smaragdakis *et al.* presented an analysis where pointer analysis is performed twice: the first run is context-insensitive, and heuristics applied to the metrics collected from the first run determine which allocation and invocation sites are treated context-sensitively in the second run [34]. Zhang *et al.* presented an analysis where a SAT solver is used to determine whether a certain level of context sensitivity is able to resolve a pointer analysis query [47].

Tan *et al.*'s analysis uses the result of a pre-analysis to construct an *object allocation graph* [40]: 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 context 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 context strings.

A *demand-driven* algorithm attempts to improve performance by only computing pointer information that is relevant to a given query [48, 37, 36, 14, 30]. An *exhaustive* analysis expressed as a logic program can be transformed into a demand-driven analysis using the *magic sets transformation* [3, 26, 41]. The various algorithms differ in their caching scheme of partial points-to information (e.g. caching path fragments of CFL-reachability queries). Sridharan and Bodík caches $L_F \cap L_C$-paths (see Section 2.2) together with the call-strings corresponding to the beginnings and ends of the paths [36]. These pairs of call stacks have the same interpretation as pairs of strings in an explicit string abstraction.

One of the earliest uses of Datalog in static analysis was Whaley and Lam's formulation of context-sensitive analysis in a BDD-based Datalog engine BDDBDDB [44]. Zhang *et al.* [47] also utilized BDDBDDB. CHORD is Java analysis framework that supports static analyses written in either Java or in Datalog [24]. DOOP is implemented in Datalog$_{\text{LB}}$, a dialect of Datalog for the proprietary Datalog engine LogicBlox [12], which has an extensive number of features including negation, complex terms, and constraints on values. LogicBlox is freely available for research use.

Chapter 8

Conclusion

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. Our formulation of pointer analysis is a unification of the concepts used in the representation of context information in context-string-based analyses and in CFL-based analyses, and we state the theoretical precision differences of the two representations.

The ideal of representing *local flow* of pointer information in a form that is invariant with respect to the calling contexts of a method, a concept that forms the backbone of summary-based pointer analysis, is embodied by our new abstraction. The result is a new abstraction of pointer information that empirically has less redundancy than the context string abstraction. Less redundancies allow precise context-sensitive analysis to take less time and memory.

Recently, implementing static analyses in Datalog has become popular. The benefits of using Datalog include rapid prototyping, and more importantly, better reproducibility of an analysis. However, there may be aspects of an analysis that appear to be incompatible with the use of a Datalog engine. In particular, static analyses use lattices as abstractions and may require the computation of join, meet, and other operations. If implemented naively in a Datalog engine, the analysis may have an unacceptably poor performance, and without a systemic translation, implementing an efficient program by hand may be a daunting task (for example, the transformer string instantiation has thousands of rules). We have presented an idea of how static analyses that use complex abstractions can be translated into an efficient Datalog program.

A direction of future work is to evaluate the efficiency difference between the explicit 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 [3]. 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.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[2] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[3] 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.

[4] 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, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[5] 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.

[6] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, September 1992.

[7] 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.

[8] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant context inference. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 133–146, New York, NY, USA, 1999. ACM.

[9] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[10] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35. ACM, 1989.

[11] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 242–256, New York, NY, USA, 1994. ACM.

[12] 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.

[13] Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 226–238, New York, NY, USA, 2009. ACM.

[14] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *ACM SIGPLAN Notices*, volume 36, pages 24–34. ACM, 2001.

[15] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM.

[16] 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.

[17] 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.

[18] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[19] 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.

[20] 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.

[21] 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.

[22] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[23] 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.

[24] Mayur Naik. Chord: A versatile platform for program analysis. In *Tutorial at ACM Conference on Programming Language Design and Implementation*, 2011.

[25] David J. Pearce, Paul H.J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of c. *ACM Trans. Program. Lang. Syst.*, 30(1), November 2007.

[26] Thomas Reps. Solving demand versions of interprocedural analysis problems. In *International Conference on Compiler Construction*, pages 389–403. Springer, 1994.

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

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

[29] Alexandru Sălcianu and Martin Rinard. Purity and side effect analysis for java programs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 199–215. Springer, 2005.

[30] Lei Shang, Xinwei Xie, and Jingling Xue. On-demand dynamic summary-based points-to analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 264–274. ACM, 2012.

[31] Micha Sharir and Amir Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

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

[33] Yannis Smaragdakis and Martin Bravenboer. Using datalog for fast and easy program analysis. In *Datalog Reloaded*, pages 245–251. Springer, 2011.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[39] Yulei Sui, Sen Ye, Jingling Xue, and Jie Zhang. Making context-sensitive inclusion-based pointer analysis practical for compilers using parameterised summarisation. *Softw. Pract. Exper.*, 44(12):1485–1510, December 2014.

[40] 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.

[41] K Tuncay Tekle and Yanhong A Liu. More efficient datalog queries: subsumptive tabling beats magic sets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 661–672. ACM, 2011.

[42] 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.

[43] Mark N Wegman and F Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, 1991.

[44] 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.

[45] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming*

Language Design and Implementation, PLDI '95, pages 1–12, New York, NY, USA, 1995. ACM.

[46] 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.

[47] 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.

[48] 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.

[49] 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.