Foundations and Trends® in Programming Languages
Vol. 1, No. 4 (2014) 269–381

DOI: 10.1561/2500000020

now
the essence of knowledge

A Framework For Efficient Modular Heap Analysis

Ravichandhran Madhavan
EPFL, Switzerland
ravi.kandhadai@epfl.ch

G. Ramalingam
Microsoft Research, India
grama@microsoft.com

Kapil Vaswani
Microsoft Research, India
kapilv@microsoft.com

Contents

Abstract

Modular heap analysis techniques analyze a program by computing summaries for every procedure in the program that describes its effects on an input heap, using pre-computed summaries for the called procedures. In this article, we focus on a family of modular heap analyses that summarize a procedure's heap effects using a context-independent, shape-graph-like summary that is agnostic to the aliasing in the input heap. The analyses proposed by Whaley, Salcianu and Rinard, Buss *et al.*, Lattner *et al.* and Cheng *et al.* belong to this family. These analyses are very efficient. But their complexity and the absence of a theoretical formalization and correctness proofs makes it hard to produce correct extensions and modifications of these algorithms (whether to improve precision or scalability or to compute more information). We present a modular heap analysis framework that generalizes these four analyses. We formalize our framework as an abstract interpretation and establish the correctness and termination guarantees. We formalize the four analyses as instances of the framework. The formalization explains the basic principle behind such modular analyses and simplifies the task of producing extensions and variations of such analyses.

We empirically evaluate our framework using several real-world $C^\sharp$ applications, under six different configurations for the parameters, and using three client analyses. The results show that the framework offers a wide range of analyses having different precision and scalability.

R. Madhavan, G. Ramalingam, and K. Vaswani. *A Framework For Efficient Modular Heap Analysis*. Foundations and Trends® in Programming Languages, vol. 1, no. 4, pp. 269–381, 2014.
DOI: 10.1561/2500000020.

1

Introduction

Compositional or modular analysis [Cousot and Cousot, 2002] is a key technique for scaling static analysis to large programs. Our interest is in techniques that analyze a procedure in isolation, using pre-computed summaries for called procedures, computing a summary for the analyzed procedure. Such analyses are widely used and have been found to scale well. However, computing such summaries for a heap analysis (or points-to analysis) is challenging because of the aliasing in the input heap. For example, consider the procedure `P` shown in Fig. 1.2(a). Its behaviour on two different input heaps is shown in Fig. 1.2(b) and Fig. 1.2(c). (The heaps are depicted as shape graphs. The input heap is shown at the top and the corresponding output heap at the bottom). It can be seen that the behaviour of `P` varies significantly depending on the aliasing between the variables `x` and `y` in the input heap. A sound summary for `P` should be able to approximate the behaviour of `P` in both these scenarios.

Existing modular heap analyses can be broadly classified into the following categories. (The following classification is not exhaustive. There are modular analyses such as [Nystrom et al., 2004] that cannot be easily classified into any of the categories mentioned. It is also pos-

```
   P (x, y) {
[1]   t = new ();
[2]   x.next = t;
[3]   t.next = y;
[4]   retval = y.next;
   }
```

Figure 1.1: A procedure `P` whose behaviour depends on the aliasing in the input heap.

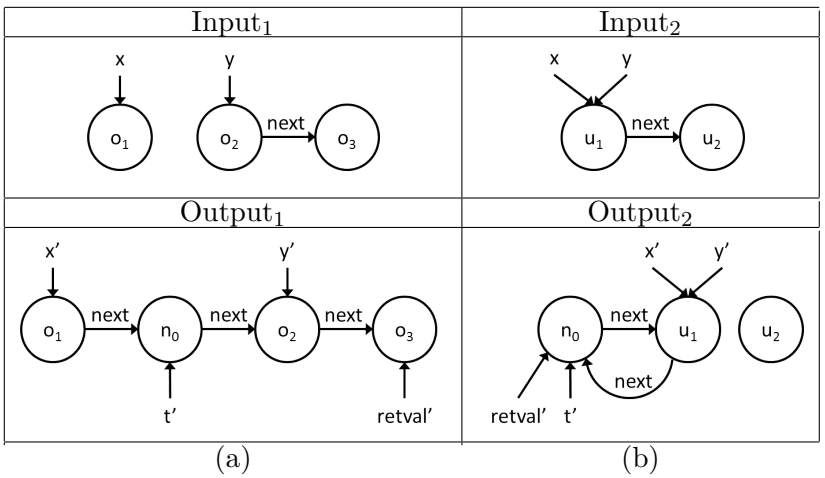


Figure 1.2: (a) Output of `P` when `x` and `y` are not aliases in the input heap. (b) Output of `P` when `x` and `y` are aliases in the input heap.

sible to design analyses that belong to more than one of the categories though we aren't aware of any.) (a) Analyses such as [Calcagno et al., 2009] compute *conditional summaries* that are applicable only in the contexts that satisfy certain conditions (e.g., aliasing or non-aliasing conditions). (b) Some analyses such as [Chatterjee et al., 1999], [Dillig et al., 2011], [Jeannet et al., 2010] enumerate all relevant configurations of the input heap belonging to a fixed abstract domain and generate summaries for each configuration. A major challenge with this approach is reducing the number of configurations that are enumerated, which can quickly become intractable, and finding efficient ways of representing them. (c) A few analyses, namely, [Whaley and Rinard, 1999], [Cheng and Hwu, 2000], [Liang and Harrold, 2001], [Lattner et al., 2007], [Buss et al., 2008] compute context-independent summaries that are agnostic to the aliasing in the input heap *without* enumerating the possible configurations of the input heap. To our knowledge, these are the only existing analyses having this property.

The analysis proposed by Whaley and Rinard [Whaley and Rinard, 1999] was later on refined and improved by Salcianu and Rinard [Salcianu and Rinard, 2005]. We will refer to this analysis as the WSR analysis. Adopting the terminology of [Lattner et al., 2007], we will refer to the analysis proposed by Lattner *et al.* as Data Structure Analysis (DSA).

In this article, we consider analyses belonging to the final category. They are interesting for several reasons. (a) They have a number of applications, discussed shortly. (b) The analyses are very efficient. DSA scales to the entire Linux kernel comprising 3 million lines of code in 3 seconds. An optimized version of WSR analysis discussed in [Madhavan et al., 2011] scales to $C^\sharp$ libraries with 250 thousand lines of code. (c) Being modular, they can analyze open programs, libraries, and, in fact, any arbitrary chunk of code without requiring any knowledge of the environment. Moreover, the summaries computed are such that they be refined incrementally when more knowledge about the environment becomes available.

These analyses have been used in a number of applications. Salcianu and Rinard present an application of their analysis to compute the

side-effects of a procedure, which are the effects of the procedure on the pre-existing state, and use it to classify procedures as *pure* (having no side-effects) or *impure* [Salcianu and Rinard, 2005]. This analysis, referred to as purity analysis, itself has a number of applications.

Whaley and Rinard applied their analysis to identify objects that can be safely allocated in the stack instead of the heap [Whaley and Rinard, 1999]. We use an extension of the WSR analysis to statically verify the correctness of the use of speculative parallelism [Prabhu et al., 2010]. Lattner *et al.* use their analysis to perform *pool allocation* in which different instances of data structures are allocated to distinct memory pools, which enables certain compiler optimizations [Lattner and Adve, 2005b].

However, the complexity of the analyses makes the task of extending and modifying these analyses challenging and time consuming. Questions such as the following often arise while designing new applications based on the analyses and there is no easy way of answering them. Can the scalability of the WSR analysis be improved at the expense of precision? Can DSA be extended to yield more precise results when more time and resources are available? Is it possible to integrate a modular static analysis that requires heap information (such as an *information flow analysis*) with these analyses as typically done in top-down whole program analyses? A sound theoretical formulation of the analyses will greatly aid in answering such questions.

Upon investigating the theoretical basis of these analyses, we realized that, in spite of the apparent dissimilarity between the analyses and the differences in the precision, scalability, and functionality, there are some fundamental ideas common to all of these analyses. This motivated us to develop a parametric framework for designing efficient modular heap analyses. The analyses listed earlier become specific instances of our framework.

We formulate our framework as a parametric abstract interpretation and establish the correctness and termination of the semantics. We present several transformations and optimizations (collectively called as specializations) of our framework and establish their correctness using the standard theory of abstraction interpretation. Our framework

with its parametric domains, parametric semantics and several correctness preserving transformations provides a convenient mechanism for obtaining modular heap analyses with different levels of precision and scalability.

We formally establish that the four analyses: [Whaley and Rinard, 1999], [Cheng and Hwu, 2000] [Lattner et al., 2007] (except for the handling of indirect calls), [Buss et al., 2008] are specific instances of our framework. We exclude the analysis proposed in [Liang and Harrold, 2001] (called as *MoPPA*) as it is very similar to [Lattner et al., 2007]. Nevertheless, it can also be expressed as an instance of our framework.

Formulating the analyses as instances of the framework has several advantages. It provides an immediate proof of correctness and termination for the analyses. It also helps understand the abstractions performed by the analyses and identify opportunities for making them more precise or scalable. In fact, we were able to identify several corner cases that were not handled by some of the algorithms and were able to fix them. Since we were unable to find complete formalization of some of the analyses, it is not clear to us if the problems we identified are bugs in the algorithm or gaps in the informal descriptions.

We implemented the framework in our open source heap analysis tool *Seal* (`seal.codeplex.com`). *Seal* is a fairly robust tool which has been used in several program analysis applications. We empirically studied the different configurations of the framework using *Seal*. We present a summary of the results in Chapter 8. The results throw light on the importance of the parameters of the framework by measuring their impact on the precision and scalability of three client analyses.

The framework presented in this article has some limitations. Most importantly, it does not support strong updates on heap locations and path-sensitivity. To our knowledge, all existing modular heap analysis approaches (such as [Dillig et al., 2011], [Jeannet et al., 2010]) that perform strong updates on heap locations enumerate the possible configurations of the input heap. Nevertheless, we believe that both these challenges can be addressed without resorting to enumeration of the input heap configurations. We briefly outline a potential approach in the Future Works section (see Chapter 9).

The following are the main contributions of this article:

- We propose a modular heap analysis framework that is a generalization of a family of existing modular heap analyses. To our knowledge, this is the first attempt to connect and develop a common theory for the different modular heap analyses proposed in the past.
- We formulate our framework as an abstract interpretation and prove the correctness and termination properties.
- We present several correctness preserving transformations that are applicable to all instances of the framework.
- We formalize four existing modular heap analyses as abstractions of instances of our framework, thereby provide a proof of correctness and termination for the analyses. The formalization exposes the relationships between the analyses and provides ways of improving and modifying them.
- We present an empirical evaluation of the framework by analyzing ten open source $C^\sharp$ applications with six different configurations of the framework. We used three client analyses, namely, *Purity and Side Effects Analysis*, *Escape Analysis* and *Call-graph Analysis* to measure the precision and scalability of each of the six configurations.

2

An Informal Overview

The distinguishing aspect of the analyses belonging to our framework is the use of a graph based representation of state transformations. We illustrate this using the example shown in Fig. 2.1(a) which was introduced in the introduction.

The *transformer graph* τ shown in Fig. 2.1(b) summarizes the heap-effect of the procedure `P`. We omit the *null* node from the figures to keep them simple. Vertices in a transformer graph are of two types: *internal* (shown as circles with a solid outline) and *external* nodes (shown as circles with a dashed outline). Internal nodes represent new heap objects created during the execution of the procedure. E.g., vertex n_0 is

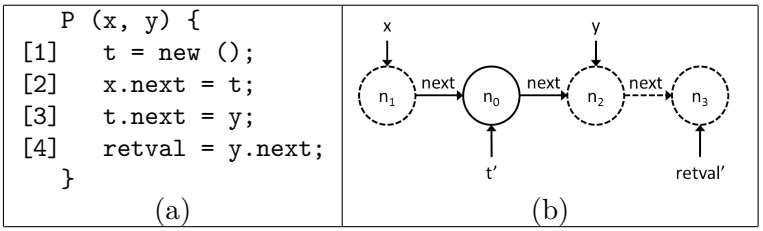


Figure 2.1: (a) A procedure P. (b) Summary graph τ of the procedure P.

an internal node and represents the object allocated in line 1. External nodes, in many cases, represent objects that exist in the heap when the procedure is invoked (but they could also represent nodes allocated inside a method as explained in the following). In our example, n_1, n_2, and n_3 are external nodes. Specifically, n_1 represents the object pointed to by formal parameter `x` when the procedure begins execution, as indicated by the arrow from `x` to n_1. Similarly, n_2 represents the object pointed to by formal parameter `y` when the procedure begins execution.

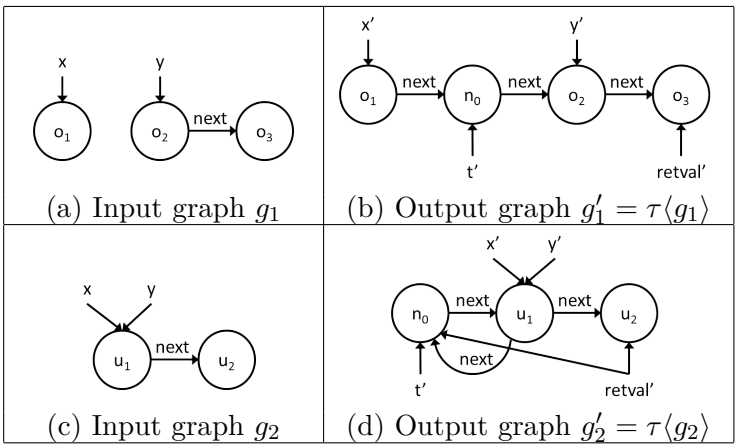
Edges in the graph are also classified into *internal* and *external* edges, shown as solid and dashed edges respectively. The edges $n_1 \rightarrow n_0$ and $n_0 \rightarrow n_2$ are internal edges. They represent updates performed by the procedure (i.e., new points-to edges added by the procedure's execution) in lines 2 and 3. External edges correspond to reads, the edge $n_2 \rightarrow n_3$ is an external edge created by the dereference "`y.next`" in line 4. This edge helps identify the node(s) that the external node n_3 represents: namely, the objects obtained by dereferencing the `next` field of objects represented by n_2.

In simple cases, internal nodes are used to represent objects created during the execution of the procedure, while external nodes are used to represent pre-existing objects (in the initial state when the procedure begins executing). More generally, external nodes are used to denote objects referenced via an access path starting from one of the procedure parameters. Thus, one may loosely associate an external node with a set of access paths.

A transformer graph τ can be interpreted as a procedure P_τ. Fig. 2.2 depicts the procedure corresponding to the transformer graph shown in Fig. 2.1(b). Every external or internal node w in a transformer graph τ corresponds to a variable $var(w)$. If w is an internal node then $var(w)$ is assigned a newly created object. Every external edge $\langle u, f, w\rangle$ corresponds to a field-read statement $var(w) = var(u).f$ and every internal edge $\langle u, f, w\rangle$ corresponds to a field-write statement $var(u).f = var(w)$. In the simplest case, the reads and writes encoded by the transformer graph via external and internal edges are assumed to happen in any order any number of times. Clearly, the procedure P_τ is an abstraction of the procedure P.

```
   (x, y) => {
[1]   while(*) {
[2]      if(*) var(n1) = x;
[3]      if(*) var(n2) = y;
[4]      var(n0) = new ();
[5]      if(*) var(n1).next = var(n0);
[6]      if(*) var(n0).next = var(n2);
[7]      if(*) var(n3) = var(n2).next;
[8]   }
[9]   t = var(n0);
[10]  retval = var(n3);
   }
```

Figure 2.2: Interpretation of τ as a procedure P_τ. $(p_1, \ldots, p_n)$ => $\{B\}$ denotes a procedure with parameters $p_1, \ldots, p_n$ and body B. (*) denotes non-deterministic choice.



(a) Input graph g_1 (b) Output graph $g'_1 = \tau\langle g_1 \rangle$

(c) Input graph g_2 (d) Output graph $g'_2 = \tau\langle g_2 \rangle$

Figure 2.3: Result of applying of τ on two different concrete states.

The above described interpretation of the transformer graph as a procedure may explain how it captures the transformation of every concrete state simultaneously. For example, consider an invocation of procedure P in an initial state given by graph g_1 shown in Fig. 2.3(a). The summary transformer when applied on the graph g_1 results in $g_1' = \tau\langle g_1 \rangle$ which represents the state after the procedure's execution (shown in Fig 2.3(b)). The graph g_1' is a conventional shape (or points-to) graph that represents a set of concrete states. The transformed graph g_1' represents the possible outputs of the procedure P_τ when executed with the concrete state g_1. Since the procedure P_τ is an abstraction of the procedure P, g_1' is an abstraction of the output of the procedure P when executed with the concrete state g_1.

An important aspect of the transformer graphs is that it can be used even in the presence of potential aliases in the input (or cut-points [Rinetzky et al., 2005]). Consider the input state g_2 shown in Fig. 2.3(c), in which parameters `x` and `y` point to the same object u_1. Executing the procedure P_τ on the input graph g_2 will result in the set of concrete graphs represented by the shape graph shown in Fig. 2.3(d). Fig. 2.3(d) is a conservative approximation of the output of the procedure P (Fig. 1.2(b) shows the actual output).

Transformer graphs Vs Shape graphs Semantically, transformer graphs represent state transformations whereas shape graphs represent concrete states. However, one might still wonder that since vertices in a transformer graph represent concrete objects and edges represent points-to relations between objects, it may be appropriate to refer to transformer graphs also as shape graphs. While the exact terminology is not important, what is important is to be aware of the differences between transformer graphs and conventional shape graphs.

Unlike conventional shape graphs, the vertices and edges in a transformer graph represent different concrete objects in different contexts. For example, in the transformer graph shown in Fig. 2.1, n_1 represents o_1 when the input graph is g_1 shown in Fig. 2.3(a), and represents u_1 when the input graph is g_2 shown in Fig. 2.3(c). Furthermore, unlike conventional shape graphs, a single concrete object may be represented

by multiple nodes in the transformer graphs. For instance, in the transformer graph shown in Fig. 2.1 when the input concrete graph is g_2 shown in Fig. 2.3(c), the nodes n_1 and n_2 represent the same concrete object u_1, and the nodes n_3 and n_0 represent the object newly allocated by the procedure.

As a consequence, the absence of an edge from an abstract node (say u) to an abstract node (say v) does not imply that objects represented by u cannot point to the objects represented by v, since the objects may have other representatives in the transformer graph.

For these reasons, many properties that hold for a shape graph do not hold for a transformer graph. For example, even if two variables point to non-intersecting sets of nodes in a transformer graph, it does not imply that the variables cannot alias (in any context).

The above informal description highlights the following properties of the transformer graphs:

- Transformer graphs are abstractions of state transformers, or equivalently, procedures. To think of them as abstractions of state is flawed.

- Transformer graphs track the (pointer valued) reads and writes performed by the procedure they summarize using internal and external edges, respectively.

- Transformer graphs can be applied to any concrete state irrespective of the aliasing between the heap cells. However, the output is a conservative approximation as the transformer graphs are abstractions of the procedure they summarize.

3

The Language and Concrete Semantics

Notation and Terminology Given a function $f : A \mapsto 2^B$, the function $\hat{f} : 2^A \mapsto 2^B$ is defined by: $\hat{f}(S) = \bigcup_{x \in S} f(x)$. Given two functions $f_1 : A \mapsto 2^B$, $f_2 : B \mapsto 2^C$ we use $(f_1 \circ f_2)(x)$ to denote the composition of f_1 with f_2 i.e., $\hat{f}_2(f_1(x))$. Note that f_1 is applied first in the composition $f_1 \circ f_2$. An element of $A \mapsto 2^B$ corresponds to a relation between A and B, and $\circ$ represents relational composition.

Syntax A program consists of a set of procedures. A procedure P consists of a control-flow graph, with an entry vertex $\mathit{entry}(P)$ and an exit vertex $\mathit{exit}(P)$. The entry vertex has no predecessor and the exit vertex has no successor. Every edge of the control-flow graph is labelled by a primitive statement. The set of primitive statements are shown in Fig. 3.1. Every procedure ends at a special primitive statement *exit*. We use $u \xrightarrow{S} v$ to indicate an edge in the control-flow graph from vertex u to vertex v labelled by statement S. We use a simple language in which all variables and fields are of pointer type.

Concrete Semantics Domain Let *Vars* denote the set of variable names used in the program, partitioned into the following disjoint sets:

the set of global variables *Globals*, the set of local variables *Locals* (assumed to be the same for every procedure), and the set of formal parameter variables *Params* (assumed to be the same for every procedure). Let *Fields* denote the set of field names used in the program. Every statement in the program has a label belonging to the set *Labels*. Let N_c be an unbounded set of locations used for dynamically allocated objects. (We will refer to an element of N_c as a vertex, node, or object.) We use a fairly common representation of the concrete state as a concrete (points-to or shape) graph.

A concrete state or points-to graph $g \in \mathbb{G}_c$ is a triple $(\mathsf{V}, \mathsf{E}, \sigma)$, where $\mathsf{V} \subseteq N_c$ represents the set of objects in the heap, $\mathsf{E} \subseteq \mathsf{V} \times \mathit{Fields} \times \mathsf{V}$ (a set of labelled edges) represents values of pointer fields in heap objects, and $\sigma \in \Sigma_c = \mathit{Vars} \mapsto \mathsf{V}$ represents the values of program variables. In particular, $(u, f, v) \in \mathsf{E}$ iff the f field of the object u points to object v. (Note that this represents the state from the perspective of a single procedure. In particular, this state does not include a call-stack, since the modular semantics can be defined without explicitly introducing a stack.) We assume N_c includes a special element *null*. Variables and fields of new objects are initialized to *null*.

Our concrete domain $\mathcal{C} = \mathbb{G}_c \mapsto 2^{\mathbb{G}_c}$ is the set of functions that map a concrete state to a set of concrete states. We define a partial order $\sqsubseteq_c$ on $\mathcal{C}$ as follows: $f_a \sqsubseteq_c f_b$ iff $\forall g \in \mathbb{G}_c . f_a(g) \subseteq f_b(g)$. Let $\sqcup_c$ denote the corresponding least upper bound (join) operation defined by: $f_a \sqcup_c f_b = \lambda g . f_a(g) \cup f_b(g)$. The subscript c may be omitted when no confusion is likely.

Lemma 3.1. $(\mathcal{C}, \sqsubseteq_c, \sqcup_c)$ is a complete lattice with the least element $\lambda g_c . \emptyset$.

Concrete Semantics Every primitive statement S has a semantics $[\![S]\!]_c \in \mathcal{C}$, as shown in Fig. 3.1. Every statement has a label $\ell \in \mathit{Labels}$ which is not used in the concrete semantics and is, hence, omitted from the figure. The execution of most statements transforms a concrete state to another concrete state, but the signature allows us to model non-determinism (e.g., dynamic memory allocation can return any unallocated object). The signature also allows us to model exe-

Statement S	**Concrete Semantics** $[\![S]\!]_c(\mathsf{V},\mathsf{E},\sigma)$
$v_1 = v_2$	$\{(\mathsf{V},\mathsf{E},\sigma[v_1 \mapsto \sigma(v_2)]\}$
$v = new\ C$	$\{(\mathsf{V} \cup \{n\}, \mathsf{E} \cup \{n\} \times \mathit{Fields} \times \{\mathit{null}\},$ $\sigma[v \mapsto n]) \mid n \in N_c \setminus \mathsf{V}\}$
$v_1.f = v_2$	$\{(\mathsf{V}, \{\langle u,l,v\rangle \in \mathsf{E} \mid u \neq \sigma(v_1)$ $\vee l \neq f\} \cup \{\langle \sigma(v_1), f, \sigma(v_2)\rangle\}, \sigma)\}$
$v_1 = v_2.f$	$\{(\mathsf{V},\mathsf{E},\sigma[v_1 \mapsto n]) \mid \langle \sigma(v_2), f, n\rangle \in \mathsf{E}\}$
exit	$\{(\mathsf{V},\mathsf{E},\lambda x.$ if $x \in (\mathit{Params} \cup \mathit{Locals})$ then *null* else $\sigma(x))\}$
$Call\ P(v_1, \cdots, v_k)$	Semantics defined below

Figure 3.1: Primitive statements and their concrete semantics.

cution errors such as null-pointer dereference, though the semantics presented simplifies error handling by treating *null* as just a special object. We will describe the semantics of a procedure-call statement along with concrete semantic equations.

We now define a concrete summary semantics $[\![P]\!]^\natural \in \mathcal{C}$ for every procedure P. The semantic function $[\![P]\!]^\natural$ maps every concrete state g_c to the set of concrete states that the execution of P with initial state g_c can produce.

We introduce a new variable φ_u for every vertex in the control-flow graph (of any procedure) and a new variable $\varphi_{u,v}$ for every edge $u \to v$ in the control-flow graph. The semantics is defined as the least fixed point of the equations shown in Fig. 3.2. The value of φ_u in the least fixed point is a function that maps any concrete state g to the set of concrete states that arise at program point u when the procedure containing u is executed with an initial state g. Similarly, $\varphi_{u,v}$ captures the states after the execution of the statement labelling edge $u \to v$.

Note that the above collection of equations is similar to those used in Sharir and Pnueli's functional approach to interprocedural analysis [Sharir and Pnueli, 1981] (extended by [Knoop and Steffen, 1992]),

$$\varphi_v = \lambda g.\{g\} \qquad v \text{ is an entry vertex} \qquad (3.1)$$

$$\varphi_v = GC_c(\bigsqcup\nolimits_c \{\varphi_{u,v} \mid u \to v\}) \qquad v \text{ is not an entry vertex} \qquad (3.2)$$

$$\varphi_{u,v} = GC_c(\varphi_u \circ [\![S]\!]_c) \qquad \text{where } u \xrightarrow{S} v \text{ and S is not a call-stmt} \qquad (3.3)$$

$$\varphi_{u,v} = GC_c(\varphi_u \circ Call_S(\varphi_{exit(Q)})) \qquad \text{where } u \xrightarrow{S} v \text{ and S is a call to proc Q} \qquad (3.4)$$

Figure 3.2: Concrete semantics equations.

with the difference that we are defining a concrete semantics here, while [Sharir and Pnueli, 1981] is focused on abstract analyses. The equations are a simple functional version of the standard equations for defining a collecting semantics, with the difference that we are simultaneously computing a collecting semantics for every possible initial state of the procedure's execution.

The first three equations are self explanatory except for the function $GC_c : \mathcal{C} \mapsto \mathcal{C}$, which is the *garbage collection* operation lifted to the domain of state transformers in $\mathcal{C}$ and is defined below.

$$\begin{aligned}
&GC_c(f) = \lambda g_i.\{RemoveUnreach(g_i, g_o) \mid g_o \in f(g_i)\} \\
&RemoveUnreach\ (\mathsf{V}_i, \mathsf{E}_i, \sigma_i)\ (\mathsf{V}_o, \mathsf{E}_o, \sigma_o) = \\
&\qquad \text{let } L = \{x \in \mathsf{V} \mid x \text{ is not reachable from } \hat{\sigma}(\mathit{Vars}) \cup \mathsf{V}_i\ \} \\
&\qquad (V_o \setminus L, E_o \setminus \{\langle u, f, v\rangle \mid u \in L\}, \lambda x.\sigma_o(x) \setminus L)
\end{aligned}$$

Given a function $f \in \mathcal{C}$, GC_c removes from the output graphs in the range of f the objects that are not reachable from the objects in the input graph and from the variables in the program. The objects in the input graph are commonly referred to as the *prestate* [Salcianu and Rinard, 2005].

Since the *exit* statement resets all local variables and parameters to *null* in the summary computed at the exit point of a procedure, the output graphs would have only objects that are reachable through the objects in the input graph or through some global variable. In other words, all the objects that are locally created by the procedure and are

$$
\begin{aligned}
push_S(\sigma) &= \lambda v.\ v \in \mathit{Globals} \rightarrow \sigma(v) \\
&\quad \mid v \in \mathit{Locals} \rightarrow \mathit{null} \\
&\quad \mid v = \mathit{Param}(i) \rightarrow \sigma(a_i) \\
pop_S(\sigma, \sigma') &= \lambda v.\ v \in \mathit{Globals} \rightarrow \sigma'(v) \\
&\quad \mid v \in \mathit{Locals} \cup \mathit{Params} \rightarrow \sigma(v) \\
Call_S(f) &= \lambda(\mathsf{V}, \mathsf{E}, \sigma).\{(\mathsf{V}', \mathsf{E}', pop_S(\sigma, \sigma')) \\
&\qquad \mid (\mathsf{V}', \mathsf{E}', \sigma') \in f(\mathsf{V}, \mathsf{E}, push_S(\sigma))\}
\end{aligned}
$$

Figure 3.3: Definition of the functions $push_S \in \Sigma_c \mapsto \Sigma_c$, $pop_S \in \Sigma_c \times \Sigma_c \mapsto \Sigma_c$, and $Call_S$ for a procedure call statement "`Call Q(`a_1`,...,`a_k`)`". In the figure, $\mathit{Param}(i)$ denotes the i-th formal parameter.

not accessible in the callers are removed from the output graphs of a summary function f.

Consider Eq. 3.4, corresponding to a call to a procedure Q. The value of $\varphi_{\mathit{exit}(Q)}$ summarizes the effect of the execution of the whole procedure Q. In the absence of local variables and parameters, we can define the right-hand-side of the equation to be simply $GC_c(\varphi_u \circ \varphi_{\mathit{exit}(Q)})$.

The function $Call_S(f)$, defined in Fig. 3.3, models the semantics of the parameter passing mechanism. Given a concrete state $(\mathsf{V}, \mathsf{E}, \sigma)$ before the call, we reset the values of all local variables to *null* and assign the formal parameter variables to the values of the corresponding actual arguments. We refer to this operation as $push_S$ as it corresponds to pushing the arguments on to the call stack and creating a new activation frame for the callee. We then apply the callee summary f that captures the effect of the procedure call on the calling context. Finally, the local variables and parameters are restored to their values before the call which corresponds to poping the activation frame of the callee. Hence, we refer to this operation as pop_S. For simplicity, we omit return values from our language.

We define $[\![P]\!]^\sharp$ to be the value of $\varphi_{\mathit{exit}(P)}$ in the least fixed point of equations (3.1)-(3.4). Specifically, let VE denote the set of vertices and edges in the control flow graph of a program. The above equations can

be expressed as a single equation $\varphi = F^\natural(\varphi)$, where $F^\natural$ is a monotonic function from the complete lattice $VE \mapsto \mathcal{C}$ to itself. Hence, $F^\natural$ has a least fixed point by Tarski's fixed point theorem.

The goal of the analysis is to compute an approximation of the set of quantities $[\![P]\!]^\natural$ using abstract interpretation.

4

The Analysis Framework

In the rest of the article we present several abstract analyses that approximate the concrete semantics presented in Chapter 3.

We represent an abstract analysis by a pair $(\mathcal{A}, \mathcal{F}_{\mathcal{A}})$, where $\mathcal{A}$ is an abstraction of the concrete domain $\mathcal{C}$, and $\mathcal{F}_{\mathcal{A}}$ maps the vertices and edges of control flow graphs to abstract transfer functions in $\mathcal{A}^n \to \mathcal{A}$ (for some positive integer n). For a vertex v of a control flow graph its transfer function is given by $\mathcal{F}_{\mathcal{A}}(v)$ (which is typically a join operation). Similarly, for an edge $u \xrightarrow{S} v$ of a control flow graph, its transfer function is given by $\mathcal{F}_{\mathcal{A}}(S)$.

The abstract analyses we present are parametric: that is, their domains and transfer functions have parameters. Instantiating the parameters using suitable definitions produces an instance. The properties that hold for the parametric semantics (like correctness and termination) carry over to the instances. Such parametric analyses can be considered as a framework that represents a family of abstract analyses. In this section, we present and discuss the most general abstract analysis $(\mathcal{A}_{\mathcal{G}}, \mathcal{F}_{\mathcal{G}})$.

The Abstract Graph Domain We now formally define the fairly standard abstract shape (or points-to) graphs used to represent a set of concrete states. The domain is parameterized by a set N_a, the universal set of all abstract graph nodes. For example, many analyses identify an abstract graph node using the label of the allocation-site. In this case, we let N_a be the set of all statement labels. An abstract shape graph $g \in \mathbb{G}_a$ is a triple $(\mathsf{V}, \mathsf{E}, \sigma)$, where $\mathsf{V} \subseteq N_a$ represents the set of abstract heap objects, $\mathsf{E} \subseteq \mathsf{V} \times \mathit{Fields} \times \mathsf{V}$ (a set of labelled edges) represents possible values of pointer fields in the abstract heap objects, and $\sigma \in \mathit{Vars} \mapsto 2^{\mathsf{V}}$ is a map representing the possible values of program variables.

Given a concrete graph $g_1 = \langle \mathsf{V}_1, \mathsf{E}_1, \sigma_1 \rangle$ and an abstract graph $g_2 = \langle \mathsf{V}_2, \mathsf{E}_2, \sigma_2 \rangle$ we say that g_1 can be embedded into g_2, denoted $g_1 \preceq g_2$, if there exists a function $h : \mathsf{V}_1 \mapsto \mathsf{V}_2$ such that

$$\langle x, f, y \rangle \in \mathsf{E}_1 \Rightarrow \langle h(x), f, h(y) \rangle \in \mathsf{E}_2 \tag{4.1}$$

$$\forall v \in \mathit{Vars}.\ \sigma_2(v) \supseteq \{h(\sigma_1(v))\} \tag{4.2}$$

The concretization $\gamma_G(g_a)$ of an abstract graph g_a is defined to be the set of all concrete graphs that can be embedded into g_a:

$$\gamma_G(g_a) = \{g_c \in \mathbb{G}_c \mid g_c \preceq g_a\}$$

4.1 The Abstract Functional Domain

A *transformer graph* $\tau \in \mathcal{A}_{\mathcal{G}}$ is a tuple $(\mathsf{EV}, \mathsf{EE}, \sigma_{in}, \mathsf{IV}, \mathsf{IE}, \sigma, \rightsquigarrow)$, where $\mathsf{EV} \subseteq N_a$ is the set of external vertices, $\mathsf{IV} \subseteq N_a$ is the set of internal vertices, $\mathsf{EE} \subseteq V \times \mathit{Fields} \times \mathsf{EV}$ is the set of external edges, where $V = \mathsf{EV} \cup \mathsf{IV}$, $\mathsf{IE} \subseteq V \times \mathit{Fields} \times V$ is the set of internal edges, $\sigma_{in} \in \mathit{Vars} \mapsto 2^V$ is a map representing the values of parameters and global variables in the *initial* state, and $\sigma \in \mathit{Vars} \mapsto 2^V$ is a map representing the possible values of program variables in the *transformed* state. $\rightsquigarrow \subseteq \mathsf{IE} \times \mathsf{EE}$ is a *may happen before* relation that tracks the relative ordering between the internal and external edges. $\langle u, f, w \rangle \rightsquigarrow \langle x, f, y \rangle$ means that the write statement $\mathit{var}(u).f = \mathit{var}(w)$ may precede the read statement $\mathit{var}(y) = \mathit{var}(x).f$ in the procedure P_τ.

Recall that in the informal overview of the transformer graph presented in Chapter 2, the program representation of a transformer graph shown in Fig. 2.2 ignores the control-flow between the field-read and field-write statements i.e. they were assumed happen in any order. A natural way to make the abstraction P_τ more precise is to preserve some of the control flow that exists in the procedure P in P_τ. For this purpose, we augment our abstract domain with a happens-before relation ($\rightsquigarrow$) that tracks the ordering between the external and internal edges.

It turns out that only the read-write ordering i.e., the ordering between external and internal edges will affect the precision of a transformer graph. The write-write ordering (ordering between internal edges) or the read-read ordering (ordering between external edges) do not affect the precision. Though, unlike reads, writes do not commute, the ordering between them can be ignored in the transformer graphs as all writes in P_τ are *non-deterministic writes*. (Notice that the writes are guarded by non-deterministic if statements in P_τ.)

Definition 4.1. Let $\tau = (\mathsf{EV}, \mathsf{EE}, \sigma_{in}, \mathsf{IV}, \mathsf{IE}, \sigma, \rightsquigarrow)$ be a transformer graph. A node u is said to be a *parameter node* if $u \in \mathit{range}(\sigma_{in})$

Definition 4.2. Let $\tau = (\mathsf{EV}, \mathsf{EE}, \sigma_{in}, \mathsf{IV}, \mathsf{IE}, \sigma, \rightsquigarrow)$ be a transformer graph. $\mathit{Escaping}(\tau) = \{y \mid \exists x \in \mathit{range}(\sigma_{in})$ s.t. y is reachable from x via $\mathsf{IE} \cup \mathsf{EE}$ edges $\}$

Intuitively, if τ is a transformer graph at some program point then $\mathit{Escaping}(\tau)$ corresponds to the set of objects that may be reachable from some *prestate* of P at that program point. (The concrete state before an invocation of P is referred to as a prestate of P). [1]

Let $f \in \mathcal{C}$ be a concrete summary. Let V_E represent the union of all the vertices in $f(g)$ that are reachable from the vertices in g, for

[1]The definition of escaping we have presented here slightly differs from the common usage of the term escaping which also includes the objects reachable from the global variables at a given program point. Our definition does not include such objects if they are not reachable from the prestate. This definition is motivated by our usage context which is explained in section 5. This definition is a generalization of the *escape set* of the WSR analysis.

each $g \in \mathbb{G}_c$. For any transformer graph τ that is an abstraction of f, $Escaping(\tau)$ is an abstraction of V_E.

Converting a Transformer graph τ to a Procedure P_τ

Interpreting a transformer graph as program helps understand the intuition behind several operations on the transformer graph. Fig. 4.1 formally presents the schema of the procedure P_τ corresponding to a transformer graph $\tau = (\mathsf{EV}, \mathsf{EE}, \sigma_{in}, \mathsf{IV}, \mathsf{IE}, \sigma, \rightsquigarrow)$. We use the program representation of a transformer graph only to convey the intuition behind the operations on the transformer graph but do not use it to formally define any of the operations.

For simplicity, we assume that the ordering relation $\rightsquigarrow = \mathsf{IE} \times \mathsf{EE}$ i.e., any write may happen before any read, and only informally describe how to extend the definition to accommodate a more precise ordering relation. In Fig. 4.1, the statements generated from the σ_{in} and the internal and external edges are guarded by non-deterministic *if* statements implying that they may or may not execute. The variable assignment statements generated from σ are enclosed by a non-deterministic *case* statement implying that at least one of these statements must execute. These constructs precisely capture the semantics of the components of the transformer graphs.

One way to extend this conversion to support a more a precise ordering relation $\rightsquigarrow$ (that indicates that certain writes cannot happen before certain reads) is to associate with every write statement W a boolean variable b_W, initialized to false. The variable is set to true after W inside the $if(*)$ construct that contains W. Every read statement R is guarded by the condition $\neg(b_1 \vee \ldots \vee b_n)$ where $b_1, b_2, \ldots, b_n$ are the boolean variables of the write statements that do not happen before R.

4.2 Concretization function

We now define the concretization function $\gamma_T : \mathcal{A}_\mathcal{G} \rightarrow \mathcal{C}$. Given a transformer graph $\tau = (\mathsf{EV}, \mathsf{EE}, \sigma_{in}, \mathsf{IV}, \mathsf{IE}, \sigma, \rightsquigarrow)$ and a concrete graph $g_c = (\mathsf{V}_c, \mathsf{E}_c, \sigma_c)$, we need to construct a graph representing the trans-

```
(p_1, p_2, ..., p_n) => {
  while(*) {
   Init(p_1)
    ⋮
   Init(p_k)
   var(z_1) = new ();
   ⋮
   var(z_s) = new ();
   if(*)
    var(u_1) = var(w_1).f_1;
    ⋮
   if(*)
    var(u_n) = var(w_n).f_n;
   if(*)
    var(x_1).g_i = var(y_1);
    ⋮
   if(*)
    var(x_m).g_m = var(y_m);
  }
  Fin(v_1)
   ⋮
  Fin(v_t)
 }
```

```
Init(p_i) = {
  //let σ_in(p_i) = {a_1, ··· , a_j}
  if(*) var(a_1) = p_i;
   ⋮
  if(*) var(a_j) = p_i;
 }

Fin(v_i) = {
  //let σ(v_i) = {b_1, ··· , b_j}
  case {
   (*) → v_i = var(b_1);
    ⋮
   (*) → v_i = var(b_j);
  }
 }
```

Figure 4.1: The program P_τ corresponding to a transformer graph $\tau = (\mathsf{EV}, \mathsf{EE}, \sigma_{in}, \mathsf{IV}, \mathsf{IE}, \sigma, \rightsquigarrow)$, where *Params* $\cup$ *Globals* $= \{p_1, p_2, \ldots, p_k\}$, *Vars* $= \{v_1, v_2, \ldots, v_t\}$, $\mathsf{IV} = \{z_1, \cdots, z_s\}$, $\mathsf{EE} = \{\langle u_1, f_1, w_1\rangle, \ldots, \langle u_m, f_m, w_n\rangle\}$, $\mathsf{IE} = \{\langle x_1, g_1, y_1\rangle, \ldots, \langle x_s, g_s, y_m\rangle\}$. In the figure, `Init` and `Fin` are macros.

formation of g_c by τ. As explained in section 4.1, the transformer graph can be interpreted as a program P_τ in which every internal and external vertex u becomes a variable $var(u)$ and every internal and external edge becomes a read or write statement. The output of the procedure P_τ when executed with the concrete state g_c is the transformation represented by τ. However, we define the concretization function mathematically without explicitly constructing the procedure P_τ.

Given a graph $g_c \in \mathcal{C}$, $\gamma_T(\tau)(g_c)$ is defined in two steps: in the first step, we compute the set of vertices each node in τ represents, which is equivalent to computing the *points-to set* for each of the variables in the program P_τ. In the second step, we compute an abstract graph g_a at the exit of the procedure P_τ using the points-to sets of the variables. These two steps together constitute the operation $\tau\langle g_c\rangle$ (which was illustrated in the Fig. 2.3). The result of $\gamma_T(\tau)(g_c)$ is the concrete image of $\tau\langle g_c\rangle$, which is the set of all graphs that can be embedded in $\tau\langle g_c\rangle$.

We now define a function $\eta[\![\tau, g_c]\!] : (\mathsf{IV} \cup \mathsf{EV}) \mapsto 2^{(\mathsf{IV} \cup \mathsf{V}_c)}$ that maps each node in the transformer graph τ to a set of concrete nodes in g_c as well as internal nodes in τ. (We ignore the implicit parameters τ and g_c of η whenever it is clear from the context). For any node $u \in (\mathsf{EV} \cup \mathsf{IV})$, $\eta(u)$ is equivalent to the *points-to set* of $var(u)$ in P_τ. η is defined as the least solution of the following set of constraints over the variable μ.

$$v \in \mathsf{IV} \Rightarrow v \in \mu(v) \tag{4.3}$$

$$v \in \sigma_{in}(\mathtt{X}) \Rightarrow \sigma_c(\mathtt{X}) \in \mu(v) \tag{4.4}$$

$$\langle u, f, v\rangle \in \mathsf{EE}, u' \in \mu(u), \langle u', f, v'\rangle \in \mathsf{E}_c \Rightarrow v' \in \mu(v) \tag{4.5}$$

$$\left\{ \begin{array}{c} \langle u, f, v\rangle \in \mathsf{EE}, \langle u', f, v'\rangle \in \mathsf{IE}, \\ \mu(u) \cap \mu(u') \neq \emptyset, \\ \langle u', f, v'\rangle \rightsquigarrow \langle u, f, v\rangle \end{array} \right\} \Rightarrow \mu(v') \subseteq \mu(v) \tag{4.6}$$

Explanation of the constraints:

An internal node v maps to itself (Eq. 4.3) as it represents a newly allocated object. If $\mathtt{X}$ is a parameter then an external node $v \in \sigma_{in}(\mathtt{X})$ represents the node pointed to by $\mathtt{X}$ in the input state g_c (Eq. 4.4). This is because, by the construction of P_τ, $var(v)$ will be assigned to $\mathtt{X}$ and hence the points-to set of $var(v)$ will include the targets of $\mathtt{X}$.

An external edge $\langle u, f, v\rangle$ corresponds to a read statement $var(v) = var(u).f$ in P_τ. If a concrete node u' belongs to the points-to set of $var(u)$ (i.e. $u' \in \mu(u)$) and if u''s f field points-to v' in the input concrete graph (i.e, $\langle u', f, v'\rangle \in \mathsf{E}_c$) then v' belongs to the points-to set of $var(v)$ (i.e. $v' \in \mu(v)$) as given by Eq. 4.5. Note that in the program P_τ no strong updates are possible. Therefore, the edge $\langle u', f, v'\rangle$ could not be removed by any statement in the program.

Finally, a read statement $var(v) = var(u).f$ implies that $var(v)$ may point to objects assigned to the f field of $var(u)$ or its aliases during the procedure's execution. Eq. 4.6 handles this case. The precondition identifies $var(u')$ as a potential alias for $var(u)$ by checking if their points-to sets intersect, and identifies the writes performed on the f field of $var(u')$ that precede the read: $var(v) = var(u).f$, using the happens before ($\rightsquigarrow$) relation. For every such write: $var(u').f = var(v')$, the objects pointed to by $var(v')$ are included in the points-to set of $var(v)$.

Given a mapping function η, we define the transformed abstract graph $\tau\langle g_c\rangle$ as $\langle \mathsf{V}', \mathsf{E}', \sigma'\rangle$, where

$$\mathsf{V}' = \mathsf{V}_c \cup \mathsf{IV} \tag{4.7}$$

$$\mathsf{E}' = \mathsf{E}_c \cup \bigcup_{\langle u,f,v\rangle \in \mathsf{IE}} \eta(u) \times \{f\} \times \eta(v) \tag{4.8}$$

$$\sigma' = \lambda x. \bigcup_{u \in \sigma(x)} \eta(u) \tag{4.9}$$

The transformed graph is an *abstract* graph that represents all concrete graphs that can be embedded in the abstract graph. Thus, we define the concretization function as below:

$$\gamma_T(\tau_a) = \lambda g_c.\gamma_G(\tau\langle g_c\rangle).$$

Containment Ordering A natural "precision ordering" exists on $\mathcal{A}_\mathcal{G}$, where τ_1 is said to be more precise than τ_2 iff $\gamma_T(\tau_1) \sqsubseteq_c \gamma_T(\tau_2)$. However, this ordering is not of immediate interest to us. (It is not even a partial order, and is hard to work with computationally.) We utilize a stricter ordering in our abstract fixed point computation. Let $\tau_1 = (\mathsf{EV}_1, \mathsf{EE}_1, \sigma_{in1}, \mathsf{IV}_1, \mathsf{IE}_1, \sigma_1, \rightsquigarrow_1)$ and $\tau_2 =$

$(\mathsf{EV}_2, \mathsf{EE}_2, \sigma_{in2}, \mathsf{IV}_2, \mathsf{IE}_2, \sigma_2, \rightsquigarrow_2)$. We define a relation $\sqsubseteq_{co}$ on $\mathcal{A}_\mathcal{G}$ by: $\tau_1 \sqsubseteq_{co} \tau_2$ iff every component of τ_1 is contained in the corresponding component of τ_2, i.e, $\mathsf{EV}_1 \subseteq \mathsf{EV}_2$, $\mathsf{EE}_1 \subseteq \mathsf{EE}_2$, $\forall x.\sigma_{in1}(x) \subseteq \sigma_{in2}(x)$, $\mathsf{IV}_1 \subseteq \mathsf{IV}_2$, $\mathsf{IE}_1 \subseteq \mathsf{IE}_2$, $\forall x.\sigma_1(x) \subseteq \sigma_2(x)$ and $\rightsquigarrow_1 \subseteq \rightsquigarrow_2$.

Lemma 4.1. $\sqsubseteq_{co}$ is a partial-order on $\mathcal{A}_\mathcal{G}$ with a join operation, denoted $\sqcup_{co}$. Further, γ_T is monotonic with respect to $\sqsubseteq_{co}$: $\tau_1 \sqsubseteq_{co} \tau_2 \Rightarrow \gamma_T(\tau_1) \sqsubseteq_c \gamma_T(\tau_2)$.

Abstraction function α_T It can be observed that there exists multiple elements in the abstract domain representing the same concrete value. There is no specific way of making a distinguishing choice among the possible alternatives. Hence, we do not define an abstraction function α_T.

Our abstract interpretation formulation uses only a concretization function. While this form is less common, it is sufficient to establish the soundness of the analysis as explained in [Cousot and Cousot, 1992], section 7. Specifically, a concrete value $f \in \mathcal{C}$ is *correctly represented* by an abstract value $\tau \in \mathcal{A}_\mathcal{G}$, denoted $f \sim \tau$, iff $f \sqsubseteq_c \gamma_T(\tau)$. We seek to compute an abstract value that correctly represents the least fixed point of the concrete semantic equations.

5

Parametric Abstract Semantics

Fig. 5.1 shows the constituents of the most general abstract semantics. The equations 5.1–5.4 are the abstract semantics equations that approximate the concrete semantics equations 3.1–3.4. We introduce a variable ϑ_u for every vertex u in the control-flow graph denoting the abstract value at a program point u, and a variable $\vartheta_{u,v}$ for every edge $u \to v$ in the control-flow graph denoting the abstract value after the execution of the statement in edge $u \to v$.

We denote using τ_{id} (defined shortly) the transformer graph representing the identity function. $\mathcal{F}_{\mathcal{G}}(S)$ is the abstract semantics (or transfer function) of a statement S. $\mathit{Simplify}_S$ is a function from $\mathcal{A}_{\mathcal{G}}$ to $\mathcal{A}_{\mathcal{G}}$ that reduces the number of vertices in the input graph (see Theorem 5.6). $GC_{\mathcal{G}}$ is an abstract garbage collection operation analogous to the concrete garbage collection operation (see Theorem 5.7).

The operations $\mathit{Simplify}_S$ and $GC_{\mathcal{G}}$ have a special property that they can be applied over the abstract value of any edge $u \xrightarrow{S} v$ or vertex of the control flow graph without affecting the correctness or termination of the analysis. This is the reason for separating these operations from the semantics equations in Fig. 5.1. Hence, Fig. 5.1 represents a family of abstract semantics equations that extend the

Semantics Equations

$$\vartheta_v = \tau_{id} \qquad v \text{ is an entry vertex} \tag{5.1}$$

$$\vartheta_v = \sqcup_{co}\{\vartheta_{u,v} \mid u \xrightarrow{S} v\} \qquad v \text{ is not an entry vertex} \tag{5.2}$$

$$\vartheta_{u,v} = \mathcal{F}_{\mathcal{G}}(S)(\vartheta_u) \qquad \text{where } u \xrightarrow{S} v, \text{S is not a call-stmt} \tag{5.3}$$

$$\vartheta_{u,v} = \mathcal{F}_{\mathcal{G}}(S)(\vartheta_u, \vartheta_{exit(Q)}) \qquad \text{where } u \xrightarrow{S} v, \text{S is a call to Q} \tag{5.4}$$

Correctness Preserving Operations

$$\mathit{Simplify}_S \in \mathcal{A}_{\mathcal{G}} \mapsto \mathcal{A}_{\mathcal{G}}$$
$$GC_{\mathcal{G}} \in \mathcal{A}_{\mathcal{G}} \mapsto \mathcal{A}_{\mathcal{G}}$$

Figure 5.1: Constituents of the parametric abstract semantics.

equations 5.1–5.4 by composing *Simplify* and $GC_{\mathcal{G}}$ with the transfer functions of any arbitrary set of edges and vertices of the control flow graphs.

5.0.1 Parameters of the Abstract Semantics

Recall that the domain $\mathcal{A}_{\mathcal{G}}$ defined earlier is parameterized by the set N_a. Similarly, the abstract semantics is also parameterized by the following functions which are used while creating abstract nodes. (In the following, $(2^{N_a} \setminus \emptyset)$ denotes the power set of N_a without the empty set.)

(a) An initialization function $\mathit{InitBind} : (\mathit{Params} \cup \mathit{Globals}) \mapsto (2^{N_a} \setminus \emptyset)$ that initializes *Params* and *Globals* to abstract vertices.

(b) An abstract vertex creation function $\mathit{Alloc}_S : (\texttt{optional}\ N_a) \mapsto (2^{N_a} \setminus \emptyset)$, where S is a statement. This function is used for creating new abstract objects. The function is passed an optional candidate vertex and returns a set of vertices representing newly created objects.

(c) An abstract vertex load function $\mathit{Load}_S : N_a \times \mathit{Fields} \times (\texttt{optional}\ N_a) \mapsto (2^{N_a} \setminus \emptyset)$, where S is a statement. This function

is used to model the reads performed on abstract objects. The function is passed a vertex that is dereferenced, the dereferenced field and an optional candidate vertex. It returns a set of vertices representing the dereferenced object.

In most cases, *InitBind*, $Load_S$ and $Alloc_S$ return a single abstract vertex. Notice that the parameters are defined for each program statement. This allows the parameter functions to have a statement specific definition. The parameters help define a generic semantics that is completely oblivious to the naming strategies used to name abstract vertices.

In fact, different instances of our framework use different naming strategies. For instance, the WSR analysis initializes parameter and global variables to abstract vertices that have the same name as the variables. The internal and external vertices are named using the *labels* of the statements that resulted in their creation. Formally, this corresponds to the following definition of the parameters in our framework.

$$\begin{aligned} N_a &= \{n_x \mid x \in \mathit{Labels} \cup \mathit{Params} \cup \mathit{Globals}\} \\ \mathit{InitBind} &= \lambda x.\{n_x\} \\ \mathit{Alloc}_{\ell:v=new()} &= \lambda x.\{n_\ell\} \\ \mathit{Load}_{\ell:v_1=v_2.f} &= \lambda(x,f,y).\{n_\ell\} \end{aligned}$$

The correctness of the abstract semantics does not depend on the definitions of the parameters. The termination of the abstract semantics only requires the parameter definitions to be terminating functions. The abstract semantics of the framework (presented shortly) uses the parameters in a controlled way in order to ensure these properties. However, the scalability and the precision of the abstract semantics depend on the definition of the parameters to a large extent.

This we believe is the main practical advantage of the framework. This allows us to tune the abstract semantics to the required level of precision and scalability without having to be concerned with the correctness or termination of the analysis. As we will illustrate later with concrete instances, the framework offers a large spectrum of options to experiment with.

Stmt S	$\mathcal{F}_{\mathcal{G}}(S)(\mathsf{EV}, \mathsf{EE}, \sigma_{in}, \mathsf{IV}, \mathsf{IE}, \sigma, \rightsquigarrow)$
$v_1 = v_2$	$(\mathsf{EV}, \mathsf{EE}, \sigma_{in}, \mathsf{IV}, \mathsf{IE}, \sigma[v_1 \mapsto \sigma(v_2)], \rightsquigarrow)$
$v = new\ C$	$let\ N = Alloc_S()\ in$ $let\ \mathsf{IE}_{new} = N \times Fields \times \{null\}\ in$ $(\mathsf{EV}, \mathsf{EE}, \sigma_{in}, \mathsf{IV} \cup N, \mathsf{IE} \cup \mathsf{IE}_{new}, \sigma[v \mapsto N], \rightsquigarrow)$
$v_1.f = v_2$	$(\mathsf{EV}, \mathsf{EE}, \sigma_{in}, \mathsf{IV}, \mathsf{IE} \cup \sigma(v_1) \times \{f\} \times \sigma(v_2), \sigma, \rightsquigarrow)$
$v_1 = v_2.f$	let $g = \lambda u.$ $(\exists x.\langle u, f, x\rangle \in \mathsf{EE}) \rightarrow \bigcup_{\langle u,f,x\rangle \in \mathsf{EE}} Load_S(u, f, x)$ $\mid Load_S(u, f)$ in let $\mathsf{EV}_{new} = \bigcup_{u \in \sigma(v_2)} g(u)$ in let $\mathsf{EE}_{new} = \bigcup_{u \in \sigma(v_2)} \{u\} \times f \times g(u)$ in $(\mathsf{EV} \cup \mathsf{EV}_{new}, \mathsf{EE} \cup \mathsf{EE}_{new}, \sigma_{in}, \mathsf{IV}, \mathsf{IE}, \sigma[v_1 \mapsto \mathsf{EV}_{new}],$ $\rightsquigarrow \cup \{(ie, ee) \mid ie \in \mathsf{IE}, ee \in \mathsf{EE}_{new}\})$
exit	$(\mathsf{EV}, \mathsf{EE}, \sigma_{in}, \mathsf{IV}, \mathsf{IE},$ $\lambda x.(x \in Params \cup Locals) \rightarrow null \mid \sigma(x))$

Figure 5.2: Abstract semantics of primitive instructions.

5.1 Abstract Semantics of Primitive Statements

The transformer graph τ_{id} used in Fig. 5.1 is the transformer graph representing the identity function and is defined as follows.

$$\tau_{id} = (\mathsf{EV}, \emptyset, \sigma_{in}, \emptyset, \emptyset, \sigma_{in}, \emptyset), \text{ where}$$
$$\mathsf{EV} = \bigcup_{x \in Params \cup Globals} InitBind(x)$$
$$\sigma_{in} = \lambda v.\ v \in Params \cup Globals \rightarrow InitBind(v) \mid v \in Locals \rightarrow \{null\}$$

Fig. 5.2 shows the transfer functions for the primitive statements. The function $\mathcal{F}_{\mathcal{G}}(S)$ can be considered as the most basic abstract se-

mantics without any optimizations. The transfer function of a variable assignment statement $v_1 = v_2$ makes the targets of v_1 equal to the targets of v_2.

Consider the transfer function of an object allocation statement $v = new\ C$. The transfer function creates new abstract nodes N for representing the newly allocated object using the *Alloc* parameter. (*Alloc* will generally create a single abstract node to represent the newly created object. However, the definition also permits the use of multiple abstract objects). The abstract nodes are added to the set of internal vertices as they represent an object created within the analysed procedure.

Recall that in our concrete semantics the fields of the newly created object are initialized to *null*. We capture this effect in the abstract semantics by creating new edges from the abstract nodes N to the *null* object. These edges are added to the set of internal edges as they represent writes. Finally, the abstract nodes N are made the new targets of the variable v.

The transfer function of a field-write statement $v_1.f = v_2$ is straightforward. It creates new internal edges from the targets of v_1 to the targets of v_2 labelled by the field f. The internal edges basically record that the field f of the targets of v_1 are assigned to the targets of v_2.

The transfer function of a field-read statement $v_1 = v_2.f$ is quite involved. However, in essence, the goal is to create external vertices to model the targets of $v_2.f$, and to create external edges to record that the field f of the targets of v_2 are read. One important question is how to choose the external vertices to represent the targets of $v_2.f$? The choice of the external vertices may affect the precision and scalability of the analysis.

For example, consider a transformer graph τ in which the targets of v_2 are two nodes u and v i.e, $\sigma(v_2) = \{u, v\}$. Also, say that there exists two variables v_3 and v_4 whose targets are u and v, respectively. That is, $\sigma(v_3) = \{u\}$ and $\sigma(v_4) = \{v\}$. If we use a single node w as the target of both the external edges starting from u and v, we end up collapsing the targets of $v_3.f$ and $v_4.f$ as well. This would result in loss of precision. On the other hand, using two different vertices

as the targets of edges starting from u and v will increase the sizes of the transformer graph, which may increase the running time of the analysis. Therefore, we parameterize the creation of the targets of the external edges in the transfer function of a field-read statement. Most of the sophistication in the transfer function presented in Fig. 5.2 is for achieving this parameterization. We describe the transfer function in detail below.

We refer to a vertex that belongs to $\sigma(v_2)$ as a *dereferenced vertex*. The function g, defined in Fig. 5.2, determines the targets of the f field of a dereferenced vertex u using the parameter *Load*. If u already has external edges on field f of the form $\langle u, f, x\rangle$ then, for every such edge, we apply the $Load_S$ parameter over the triple (u, f, x). This allows us to define a semantics in which the external vertices created during the previous field-read statements are reutilized. For example, if $Load_S(u, f, x)$ is defined as $\{x\}$ then the targets of the previous reads of the field f of the vertex u would be reused to represent the targets of the current read. On the other hand, if there exists no external edge from u on field f then we apply *Load* on (u, f) as we do not have a candidate vertex to reuse.

Once we know the targets of the f fields of the dereferenced vertices, the rest of the semantics is straight forward. We add the targets of all the dereferenced vertices, computed by the function g, to the external vertex set. We also make them the new targets of v_1. For every dereferenced vertex u, we create external edges from u to the vertices in $g(u)$. Note that the internal edges existing in the transformer graph before the field-read statement represent writes that have happened before the current read. We update the happens before relation $\rightsquigarrow$ to reflect this fact.

5.2 Abstract Semantics of Procedure Call

Let $S : Q(a_1, a_2, \cdots, a_n)$ be a call statement. Let τ_r be the transformer graph in the caller before the statement S and let τ_e be the abstract summary of Q. The function $\mathcal{F}_{\mathcal{G}}(S)(\tau_r, \tau_e)$ is defined as follows:

$$\mathcal{F}_{\mathcal{G}}(S)(\tau_r, \tau_e) = pop^{\sharp}_S(\tau_e \langle\!\langle push^{\sharp}_S(\tau_r)\rangle\!\rangle_S, \tau_r) \tag{5.5}$$

$$
\begin{aligned}
push^{\sharp}_S(\sigma) &= \lambda v.\ (v = Param(i) \rightarrow \sigma(a_i) \\
&\quad \mid v \in Globals \rightarrow \sigma(v) \\
&\quad \mid v \in Locals \rightarrow null) \\
pop^{\sharp}_S(\sigma, \sigma') &= \lambda v.\ (v \in Params \cup Locals \rightarrow \sigma(v) \\
&\quad \mid v \in Globals \rightarrow \sigma'(v)) \\
push^{\sharp}_S(\tau) &= (\mathsf{EV}, \mathsf{EE}, \sigma_{in}, \mathsf{IV}, \mathsf{IE}, push^{\sharp}_S(\sigma)) \\
pop^{\sharp}_S(\tau', \tau) &= (\mathsf{EV}', \mathsf{EE}', \sigma_{in}{}', \mathsf{IV}', \mathsf{IE}', pop^{\sharp}_S(\sigma, \sigma'))
\end{aligned}
$$

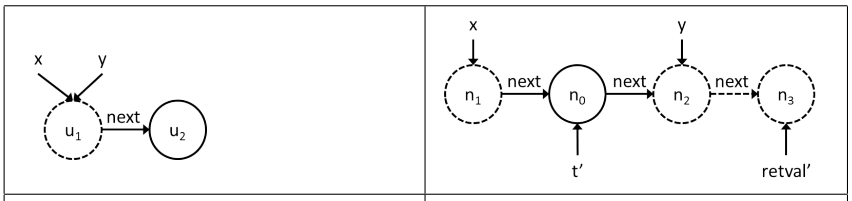
Figure 5.3: Definitions of the abstract push and pop operations $push^{\sharp}_S$ and $pop^{\sharp}_S$. In the figure, S is the call statement $Q(a_1, \cdots, a_n)$, $\tau = (\mathsf{EV}, \mathsf{EE}, \sigma_{in}, \mathsf{IV}, \mathsf{IE}, \sigma, \rightsquigarrow)$ and $\tau' = (\mathsf{EV}', \mathsf{EE}', \sigma_{in}{}', \mathsf{IV}', \mathsf{IE}', \sigma')$.

where, the $push^{\sharp}_S$ and $pop^{\sharp}_S$, defined in Fig. 5.3, are the abstract analogues of the concrete $push_S$ and pop_S operations. They perform the mapping of the formal arguments to actual parameters and vice versa. The function $\langle\langle\rangle\rangle : \mathcal{A}_{\mathcal{G}} \times \mathcal{A}_{\mathcal{G}} \mapsto \mathcal{A}_{\mathcal{G}}$ is the composition operation for the transformer graphs and is explained in the sequel.

5.2.1 The Composition Operation

Given two transformer graphs τ_1 and τ_2, $\tau_2 \langle\langle \tau_1 \rangle\rangle_S$ is a transformer graph equivalent to applying τ_1 followed by τ_2. We describe the basic idea behind the composition operation using an example before presenting a formal definition.

Consider Fig. 5.4. Let τ_1 and τ_2 be the two transformer graphs shown at the top of the Fig. 5.4. The programs P_{τ_1} and P_{τ_2} constructed from τ_1 and τ_2 are shown at the bottom of the Fig. 5.4. For conciseness we use the node ids to denote the corresponding variables. Fig. 5.5(a) shows the program P' obtained by composing the programs P_{τ_1} and P_{τ_2} i.e, $P' = P_{\tau_1}; P_{\tau_2}$. The goal is to construct a transformer graph that is an abstraction of the program P'. Unfortunately, P' itself cannot be interpreted as a transformer graph. This is because P' has multiple while loops and has statements that assign values to program variables




```
   (x, y) => {
[1]   while(*) {
[2]      if(*) u1 = x;
[3]      if(*) u1 = y;
[4]      u2 = new ();
[5]      if(*) u1.next = u2;
[6]   }
[7]   x = u1;
[8]   y = u2;
   }
```

(a)

```
   (x, y) => {
[1]   while(*) {
[2]      if(*) n1 = x;
[3]      if(*) n2 = y;
[4]      n0 = new ();
[5]      if(*) n1.next = n0;
[6]      if(*) n0.next = n2;
[7]      if(*) n3 = n2.next;
[8]   }
[9]   t = n0;
[10]  retval = n3;
[11]  x = n1;
[12]  y = n2;
   }
```

(b)

Figure 5.4: (a) Transformer graph τ_1 and its program interpretation P_{τ_1}. (b) Transformer graph τ_2 and its program interpretation P_{τ_2}.

`x`, `y` (see lines [7] and [8]). In a program that corresponds to a transformer graph such statements can occur only at the end of the program (see the formal definition of constructing a program from a transformer graph presented in section 4.1).

The statements at lines [7] and [8] define the values of the variables after the application of τ_1. They can be eliminated by replacing the parameter nodes of P_{τ_2} corresponding to the parameter variables x and y, namely n_1 and n_2, by the values of x and y at the end of P_{τ_1}, namely u_1. After this step, the two loops can be abstracted into a single while loop encompassing the statements of the loops. The program thus obtained is shown in Fig. 5.5(b). The lines [13]–[15] in P' become lines [7]–[9] in $P_{\tau'}$, in which n_1 and n_2 are replaced by u_1. This program can be interpreted as a transformer graph τ' which is shown in Fig. 5.5(c).

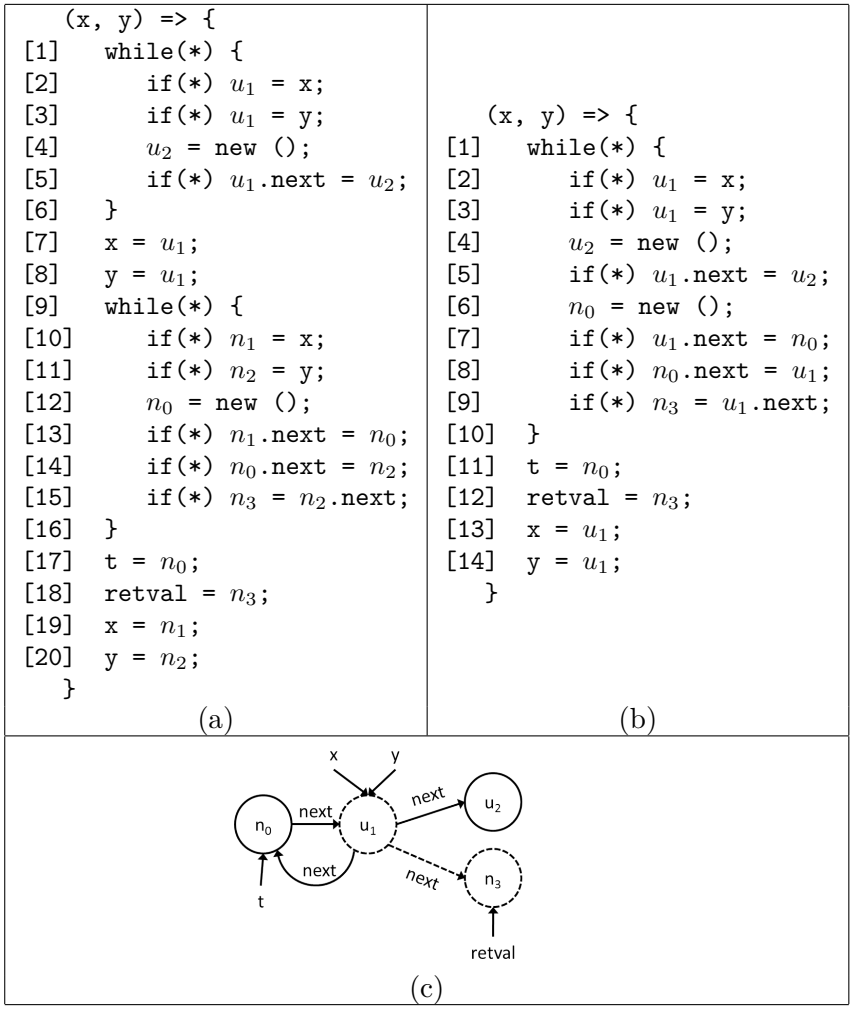


Figure 5.5: (a) The procedure $P' = P_{\tau_1}; P_{\tau_2}$. (b) Procedure $P_{\tau'}$ obtained by eliminating parameter nodes of τ_2. (c) The transformer graph corresponding to $P_{\tau'}$, which is equal to $\tau_2\langle\langle\tau_1\rangle\rangle_S$.

The approach illustrated above generalizes to any pair of transformer graphs. Informally, to compose a transformer graph τ_1 with τ_2 we perform the following steps. For every parameter variable $\mathtt{X}$ of τ_2, we eliminate the parameters nodes $\sigma_{in}(\mathtt{X})$ from every component of the transformer graph τ_2 by replacing each of them with $\sigma_1(\mathtt{X})$, which are the (possible) value of the variable $\mathtt{X}$ resulting at the end of τ_1. We define the (internal and external) edges of the composed graphs as the union of the edges of τ_1 and the edges of τ_2 obtained after the elimination of parameter nodes.

The initial variable mapping σ_{in} of the composed graph is given by σ_{in1}. The final variable mapping of the composed graph is given by the mapping obtained after the elimination of parameter nodes from σ_2. The internal and external vertices, other than the parameter nodes, contained in the transformer graph τ_2 are retained in the composed transformer graph. With this informal description we now proceed to the formal definition.

Let $\mathsf{V}_2 = \mathsf{EV}_2 \cup \mathsf{IV}_2$. We first define a function $\eta[\![\tau_2, \tau_1]\!] : \mathsf{V}_2 \mapsto 2^{N_a}$ that maps the vertices in τ_2 to a set of abstract vertices belonging to τ_1 and τ_2. We elide the implicit parameters of η namely τ_1, τ_2 whenever it is clear from the context. η is defined using the following constraints:

$$x \in (\mathsf{EV}_2 \setminus range(\sigma_{in2})) \Rightarrow x \in \eta(x) \tag{5.6}$$

$$x \in \sigma_{in2}(\mathtt{X}) \Rightarrow \sigma_1(\mathtt{X}) \subseteq \eta(x) \tag{5.7}$$

$$x \in \mathsf{IV}_2 \Rightarrow Alloc_S(x) \in \eta(x) \tag{5.8}$$

$$\langle u, f, x\rangle \in \mathsf{EE}_2, a \in \eta(u) \Rightarrow Load_S(a, f, x) \in \eta(x) \tag{5.9}$$

The first two constraints 5.6 and 5.7 follow from the informal explanation. The parameter nodes of a variable $\mathtt{X}$ are mapped to the values of $\mathtt{X}$ at the end of P_{τ_1} which is given by $\sigma_1(\mathtt{X})$, and the remaining external vertices are mapped to themselves. The constraints 5.8 and 5.9 add more flexibility to the composition operation by incorporating the parameters of the abstract semantics. This will be explained in more detail shortly.

For example, consider the composition of the transformer graph τ_1 with the transformer graph τ_2 shown in Fig. 5.4(a) and (b), respectively. In this case, $\eta(n_1) = \eta(n_2) = \{u_1\}$, $\eta(n_3) = Load_S(u_1, next, n_3)$

$$
\begin{aligned}
trans_{IE}(\mathsf{IE}, \mu) &= \bigcup_{\langle u,f,v\rangle \in \mathsf{IE}} \mu(u) \times \{f\} \times \mu(v) \\
trans_{EE}(\mathsf{EE}, \mu) &= \bigcup_{\langle u,f,v\rangle \in \mathsf{EE}} \{ \bigcup_{a \in \mu(u)} \{a\} \times \{f\} \times Load_S(a, f, v)\} \\
trans_{Sigma}(\sigma, \mu) &= \lambda x.\ \hat{\mu}(\sigma(x)) \\
trans_{HB}(\rightsquigarrow, \mu) &= \bigcup_{ie \rightsquigarrow ee} trans_{IE}(\{ie\}, \mu) \times trans_{EE}(\{ee\}, \mu)
\end{aligned}
$$

Figure 5.6: Translation of a transformer graph with respect to a mapping μ.

and $\eta(n_0) = Alloc_S(n_0)$. For now, assume that the *Load* and *Alloc* functions return the candidate vertex passed to the functions. That is, $Load_S(u_1, next, n_3) = \{n_3\}$ and $Alloc_S(n_0) = \{n_0\}$.

Consider the family of operations *trans* shown in Fig. 5.6 that given a mapping μ on abstract nodes translates a component of a transformer graph, which could be a set of internal or external edges, a happens before relation, or a mapping from variables to abstract objects (σ), by applying the function μ point-wise on every vertex contained in the component. The translation of external edges, however, applies the parameter *Load* instead of μ to the targets of external edges.

Intuitively, if the *trans* function is applied over η (defined by 5.6–5.9) and a component of the transformer graph τ_2 then it replaces the parameters nodes of τ_2 by the values at the end of τ_1 in the given component of τ_2.

For example, for the transformer graphs τ_1 and τ_2 shown in Fig. 5.4(a) and (b) respectively, $trans_{EE}(\{\langle n_2, next, n_3\rangle\}, \eta)$ is equal to $\{\langle u_1, next, n_3\rangle\}$, $trans_{IE}(\{\langle n_1, next, n_0\rangle\}, \eta)$ equals $\{\langle u_1, next, n_0\rangle\}$, and $trans_{Sigma}(x \mapsto n_1, \eta) = (x \mapsto u_1)$, where η is the mapping function presented earlier.

We define the composed transformer graph as:

$$\tau_2 \langle\!\langle \tau_1 \rangle\!\rangle_S = Append(\tau_1, \tau_2, \eta),$$

where *Append*, defined below, is a function that translates the components of τ_2 by applying η and appends it to τ_1. Let the components of

τ_1 be denoted using subscript $_1$ and those of τ_2 using the subscript $_2$. $Append(\tau_1, \tau_2, \eta) = (\mathsf{EV}', \mathsf{EE}', {\sigma_{in}}', \mathsf{IV}', \mathsf{IE}', \sigma', \rightsquigarrow')$, where

$$\begin{aligned}
\mathsf{EV}' &= \mathsf{EV}_1 \cup (\mathsf{EV}_2 \setminus range(\sigma_{in2}) \cup \{v \mid \langle u, f, v\rangle \in \mathsf{EE}'\} \\
\mathsf{IV}' &= \mathsf{IV}_1 \cup \hat{Alloc_S}(\mathsf{IV}_2) \\
\mathsf{EE}' &= \mathsf{EE}_1 \cup trans_{EE}(\mathsf{EE}_2, \eta) \\
\mathsf{IE}' &= \mathsf{IE}_1 \cup trans_{IE}(\mathsf{IE}_2, \eta) \\
{\sigma_{in}}' &= \sigma_{in1} \\
\sigma' &= trans_{Sigma}(\sigma_2, \eta) \\
\rightsquigarrow' &= \rightsquigarrow_1 \cup trans_{HB}(\rightsquigarrow_2, \eta) \cup \mathsf{IE}_1 \times \mathsf{EE}_2
\end{aligned}$$

Except for the definitions of EV' (the set of external vertices of the composed graph) and $\rightsquigarrow'$ the other definitions are straight forward. EV' includes all the external vertices of τ_1, all the external vertices of τ_2 except the parameter vertices, and all the vertices that were created using the *Load* parameter during the translation of external edges (these vertices are the targets of external edges in the composed transformer graph).

The ordering relation $\rightsquigarrow'$ includes $\mathsf{IE}_1 \times \mathsf{EE}_2$ as we are concatenating τ_1 with τ_2 which implies that the edges in τ_1 happen before those in τ_2.

5.2.2 Parmeterizing the Composition Operation

We now explain the need for incorporating the parameters *Alloc* and *Load* in the definition of composition operation presented above.

The parameters *Alloc* and *Load* enable fine tuning of the *context-sensitivity* of the analysis. It is well known that when abstract objects representing newly allocated objects are named based on their allocation sites, cloning (or renaming) of abstract objects for each call site during a heap analysis increases the context-sensitivity of the analysis (e.g, see [Liang and Harrold, 2001], [Lattner et al., 2007]). Since the internal vertices represent objects newly allocated within the analysed scope and they *may be* named based on their allocation sites, we incorporate the parameter $Alloc_S$ to support cloning of internal vertices during call statements.

The parameter $Load_S$ serves a similar purpose, namely to support the cloning of external vertices. Since the external vertices are somewhat unique to the transformer graphs, the need for cloning them may not be immediately obvious. Hence, as an example, consider again the transformer graphs and programs shown in Fig. 5.4(a) and (b). Consider a slight variation of τ_1 in which the variable y additionally points-to a node u_3.

In this case, the parameter node n_2 of τ_2 will have to be replaced by two nodes u_1 and u_3 during the composition operation. Without any form of cloning, the statement at line [7] in procedure P_{τ_2}, shown in Fig. 5.4(b), would be replaced by two new statements $n_3 = u_1.next$ and $n_3 = u_3.next$. The node n_3 would then represent the targets of both $u_1.next$ and $u_3.next$. Though this wouldn't affect correctness, it may result in loss of precision as the targets of two possibly different references are collapsed in the composed transformer graph.

Thus, for the generality of the semantics, we compute the targets of the external edges (or field-reads) created during the composition operation using the parameter $Load_S$. In this example, the target of $u_1.next$ would be determined using $Load_S(u_1, next, n_3)$ and that of $u_3.next$ using $Load_S(u_3, next, n_3)$.

5.3 Simplifying the Transformer Graphs

5.3.1 The $Simplify_S$ Operation

We now describe the *Simplify* operation that reduces the number of vertices in a transformer graph without altering its concrete image. In a nutshell, the *Simplify* operation propagates the reads and writes on an external vertex w to the vertices in the transformer graphs that represent a subset of concrete objects represented by w.

Let $\tau = (\mathsf{EV}, \mathsf{EE}, \sigma_{in}, \mathsf{IV}, \mathsf{IE}, \sigma, \rightsquigarrow)$ be a transformer graph. Say there is an external edge $\langle u, f, w\rangle \in \mathsf{EE}$, an internal edge $\langle u, f, x\rangle \in \mathsf{IE}$ and the internal edge may happen before the external edge (i.e, write happens before the read). In this case, the values of x *may flow* to w. Clearly, w represents a superset of concrete objects represented by x. Hence, a write (or read) on w can be considered as a write (or read) on x. That

$$x \in \mathsf{EV} \cup \mathsf{IV} \Rightarrow x \in \mathit{Incl}(x) \tag{5.10}$$

$$\left\{ \begin{array}{c} \langle u, f, v\rangle \in \mathsf{IE}, \langle u', f, v'\rangle \in \mathsf{EE}, \\ \mathit{Incl}(u) \cap \mathit{Incl}(u') \neq \emptyset, \\ \langle u, f, v\rangle \rightsquigarrow \langle u', f, v'\rangle \end{array} \right\} \Rightarrow \mathit{Incl}(v) \subseteq \mathit{Incl}(v') \tag{5.11}$$

$$\langle u, f, v\rangle \in \mathsf{EE}, a \in \mathit{Incl}(u) \Rightarrow \mathit{Load}_S(a, f, v) \in \mathit{Incl}(v) \tag{5.12}$$

Figure 5.7: The function *Incl* (for a transformer graph $(\mathsf{EV}, \mathsf{EE}, \sigma_{in}, \mathsf{IV}, \mathsf{IE}, \sigma, \rightsquigarrow)$) is defined as the least solution satisfying the above constraints.

is, $\langle w, f, y\rangle \in \mathsf{IE}$ (or EE) implies $\langle x, f, y\rangle \in \mathsf{IE}$ (or EE), respectively. The function $\mathit{Incl} : \mathsf{V} \mapsto \mathsf{V}$, defined in Fig. 5.7, maps every vertex w to the set of vertices whose values may flow to w. We clone the external vertices during *Simplify* (as shown in Fig. 5.7) for the same reasons described while presenting the composition operation in section 5.2.1.

$$\begin{aligned} \mathit{Simplify}_S(\tau) &= \mathit{removeNonEscaping}(\tau'), \text{ where} \\ \tau' &= (\mathsf{EV} \cup \{v \mid \langle u, f, v\rangle \in \mathsf{EE}'\}, \mathsf{IV}, \sigma_{in}, \mathsf{EE}', \mathit{trans}_{IE}(\mathsf{IE}, \mathit{Incl}), \\ &\qquad \mathit{trans}_{Sigma}(\sigma, \mathit{Incl}), \mathit{trans}_{HB}(\rightsquigarrow, \mathit{Incl})) \\ \mathsf{EE}' &= \mathit{trans}_{EE}(\mathsf{EE}, \mathit{Incl}) \end{aligned}$$

where, the *trans* operations, defined in section 5.2.1, apply the mapping *Incl* point-wise on each of the vertices in a given component of a transformer graph. We now describe the operation *removeNonEscaping*.

The operation *removeNonEscaping*, formally defined in Fig. 5.8, performs the actual simplification by removing edges and vertices from a transformer graph. We say that a vertex (internal or external) in a transformer graph τ is non-escaping if it does not belong to $\mathit{Escaping}(\tau)$. Given a transformer graph τ, the operation *removeNonEscaping* removes external edges starting from non-escaping vertices and external vertices that do not have any external edges ending at them. When a vertex is removed from a transformer graph, all edges (internal or external) that start or end at the vertex would also be removed from the transformer graph.

$$
\begin{aligned}
&\mathit{removeNonEscaping}(\mathsf{EV}, \mathsf{EE}, \sigma_{in}, \mathsf{IV}, \mathsf{IE}, \sigma, \rightsquigarrow) = \\
&\quad \text{let } \mathsf{EE}_{un} = \{\langle u, f, v\rangle \in \mathsf{EE} \mid u \notin \mathit{Escaping}(\tau)\} \text{ in} \\
&\quad \text{let } \mathsf{EV}_{un} = \{w \in \mathsf{EV} \mid \neg\exists \langle u, f, w\rangle \in (\mathsf{EE} \setminus \mathsf{EE}_{un})\} \text{ in} \\
&\quad \text{let } \mathsf{IE}_{un} = \{\langle u, f, v\rangle \in \mathsf{IE} \mid \text{ u (or) v belongs to } (\mathsf{EV}_{un} \setminus \mathsf{IV})\} \text{ in} \\
&\quad \text{let } \sigma' = \lambda x.\sigma(x) \setminus (\mathsf{EV}_{un} \setminus \mathsf{IV}) \text{ in} \\
&\quad \text{let } \rightsquigarrow' = \rightsquigarrow \setminus \{(ie, ee) \in \rightsquigarrow \mid ie \in \mathsf{IE}_{un} \vee ee \in \mathsf{EE}_{un}\} \text{ in} \\
&\quad (\mathsf{EV} \setminus \mathsf{EV}_{un}, \mathsf{EE} \setminus \mathsf{EE}_{un}, \sigma_{in}, \mathsf{IV}, \mathsf{IE} \setminus \mathsf{IE}_{un}, \sigma', \rightsquigarrow')
\end{aligned}
$$

Figure 5.8: Definition of the function $\mathit{removeNonEscaping} : \mathcal{A}_{\mathcal{G}} \mapsto \mathcal{A}_{\mathcal{G}}$.

Correctness of the *Simplify* Operation The correctness of the simplify operation is formalized in Theorem 5.6. The lemma states the concrete images of a transformer graph τ and $\mathit{Simplify}_S(\tau)$ are equal. This implies that the edges and vertices removed by the *removeNonEscaping* operation are redundant. Let w be a vertex read from a non-escaping vertex x i.e, $\langle x, f, w\rangle \in \mathsf{EE}$. The proof of the Theorem 5.6 establishes that, under all contexts, the concrete objects that w may represent is the union of all the concrete objects represented by the vertices in $\mathit{Incl}(w)$ that are escaping. This implies that the external edge $\langle x, f, w\rangle$ is redundant as every internal and external edge incident on w are translated to every vertex in $\mathit{Incl}(w)$.

An external vertex that is not the target of any external edge can be removed as such external vertices will not represent any concrete object by the definition of γ_T. (The mapping η computed during the concretization operation will always map them to empty sets.)

However, it is to be noted that a similar simplification cannot be applied to external edges that emanate from an escaping internal vertex, though an internal vertex represents objects allocated within the analysed code fragment. Let w be a vertex read from an internal vertex. A pertinent question to ask is whether $\mathit{Incl}(w)$ include all vertices whose values may flow to w?

Fig. 5.9 shows that this is not always the case. Consider the *Incl* mapping computed for the transformer graph shown at the right side

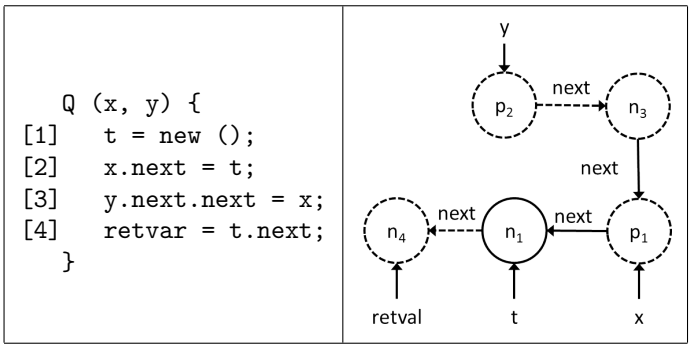


Figure 5.9: A program illustrating the effect of aliasing in the input state on internal escaping vertices.

of Fig. 5.9. The *Incl* function would map the node n_4 (which is a vertex read from an internal vertex) to only itself i.e, $Incl(n_4) = \{n_4\}$. Suppose that in some calling context the parameters `x` and `y` of the procedure Q are aliases. The field-write at line [3] would actually update the `next` field of the object allocated inside Q. The subsequent read of the `next` field of the newly allocated object will get the value written at line [3] which is the parameter object p_1. Therefore, p_1 flows to n_4 if `x` and `y` alias in the calling context.

This example illustrates that the values read from escaping vertices are dependent on the calling contexts even if they are allocated inside the analysed procedure, implying that external edges on escaping vertices ought to be preserved in the transformer graphs.

5.3.2 Abstract Garbage Collection

Analogous to the function GC_c used in the concrete semantics, the function $GC_{\mathcal{G}}$ used in the abstract semantics equations trims the transformer graphs by removing internal vertices (and the edges incident on the vertices) that are not reachable from the variables in the program and from the prestate.

Fig. 5.10 shows the formal definition of $GC_{\mathcal{G}}$. This operation removes all internal vertices that are unreachable from the variables, the external vertices and *vertices with external edges starting from them.*

$$
\begin{array}{l}
GC_{\mathcal{G}}(\tau) = \\
\quad \text{let } S = \hat{\sigma}(\mathit{Vars}) \cup \mathsf{EV} \cup \{u \mid \langle u, f, x\rangle \in \mathsf{EE}\} \\
\quad \text{let } \mathsf{IV}_{un} = \{x \in \mathsf{IV} \mid \neg\exists y \in S.\ x \text{ is reachable from y }\} \text{ in} \\
\quad \text{let } \mathsf{E}_{un} = \{\langle u, f, v\rangle \in \mathsf{IE} \cup \mathsf{EE} \mid \text{ u (or) v belongs to } \mathsf{IV}_{un}\} \text{ in} \\
\quad \text{let } \sigma' = \lambda x.\sigma(x) \setminus \mathsf{IV}_{un} \text{ in} \\
\quad \text{let } \rightsquigarrow' = \rightsquigarrow \setminus \{(ie, ee) \in \rightsquigarrow \mid ie \text{ or } ee \text{ belongs to } \mathsf{E}_{un}\} \text{ in} \\
\quad (\mathsf{EV}, \mathsf{EE} \setminus \mathsf{E}_{un}, \sigma_{in}, \mathsf{IV} \setminus \mathsf{IV}_{un}, \mathsf{IE} \setminus \mathsf{E}_{un}, \sigma', \rightsquigarrow')
\end{array}
$$

Figure 5.10: The definition of the abstract garbage collection operation $GC_{\mathcal{G}} : \mathcal{A}_{\mathcal{G}} \mapsto \mathcal{A}_{\mathcal{G}}$. In the figure, $\tau = (\mathsf{EV}, \mathsf{EE}, \sigma_{in}, \mathsf{IV}, \mathsf{IE}, \sigma, \rightsquigarrow)$.

Theorem 5.7 establishes the correctness of the garbage collection operation. In particular, it proves that the set IV_{un}, the set of vertices unreachable from the set S, represent concrete objects that are unreachable from the prestate and the variables in the program.

Recall that when the *Simplify* operation is performed on a transformer graph, the external edges starting from non-escaping vertices and external vertices that are not targets of external edges are removed from the transformer graphs. Therefore, applying $GC_{\mathcal{G}}$ on a "simplified" transformer graph will remove all non-escaping internal vertices that are unreachable from the variables. Many instances of the framework e.g. [Lattner et al., 2007], [Salcianu and Rinard, 2005], that maintain a (partially) simplified transformer graphs at every program point as explained later, adopt this definition for the garbage collection operation. However, $GC_{\mathcal{G}}$ defined above is more general and is applicable to any transformer graph in $\mathcal{A}_{\mathcal{G}}$.

5.4 Correctness and Termination of the Framework

We now state and prove a few insightful lemmas that help in establishing the correctness of the abstract semantics. As usual, we say that a concrete value $f \in \mathcal{C}$ is *correctly represented* by an abstract value $\tau \in \mathcal{A}_{\mathcal{G}}$, denoted $f \sim \tau$, iff $f \sqsubseteq_c \gamma_T(\tau)$. For brevity, we only provide

proof sketches and also completely omit proofs when they are straight forward to derive from the definitions.

In most of the proofs, we use induction over the computation of $\eta[\![\tau, g_c,]\!]$ which is defined as the least solution of a set of recursive implications (4.3)–(4.6) (see section 4.2 for more details) which naturally lends itself to an inductive proof structure. To prove that η satisfies a claim, we hypothesize that the claim holds in the antecedent of each rule and establish that the claim holds in the consequent of the rule.

Lemma 5.1. (First Escape Lemma) Let τ be a transformer graph such that every external edge has an escaping source vertex i.e. $\langle u, f, v\rangle \in \mathsf{EE} \implies u \in \mathit{Escaping}(\tau)$. Let $g_c = (\mathsf{V}_c, \mathsf{E}_c, \sigma_c)$ be a concrete graph. If x is a vertex in τ then $y \in \eta(x) \wedge y \in \mathsf{V}_c \implies x \in \mathit{Escaping}(\tau)$.

In simple words, the lemma states that, when all the external edges on the transformer graph have only escaping source vertices (the precondition), if a vertex x represents a concrete object (in some context) then x is an escaping vertex i.e. x is transitively reachable from the parameters (or globals).

Proof. We prove this by induction on the computation of η. It is easy to see that in each of the four rules when the claim holds for η in the antecedent, it also holds in the consequent. □

Lemma 5.2. (Second Escape Lemma) Let τ be a transformer graph satisfying the preconditions of the first escape lemma. Let $g_c = (\mathsf{V}_c, \mathsf{E}_c, \sigma_c)$ be a concrete graph. For any pair of vertices x, y in τ, $y \in \eta(x) \wedge y \neq x \implies x, y \in \mathit{Escaping}(\tau)$.

In simple words, the lemma states that, when the preconditions hold, if a vertex x represents a vertex other than itself (in some context) then it must be escaping. The contrapositive form of the above statement is more intuitive. It states that if a vertex is non-escaping then it represents only itself and is not a placeholder for any other vertex.

Proof. Induction on the computation of η. The only non-trivial case is establishing that the claim holds for the *alias rule* i.e. (4.6). Consider the antecedent of the alias rule given below.

$$\langle u, f, x\rangle \in \mathsf{EE}, \langle r, f, s\rangle \in \mathsf{IE}, \eta(u) \cap \eta(r) \neq \emptyset \tag{5.13}$$

The constraint on $\rightsquigarrow$ is omitted as it is unimportant here. We need to establish that for all vertices $y \in \eta(s)$, if $y \neq x$ then y escapes (as y would be added to $\eta(x)$ by the consequent).

If $y \neq s$ then y escapes by hypothesis. Say $y = s$. By (5.13), $\langle r, f, y\rangle \in \mathsf{IE}$. If r escapes then, by the definition of *Escaping*, y escapes as we have hypothesized that there is an internal edge from r to s (which is same as y). Say $r \notin \mathit{Escaping}(\tau)$. We will now show that this case is not possible. Let $p = \eta(u) \cap \eta(r)$. If $r \neq p$ then r escapes by hypothesis. Therefore, the only possibility is $r = p$. Hence, $r \in \eta(u)$. Again, if $r \neq u$ then r escapes by hypothesis. Therefore, r must be equal to u. By (5.13), $\langle r, f, x\rangle \in \mathsf{EE}$. However, by the prerequisite on the transformer graph, r escapes, which is a contradiction to the assumption that $r \notin \mathit{Escaping}(\tau)$. Hence, $r \notin \mathit{Escaping}(\tau)$ is not possible. □

Corollary 5.3. Let τ be a transformer graph satisfying the conditions of first escape lemma. Let g_c be a concrete graph. For any vertices x, y in the transformer graph,

$$\eta(x) \cap \eta(y) \neq \emptyset \wedge x \neq y \implies x, y \in \mathit{Escaping}(\tau)$$

Proof. Let $r = \eta(x) \cap \eta(y)$. If $r \in \mathsf{V}_c$ or r is different from x and y, x and y both escape by the first and second escape lemmas. Say $r = x$. $x \in \eta(y)$ and $x \neq y$ implies, by the above lemma, that x and y escape. Similarly, if $r = y$ and $x \neq y$, x and y escape. □

Lemma 5.4. For every primitive statement S, if $f \sim \tau$ then $f \circ [\![S]\!]_c \sim \mathcal{F}_{\mathcal{G}}(S)(\tau)$, where $\mathcal{F}_{\mathcal{G}}(S)$ is the transfer function of a primitive statement defined in Fig. 5.2.

Proof. $f \sim \tau$ implies that $f \sqsubseteq_c \gamma_T(\tau)$. By the definition of γ_T, for any concrete graph $g_c \in \mathbb{G}_c$, every concrete graph in $f(g_c)$ embeds in $\tau\langle g_c\rangle$. Therefore, it suffices to show that for any $g \in \mathbb{G}_c$ such that $g \preceq \tau\langle g_c\rangle$, $[\![S]\!]_c(g) \preceq \tau_{out}\langle g_c\rangle$, where $\tau_{out} = \mathcal{F}_{\mathcal{G}}(S)(\tau)$. This can be proved from the definitions of $[\![S]\!]_c$, $\mathcal{F}_{\mathcal{G}}(S)$, $\tau\langle g_c\rangle$, and by induction on the computation of $\eta[\![\tau, g_c]\!]$. We omit a detailed proof for brevity. □

Lemma 5.5. if $f_1 \sim \tau_1$ and $f_2 \sim \tau_2$, then $f_1 \circ f_2 \sim \tau_2\langle\!\langle\tau_1\rangle\!\rangle_S$.

Proof. Let $g_c \in \mathbb{G}_c$ be a concrete graph. $f_1 \sim \tau_1$ implies that every concrete graph in $f_1(g_c)$ embeds in $\tau_1\langle g_c\rangle$. Similarly, every concrete graph in $f_2(g_c)$ embeds in $\tau_2\langle g_c\rangle$. Hence, every graph in $f_2(f_1(g_c))$ will embed in $\tau_2\langle g\rangle$ for some concrete graph g such that $g \preceq \tau_1\langle g_c\rangle$. Hence, it suffices to show that for any $\{g, g'\} \subseteq \mathbb{G}_c$ such that $g \preceq \tau_1\langle g_c\rangle$ and $g' \preceq \tau_2\langle g\rangle$, $g' \preceq \tau_{out}\langle g_c\rangle$, where $\tau_{out} = \tau_2\langle\!\langle\tau_1\rangle\!\rangle_S$. This can be proved from the definitions of $\langle\!\langle\rangle\!\rangle_S$, $\langle\rangle$, and by induction on the computation of $\eta[\![\tau_2, g]\!]$. We omit a detailed proof for brevity. □

5.4.1 Correctness of the *Simplify* operation

Theorem 5.6. Let S be any statement and $\tau \in \mathcal{A}_{\mathcal{G}}$ be a transformer graph. $\gamma_T(\tau) = \gamma_T(\mathit{Simplify}_S(\tau))$

Proof. Let τ' denote $\mathit{Simplify}(\tau)$. The proof of this lemma is quite involved. The proof consists of two parts, in the first part we show that $\gamma_T(\tau')$ is an over-approximation of $\gamma_T(\tau)$ (i.e., $\gamma_T(\tau) \sqsubseteq_c \gamma_T(\tau')$) and in the second part we prove the converse.

(Part-I): For every input graph g_c, γ_T constructs an abstract graph $\tau\langle g_c\rangle$ representing the possible output concrete states. Every external or internal edge in τ is primarily used in identifying and adding edges to the abstract graph $\tau\langle g_c\rangle$ as discussed in section 4.2. To prove that τ' over-approximates τ it suffices to show that the edges removed by *Simplify* do not matter, that is, for every vertex u in τ there exists a vertex in τ' that adds all the edges added by u to the output abstract graph $\tau'\langle g_c\rangle$. Given this, it is easy to see that $\tau\langle g_c\rangle \sqsubseteq_{co} \tau'\langle g_c\rangle$ (which implies the claim). The function *Incl* used by *Simplify* translates all the edges on a vertex u to every vertex in $\mathit{Incl}(u)$. Hence, it suffices to show that, for every node x that u represents in τ, there exists at least one vertex belonging to both $\mathit{Incl}(u)$ and τ' that represents x, which is formally stated below and pictorially illustrated in Fig. 5.11. In Fig. 5.11 and in the sequel, η denotes $\eta[\![\tau, g_c]\!](u)$ and η' denotes $\eta[\![\tau', g_c]\!]$ where $\tau' = \mathit{Simplify}(\tau)$.

Let V denote the vertices in τ.

$$u \in \mathsf{V}, x \in \eta(u) \implies x \in \hat{\eta'}(\mathit{Incl}(u))$$

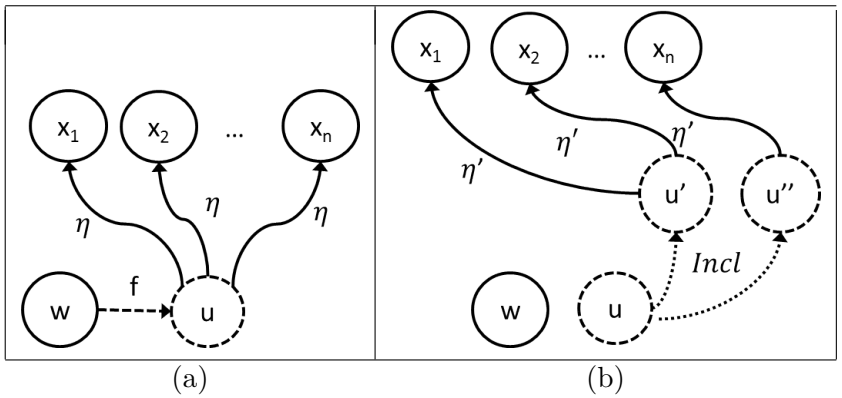


Figure 5.11: Pictorial illustration of the correctness of *Simplify* operation. Say u is a node removed by *Simplify*. Part (a) of the figure shows that the vertex u is found to represent the objects $x_1, \cdots, x_n$ in τ. Part (b) of the figure shows that *Incl* mapping computed by the *Simplify* operation will contain vertices (here, u', u'') which together will represent all the objects that u represented in τ.

This can be formally proved using induction on the computation of η and using the first and second escape lemmas.

(Part II): We now establish that $\gamma_T(\tau') \sqsubseteq_c \gamma_T(\tau)$. *Simplify* adds internal and external edges to τ which in turn could result in the addition of edges to the abstract graph $\tau'\langle g_c \rangle$.

Let $\langle u, f, w \rangle \in \mathsf{IE}'$. Say u is found to represent an object c in the concretization of τ' (i.e. $c \in \eta'(u)$) and w represents an object d. Clearly, $\langle c, f, d \rangle$ will be added to $\tau'\langle g_c \rangle$ (by construction). To prove the claim it suffices to establish that $\langle c, f, d \rangle$ would also be added to $\tau\langle g_c \rangle$. However, this would be implied when the following two conditions hold as explained below:

(a) For any $u, v \in \mathsf{V}$, $u \in \mathit{Incl}(v) \implies \eta(u) \subseteq \eta(v)$.
(b) For any $x \in \mathsf{V}'$, $\eta'(x) \subseteq \eta(x)$.

By the second condition, $c \in \eta'(u)$ implies that $c \in \eta(u)$. Similarly, $d \in \eta(w)$. Now say $\langle u, f, w \rangle \notin \mathsf{IE}$ i.e. the internal edge is added by *Simplify* (otherwise the claim directly follows). By the definition of *Simplify*, $\exists p, q \in \mathsf{V}$ s.t. $u \in \mathit{Incl}(p), w \in \mathit{Incl}(q)$, $\langle p, f, q \rangle \in \mathsf{IE}$. The first

condition implies that $\eta(u) \subseteq \eta(p)$ and $\eta(w) \subseteq \eta(q)$. Hence, $c \in \eta(p)$, $d \in \eta(q)$. Therefore, $\langle c, f, d\rangle$ would be added to $\tau\langle g_c\rangle$ as required.

Hence, it just suffices to establish that the conditions (a) and (b) hold. Condition (a) follows directly from the definition of *Incl* and the *alias rule* (4.6) of η computation. Condition (b) can be proved using induction on the computation of $\eta[\![\tau', g_c]\!]$ and using condition (a) (which is essential for proving that the claim holds for the constraints (4.5) and (4.6)). Details elided for brevity. □

5.4.2 Correctness of the $GC_{\mathcal{G}}$ operation

Theorem 5.7. If $f \sim \tau$ then $GC_c(f) \sim \tau$ and $GC_c(f) \sim GC_{\mathcal{G}}(\tau)$.

Proof. The first claim $GC_c(f) \sim \tau$ is relatively easier to establish. For any $g_c \in \mathbb{G}_c$, if $g_1 \in GC_c(f)(g_c)$ then there exists a $g_2 \in f(g_c)$ such that $g_1 \preceq g_2$, as GC_c only removes vertices and edges from the graphs in $f(g_c)$. Since $f \sim \tau$, $g_2 \preceq \tau\langle g_c\rangle$. Therefore, $g_1 \preceq g_2 \preceq \tau\langle g_c\rangle$. Hence, $g_1 \in \gamma(\tau)(g_c)$.

Consider the second part. Let $g_c = (\mathsf{V}_c, \mathsf{E}_c, \sigma_c)$ be a concrete graph. Let $\tau\langle g_c\rangle = (\mathsf{V}_r, \mathsf{E}_r, \sigma_r)$. Let η denote $\eta[\![\tau, g_c]\!]$. To prove this part it suffices to show that the vertices removed by $GC_{\mathcal{G}}$ in τ will only represent vertices that are unreachable from V_c and *Vars* in $\tau\langle g_c\rangle$ as those correspond to the vertices that would be removed by GC_c.

Claim: if u is not reachable from the set $S = \hat{\sigma}(\mathit{Vars}) \cup \mathsf{EV} \cup \{u \mid \langle u, f, x\rangle \in \mathsf{EE}\}$ then every vertex y in $\eta(u)$ is not reachable from V_c and $\hat{\sigma_r}(\mathit{Vars})$ in $\tau\langle g_c\rangle$.

Before we sketch the proof, we establish a unique property of the vertices unreachable from the set S. Let u be a vertex in τ that is unreachable from S. For any concrete graph g_c, $u \in \eta(u') \Rightarrow u = u'$. That is, no vertex other than u itself represents u. This precludes the existence of other vertices in τ representing u. This property can be proved, via induction on η, by establishing that no constraint other than the internal vertex rule (4.3) can add u to $\eta(u')$ for any vertex u'.

Now consider the claim. Let w be a vertex not reachable from S and $y \in \eta(w)$. Consider a path $x_1, \ldots, x_n$, where $x_n = y$, in $\tau\langle g_c\rangle$ ending at y. It suffices to show that no vertex in this path belongs to V_c or $\hat{\sigma_r}(\mathit{Vars})$.

Since w is not reachable from S, $w \in \mathsf{IV} \setminus \mathsf{EV}$. By the definition of η, $y = w$. Since there is an edge from x_{n-1} to y in E_r, by the definition of $\langle\rangle$, $\exists p, q \in \mathsf{V}$, $x_{n-1} \in \eta(p)$, $y \in \eta(q)$, $\langle p, f, q\rangle \in \mathsf{IE}$. Since, y (which is same as w) is not reachable from S, by the above property, $q = y$. Hence, $\langle p, f, y\rangle \in \mathsf{IE}$. Since y is not reachable from S so is p. Hence, $x_{n-1} = p$ and $p \in \mathsf{IV} \setminus \mathsf{EV}$. By induction, all of the vertices in the path are internal vertices and hence do not belong to V_c. Similarly, it can be shown that no vertex in the path belongs to $\hat{\sigma_r}(\mathit{Vars})$. □

Instantiating $(\mathcal{A}_\mathcal{G}, \mathcal{F}_\mathcal{G})$

An abstract semantics $(\mathcal{A}, \mathcal{F}_\mathcal{A})$ is an instance of $(\mathcal{A}_\mathcal{G}, \mathcal{F}_\mathcal{G})$ iff $\mathcal{A}$ is obtained by defining the set of abstract nodes N_a in $\mathcal{A}_\mathcal{G}$, and $\mathcal{F}_\mathcal{A}$ uses a set of semantic equations that belong to the family of equations represented by the Fig. 5.1 and defines the parameters Load_S, Alloc_S and *InitBind* suitably.

Later, in section 6 we present analyses that not only instantiate the parameters but also specialize the semantics of the framework. For such instances, the correctness and termination also depend on the additional specializations that they perform.

Abstract Fixed Point Computation

Given an instance $(\mathcal{A}, \mathcal{F}_\mathcal{A})$ of $(\mathcal{A}_\mathcal{G}, \mathcal{F}_\mathcal{G})$ with a set of abstract semantic equations it can be viewed as a single equation $\vartheta = F^\sharp(\vartheta)$, where $F^\sharp$ is a function from $\mathit{VE} \mapsto \mathcal{A}$ to itself. (VE denotes the set of vertices and edges in the control flow graph.) Let $\perp$ denote $\lambda x.(\emptyset, \emptyset, \lambda v.\emptyset, \emptyset, \emptyset, \lambda v.\emptyset, \emptyset)$. The analysis iteratively computes the sequence of values $F^{\sharp i}(\perp)$ and terminates when $F^{\sharp i}(\perp) = F^{\sharp i+1}(\perp)$. We define $[\![P]\!]^\sharp$ (the summary for a procedure P) to be the value of $\varphi_{\mathit{exit}(P)}$ in the final solution.

To prove the correctness of the analysis we need to establish that $F^\sharp$ is a sound approximation of $F^\natural$, which follows if the corresponding components of $F^\sharp$ are sound approximations of the corresponding components of $F^\natural$.

Lemma 5.8. The abstract semantics equations 5.1–5.4 is a sound abstraction of the concrete semantics equations 3.1–3.4.

a. $\lambda g.\{g\} \sim \tau_{id}$

b. For any primitive statement S, if $f \sim \tau$, then $f \circ [\![S]\!]_c \sim \mathcal{F}_{\mathcal{G}}(S)(\tau)$.

c. $\sqcup_{co}$ is a sound approximation of $\sqcup_c$: if $f_1 \sim \tau_1$ and $f_2 \sim \tau_2$, then $(f_1 \sqcup_c f_2) \sim (\tau_1 \sqcup_{co} \tau_2)$.

d. For any call statement S, if $f_1 \sim \tau_1$ and $f_2 \sim \tau_2$, then $f_1 \circ \mathit{Call}_S(f_2) \sim \mathcal{F}_{\mathcal{G}}(S)(\tau_1, \tau_2)$.

e. if $f \sim \tau$ then $GC_c(f) \sim \tau$.

Proof. The lemma directly follows from the Lemmas 5.4, 5.5, Theorem 5.7 and from the properties of join. □

Lemma 5.8, Theorem 5.6 and Theorem 5.7 imply the following soundness theorem in the standard way (e.g., see Proposition 4.3 of [Cousot and Cousot, 1992]).

Theorem 5.9. Let $[\![P]\!]^\sharp$ be the abstract summary of a procedure P computed by an abstract semantics that is an instantiation of $(\mathcal{A}_{\mathcal{G}}, \mathcal{F}_{\mathcal{G}})$. The computed procedure summaries are correct. For every procedure P, $[\![P]\!]^\natural \sim [\![P]\!]^\sharp$.

5.4.3 Termination

For ensuring termination, we require the parameters of our semantics to satisfy the following assumption.

Assumption 5.10. (a) N_a is finite. (b) For every program statement S, *InitBind*, Alloc_S and Load_S terminate on all inputs

The following lemmas establish the monotonicity of $F^\sharp$. We elide proofs of the following monotonicity claims as they are straight forward to derive from the definitions.

Lemma 5.11. For every statement S, $\mathcal{F}_{\mathcal{G}}(S)$ is monotonic with respect to $\sqsubseteq_{co}$:

a. If S is a primitive statement and if $\tau_1 \sqsubseteq_{co} \tau_2$, then $\mathcal{F}_{\mathcal{G}}(S)(\tau_1) \sqsubseteq_{co} \mathcal{F}_{\mathcal{G}}(S)(\tau_2)$.

b. If S is a procedure call and if $\tau_1 \sqsubseteq_{co} \tau_1'$ and $\tau_2 \sqsubseteq_{co} \tau_2'$ then $\mathcal{F}_{\mathcal{G}}(S)(\tau_1, \tau_2) \sqsubseteq_{co} \mathcal{F}_{\mathcal{G}}(S)(\tau_1', \tau_2')$. (The composition operation is also monotonic with respect to τ_1 and τ_2.)

Lemma 5.12. *Simplify*$_S$ operation is monotonic with respect to $\sqsubseteq_{co}$: if $\tau_1 \sqsubseteq_{co} \tau_2$ then *Simplify*$_S(\tau_1) \sqsubseteq_{co}$ *Simplify*$_S(\tau_2)$.

Lemma 5.13. $GC_{\mathcal{G}}$ operation is monotonic with respect to $\sqsubseteq_{co}$: if $\tau_1 \sqsubseteq_{co} \tau_2$ then $GC_{\mathcal{G}}(\tau_1) \sqsubseteq_{co} GC_{\mathcal{G}}(\tau_2)$.

Theorem 5.14. If $(\mathcal{A}, \mathcal{F}_{\mathcal{A}})$ is a instance of $(\mathcal{A}_{\mathcal{G}}, \mathcal{F}_{\mathcal{G}})$ that satisfies Assumption 5.10 then the following conditions hold (*the ascending chain conditions*):

a. $\mathcal{A}$ has only finite ascending chains.

b. $\{F^{\sharp^i}(\bot), i \geq 0\}$ is an ascending chain in $VE \mapsto \mathcal{A}$.

c. $(\mathcal{A}, \mathcal{F}_{\mathcal{A}})$ terminates.

6

Specializations of the Framework

Specializations transform the framework $(\mathcal{A}_\mathcal{G}, \mathcal{F}_\mathcal{G})$ to abstract analyses that are less general in some aspects compared to $(\mathcal{A}_\mathcal{G}, \mathcal{F}_\mathcal{G})$. We classify the specializations considered in this section into three categories: *Simple instantiations*, *Restrictions* and *Abstractions*. Simple instantiations refer to the strategies for defining the parameters of the framework. Restrictions refer to specializations that restrict the abstract semantics $\mathcal{F}_\mathcal{G}$ to operate over a subset of $\mathcal{A}_\mathcal{G}$ satisfying specific properties. The restrictions we consider are lossless i.e, the restricted semantics and the original semantics produce equivalent results. Lossy abstractions of the framework are referred to as abstractions. We use the categorization only for explanatory purposes. From a theoretical perspective, the analyses produced by specializations are abstract interpretations [Cousot and Cousot, 1992] of the framework $(\mathcal{A}_\mathcal{G}, \mathcal{F}_\mathcal{G})$.

Multiple specializations discussed in this section can be applied in conjunction, provided their preconditions, if any, are satisfied. We refer to an analysis obtained by applying one or more specializations to $(\mathcal{A}_\mathcal{G}, \mathcal{F}_\mathcal{G})$ as an instance of the framework. We eventually show that the analyses: [Whaley and Rinard, 1999], [Cheng and Hwu, 2000], [Lattner et al., 2007] and [Buss et al., 2008] are instances of the framework.

6.1 Instantiations

In this section, we discuss the instantiations of the parameters of the framework that are used by the instances that we study.

6.1.1 Node Naming Strategies and Node Allocation Functions

Creation site based naming strategy (*Creation-site-naming*) This naming strategy names the vertices created during a statement with the label of the statement (which uniquely identifies the statement). Formally, the set of abstract nodes is defined as

$$N_a = \{n_x \mid x \in \mathit{Params} \cup \mathit{Globals} \cup \mathit{Labels}\}$$

The parameters of the abstract semantics are defined as follows.

$$\mathit{InitBind}(var) = \{n_{var}\}$$

$$\mathit{Alloc}_S(x) = \begin{cases} \{n_\ell\} & \text{if } S \text{ is } \ell : v_1 = \mathit{new}\ C \text{ or if } x \text{ is unspecified} \\ \{x\} & \mathit{Otherwise} \end{cases}$$

$$\mathit{Load}_S(u, f, x) = \begin{cases} \{n_\ell\} & \text{if } S \text{ is } \ell : v_1 = v_2.f \text{ or if } x \text{ is unspecified} \\ \{x\} & \mathit{Otherwise} \end{cases}$$

Creation site based naming with reuse (*Creation-site-naming-reuse*) This strategy is a variant of *Creation-site-naming* that uses a slightly different definition for the *Load* parameter in order to reuse the external vertices whenever possible.

$$\mathit{Load}_S(u, f, x) = \begin{cases} \{n_\ell\} & \text{if } x \text{ is unspecified} \\ \{x\} & \mathit{Otherwise} \end{cases}$$

Creation site based naming with cloning (*Creation-site-naming–cloning*) This strategy names every vertex using a pair: the label of the creation-site of the vertex and a *call-string*, which is a (possibly empty) sequence of labels of call statements. The *call-string* associated with a vertex at any given program point denotes the sequence of call statements that lead to the creation-site of the vertex from that

program point. This is similar to the context-sensitive object naming schemes employed by several top-down pointer analyses [Smaragdakis et al., 2011]. The set of abstract nodes are defined as:

$$N_a = \{(cs, n_x) \mid cs \in \bigcup_{i \leq n} \mathit{Labels}^i, x \in (\mathit{Params} \cup \mathit{Globals} \cup \mathit{Labels})\}$$

The length of the call-strings is bounded by n – the size of the set *Labels*. The definition of the parameters of the abstract semantics are shown below. The definitions ensure that a statement label appears at most once in any call-string generated during the analysis. (We use $cs \cdot \ell$ to denote the concatenation of label ℓ with the call-string cs, and $\ell \notin cs$ to denote that the label ℓ does not occur in the call-string cs.)

$$\mathit{InitBind}(var) = \{n_{var}\}$$

$$\mathit{Alloc}_S(x) = \begin{cases} \{(\emptyset, n_\ell)\} & \text{if } S \text{ is } \ell : v_1 = new\ C \\ & \quad \text{or if } x \text{ is unspecified} \\ \{(cs \cdot \ell, n_{\ell'})\} & \text{if } S \text{ is } \ell : Call\ P, \\ & \quad x = (cs, n_{\ell'}) \text{ and } \ell \notin cs \\ \{x\} & Otherwise \end{cases}$$

$$\mathit{Load}_S(u, f, x) = \begin{cases} \{(\emptyset, n_\ell)\} & \text{if } S \text{ is } \ell : v_1 = v_2.f \\ & \quad \text{or if } x \text{ is unspecified} \\ \{(cs \cdot \ell, n_{\ell'})\} & \text{if } S \text{ is } \ell : Call\ P \\ & \quad x = (cs, n_{\ell'}) \text{ and } \ell \notin cs \\ \{x\} & Otherwise \end{cases}$$

Note that the abstract semantics $\mathcal{F}_\mathcal{G}$ when instantiated as above renames the abstract vertices along all acyclic call-paths in the program, which [Lattner et al., 2007], [Liang and Harrold, 2001] refer to as full *heap cloning*.

Access-path based naming strategy (*Accesspath-based-naming*) In this naming strategy the abstract vertices in the transformer graphs are named using *access-paths* which are sequences of field dereferences starting from variables or statement labels. An interesting aspect of

this naming strategy is that it is possible to reconstruct the external edges in a transformer from the names of the vertices. Intuitively, an external vertex in a transformer graph corresponds to an object that is transitively read from parameters, global variables or internal vertices, through a sequence of field dereferences. In other words, every external vertex corresponds to one or more access-paths. The following naming strategy makes this relationship explicit.

Access-Paths Let $AP = RAP \cup LAP$, where $RAP = \{\delta^* \mid \delta \in (LAP \cup \mathit{Labels})\}$, $LAP = \mathit{Vars} \cup (RAP \times \mathit{Fields})$, denote the set of all access-paths that are in one of the following forms: v, $v*$, ℓ^*, $x^*.f_1^*.\ldots.f_{n-1}^*f_n$, $x^*.f_1^*.\ldots.f_n^*$, where v is a variable, ℓ is a statement label (of an allocation site), $x \in (\mathit{Vars} \cup \mathit{Labels})$ and $f_1, \ldots, f_n \in \mathit{Fields}$. We use $*$ to distinguish between objects and pointers. If an access-path ends with a star or if it is a label ℓ then it represents an object.

We use a subscript r to denote the set of access-paths of length at most r. For example, LAP_2 denotes the set of access-paths in LAP of length at most 2.

In this naming strategy, N_a is defined as RAP_r where r is some predefined integer value greater than one. The parameters of the abstract semantics are defined as follows.

$$\mathit{InitBind}(var) = \{var\}$$

$$\mathit{Alloc}_S(x) = \begin{cases} \{\ell\} & \text{if } S \text{ is } \ell : v_1 = \mathit{new}\ C \\ & \quad \text{or if } x \text{ is unspecified} \\ \{x\} & \mathit{Otherwise} \end{cases}$$

$$\mathit{Load}_S(\delta, f, x) = \{\delta.f^*\}$$

Note that in the above definition Load_S is completely independent of the statement S and depends only on the dereferenced vertex. This is an important characteristic of this naming strategy. For simplicity, the definition of *Alloc* presented here does not clone the internal vertices during call statements. It is straightforward to extend the above definition to support cloning by associating call-strings with names of the vertices as in the *Creation-site-naming-cloning* strategy.

When the parameters are defined as above, the abstract transfer functions preserve the following invariants. (It is straightforward to plug in the definition of the parameters in the abstract transfer functions and verify the following.)

(a) Every internal node is named by a statement label such as ℓ, as a result of the definition of *Alloc*.

(b) Every external node is named by an access-path that is not a statement label, which is a consequence of the definition of *Load*.

(c) The presence of a node with name $\delta.f^*$ implies that there exists an external edge from δ to $\delta.f^*$ and vice versa. This follows from the definition of *Load*.

These invariants enable the following interesting simplifications to the abstract domain. Since the node names implicitly record if a node is external or internal, we can drop internal and external vertex components from the transformer graphs. (Note that isolated nodes can be encoded using dummy edges that end at *null*). We can also omit the external edge component from the transformer graphs as external edges are implicitly represented by the names of the nodes.

Instantiating the abstract semantics with *Simplify* and $GC_{\mathcal{G}}$ operations In most of the instances that we study the *Simplify* and $GC_{\mathcal{G}}$ operations are lazily performed only at the exit point of a procedure. We refer to this strategy as lazy simplification and lazy garbage collection (*Lazy-simplification* and *Lazy-garbage-collection*), respectively.

6.2 Restrictions

In this section, we discuss the specializations of the abstract semantics obtained by restricting the transfer functions $\mathcal{F}_{\mathcal{G}}$ to a subset of $\mathcal{A}_{\mathcal{G}}$. All the specializations we consider in this section are lossless, i.e, they are semantically equivalent to the abstract semantics $(\mathcal{A}_{\mathcal{G}}, \mathcal{F}_{\mathcal{G}})$.

Omitting the σ_{in} component (*Omitting-σ_{in}*) This specialization applies only when N_a includes a special set of nodes subscripted by parameter and global variable names and *InitBind* is defined as $\lambda x.\{n_x\}$.

Consider the set that contains $\perp$ (the empty transformer graph) and the transformer graphs where σ_{in} is the function $\lambda x.\{n_x\}$. The transfer functions $\mathcal{F}_{\mathcal{G}}$ are closed with respect to this set of transformer graphs. By definition of $\mathcal{F}_{\mathcal{G}}$, the σ_{in} component remains constant.

This specialization omits the component σ_{in} and modifies the transfer functions so that they use n_x instead of $\sigma_{in}(x)$. Clearly, the specialized semantics is correct and terminates if the original semantics does.

6.2.1 Incorporating partial simplification

This specialization, abbreviated as *Partial-eager-simplification*, restricts the abstract semantics of the framework ($\mathcal{F}_{\mathcal{G}}$) to operate over an abstract domain $\mathcal{A}_{\mathcal{I}} \subseteq \mathcal{A}_{\mathcal{G}}$. Transformer graphs belonging to $\mathcal{A}_{\mathcal{I}}$ do not contain any external edges with a non-escaping source vertex. Such edges are not essential: e.g., given any transformer $\tau \in \mathcal{A}_{\mathcal{G}}$, the *Simplify* operation presented earlier returns an equivalent transformer that has no such edges. In other words, $\mathit{Simplify}(\tau) \in \mathcal{A}_{\mathcal{I}}$.

Two of the analysis instances we study are based on the restricted domain $\mathcal{A}_{\mathcal{I}}$. The primary reason for working with the domain $\mathcal{A}_{\mathcal{I}}$ is efficiency, as these graphs are more compact.

It turns out that the abstract transfer functions $\mathcal{F}_{\mathcal{G}}(S)$ is closed with respect to $\mathcal{A}_{\mathcal{I}}$ for most types of statements S. We need to adapt the definition of $\mathcal{F}_{\mathcal{G}}(S)$ only for field-read statements and call statements. One way of doing this would be to apply *Simplify* as the last step (to the transformer graph produced by $\mathcal{F}_{\mathcal{G}}(S)$). Instead, we utilize a *partial simplification* which is more efficient. The *Simplify* operation used by $(\mathcal{A}_{\mathcal{G}}, \mathcal{F}_{\mathcal{G}})$ removes external vertices and edges from a transformer graph and also adds new edges (that were implicitly represented in the original graph). The partial simplification we utilize removes all external edges from non-escaping nodes, but avoids adding some of the edges added by *Simplify*. Appendix A formally presents this specialization and proves its correctness and termination.

6.3 Abstractions

Omitting the *may happens before* component (*Omitting-happen-before*) This abstraction omits the may happens before component of the transformer graphs that tracks the ordering between the internal and the external edges. This abstraction modifies the transfer functions of the framework $\mathcal{F}_{\mathcal{G}}$ so that they conservatively assume that any internal edge may happen before any external edge. All instances that we consider except the flow-aware analysis [Buss et al., 2008] omit the happens before component. The flow-aware analysis tracks an approximation of the happens before relation.

Ignoring the distinction between internal and external edges and vertices (*No-internal-external-distinction*) This abstraction compresses the abstract domain by unifying the following components: internal and external edges, internal and external vertices, and the initial and final variable mappings (σ_{in} and σ). Every internal edge (or vertex) is considered an external edge (or vertex) and vice-versa. This essentially reduces the seven-tuple transformer graphs to quadruples of the form $(\mathsf{V}, \mathsf{E}, \sigma, \rightsquigarrow)$, which actually corresponds to the transformer graph $(\mathsf{V}, \mathsf{E}, \sigma, \mathsf{V}, \mathsf{E}, \sigma, \rightsquigarrow)$. The transfer functions are also simplified using this property.

Flow Insensitivity Some instances that we consider perform a flow-insensitive analysis. Flow-insensitivity is an abstraction applied to the control-flow graph of the procedures that ignores the control-flow between the program statements. Our framework or more generally any abstract interpretation formalization is parameterized by the control-flow graphs of the program. Therefore, flow-insensitivity requires no additional extensions to the abstract domain or semantics.

Nevertheless, the abstract semantics could be made more efficient when it is used in a flow-insensitive analysis. For example, we can avoid performing strong updates on variables. These optimizations, though important, are conceptually straightforward.

This abstraction also makes the happens before component of the transformer graph obsolete because every statement may happen be-

fore every other statement in a flow-insensitive analysis. Therefore, this abstraction subsumes the *Omitting-happen-before* abstraction.

6.3.1 Node merging abstraction

This abstraction allows nodes in the transformer graphs to be merged together at arbitrary points during the analysis, as an efficiency heuristic, without sacrificing correctness or termination. This is formalized in Appendix B.

Informally, node merging is an operation that replaces a set of nodes $\{n_1, n_2 \ldots n_m\}$ by a single node n_{rep} such that any predecessor or successor of the nodes $n_1, n_2, \ldots, n_m$ becomes, respectively, a predecessor or successor of n_{rep}. While merging nodes seems like a natural heuristic for improving efficiency, it does introduce some subtle issues and challenges. The intuition for merging nodes arises from their use in the context of heap analyses where graphs represent sets of concrete states. However, in our context, graphs represent state transformers.

This abstraction introduces an equivalence relation on nodes (representing the nodes currently merged together) into the abstract domain and updates the transfer functions accordingly (see Appendix B). The resulting semantics (denoted as $\mathcal{F}_{\tilde{\mathcal{A}}}$) computes a transformer graph and equivalence relation pair at every program point as opposed to a single transformer graph computed by the abstract semantics $\mathcal{F}_{\mathcal{G}}$.

This abstraction also introduces a new parameter *NodesToMerge* (for each statement and program point) that is a function from transformer graphs to equivalence relations. The equivalence classes in the equivalence relation returned by *NodesToMerge*, at a statement or program point, specify the set of nodes that have to be collapsed after applying the transfer function of the statement or program point. Different node-merging strategies can be realized by appropriately instantiating the *NodesToMerge* parameter.

The correctness and termination of this abstraction is non-trivial and is formally established in Appendix B.

Realizing a unification-based analysis using node merging abstraction A unification-based analysis (such as [Steensgaard, 1996], [Das,

Stmt S	$NodesToMerge_S((\mathsf{EV}, \mathsf{EE}, \sigma_{in}, \mathsf{IV}, \mathsf{IE}, \sigma, \rightsquigarrow))$
$v_1 = v_2$	$[\{(x, y) \mid x \in \sigma(v_1), y \in \sigma(v_2)\}]$
$\ell : v = new\ C$	$[\{(x, n_\ell) \mid x \in \sigma(v)\}]$
$v_1.f = v_2$	$[\{(x, y) \mid \exists u \in \sigma(v_1), \langle u, f, x\rangle \in \mathsf{IE} \cup \mathsf{EE}, y \in \sigma(v_2)\}]$
$\ell : v_1 = v_2.f$	$[\{(x, y) \mid y \in \sigma(v_1), \exists u \in \sigma(v_2), \langle u, f, x\rangle \in \mathsf{IE} \cup \mathsf{EE}\}]$
$Call\ Q(args)$	$[\{(x, y) \mid \mathtt{V} \in args \cup Globals, x \in \sigma(\mathtt{V}), y \in \sigma(\mathtt{V})\}]$

$$[R] = lfp\ \lambda X.\ X^* \cup \bigcup_{(u,v)\in X, f\in Fields} Targets(u, f) \times Targets(v, f)$$

Figure 6.1: Definition of *NodesToMerge* for the statements of our language. X^* denotes the reflexive, symmetric, transitive closure of a relation X. $Targets(v, f)$ denotes the vertices that are the targets of the internal or external edges that start from the vertex v and labelled by the field f.

2000]) maintains equivalence classes of abstract nodes and tracks the points-to relations between the equivalence classes at every program point. Unification is typically used by top-down heap analyses that compute abstract shape graphs. But here, we would like to use unification in our bottom-up analysis that compute abstract state transformers.

A unification-based analysis treats the assignment statements in the program *bidirectionally*. For instance, given an assignment statement of the form $v_1 = v_2$, a unification-based analysis makes the targets of v_1 and v_2 equivalent instead of making the targets of v_1 a superset of those of v_2. Making two vertices equivalent in turn induces an equivalence between their targets. Therefore, the transfer function of an assignment statement transitively 'unions' (or merges) the equivalence classes of the targets of left-hand-side (LHS) and right-hand-side (RHS) of the assignment statement. This behavior can be simulated using the node-merging abstraction (discussed above) by appropriately defining the *NodesToMerge* parameter.

Fig. 6.1 formally defines the *NodesToMerge* parameter for every statement of our language that will enable a unification-based analysis.

As shown in Fig. 6.1, for every primitive statement, we record that the targets of LHS and RHS are equivalent which implies that their corresponding equivalence classes have to be merged.

The function [ ], defined in Fig. 6.1, computes the set of equivalences that are induced by a relation R representing a set of pairs of equivalent nodes. The induced equivalences are defined as follows. (a) Given a set of pairs of equivalent nodes X, every element in the reflexive, symmetric, transitive closure X^* are equivalent nodes. (b) Given two nodes that are equivalent, the targets of the external and internal edges starting from the nodes that are labelled by the same field are equivalent.

Consider the definition of *NodesToMerge* for procedure calls. Invoking a procedure may introduce new equivalences between the abstract nodes because of the assignments that happen within the procedure. The definition of *NodesToMerge* parameter presented in Fig. 6.1 captures all such equivalences.

7

Instances of the Framework

7.1 Overview of the Instances

In this section, we present an informal overview of the analyses that we would be formalizing as instances of the framework in the next section. Fig. 7.1 shows two procedures `listAdd` and `first` that manipulate a singly linked-list, and two clients `P` and `Q` of the procedures that perform a sequence of calls to `listAdd` and `first`. We use this example to illustrate each of the four instances and also highlight the differences in their precision and scalability.

Say we are interested in the contents of the lists manipulated by the procedures `P` and `Q`. We define content of a list as the set of nodes in the list. For example, the procedure `P` adds `x` to the contents of `l1`, `x` and `y` to the contents of `l2`, and `z` to the contents of `l3`.

Figures 7.2, 7.3 and 7.4 show the summaries computed by the four instances for the procedure `listAdd` and `P`. The internal edges and vertices are shown with solid lines, and the external edges and vertices are shown with dashed lines. All edges (internal and external) of all the transformer graphs shown in the figures are labelled by the field `next` which is not shown for clarity. The arrows from variables to nodes show the targets of the variables at the end of the procedure (σ).

```
   listAdd (l, n) {
1     while(l.next != null)
2        l = l.next
3     l.next = n
   }

   first (l) {
4     return l.next
   }
```

```
   P (l1, l2, x, y, z) {
5     fst = first(l1)
6     listAdd(l1, x)
7     listAdd(l2, x)
8     listAdd(l2, y)
9     l3 = new List()
10    listAdd(l3, z)
   }

   Q (l, x, y, z) {
11    l = new List()
12    listAdd(l, z)
13    P(l, l, x, y, z)
   }
```

Figure 7.1: A list manipulating program. The procedure `listAdd` adds a node to the end of a list. The procedure `first` returns the first element of the list. The procedures `P` and `Q` perform a sequence of `listAdd` and `first` operations. The variables `l3` and `fst` are global variables.

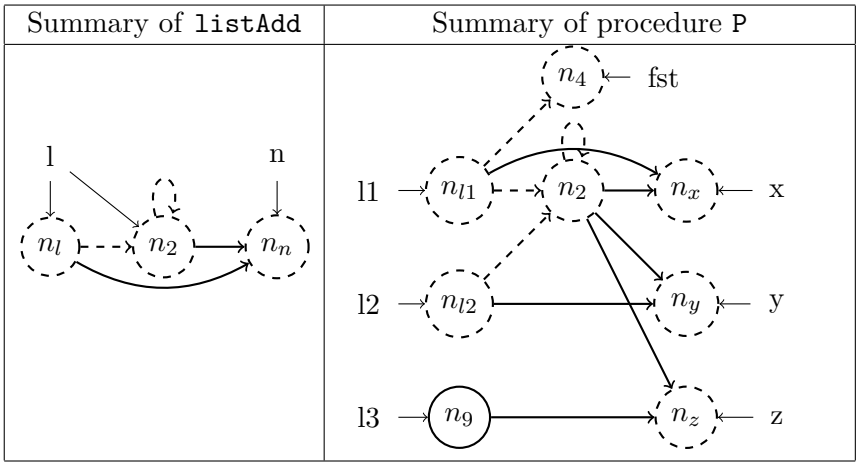


Figure 7.2: Summaries computed by the WSR analysis for the procedures `listAdd` and `P` shown in Fig. 7.1.

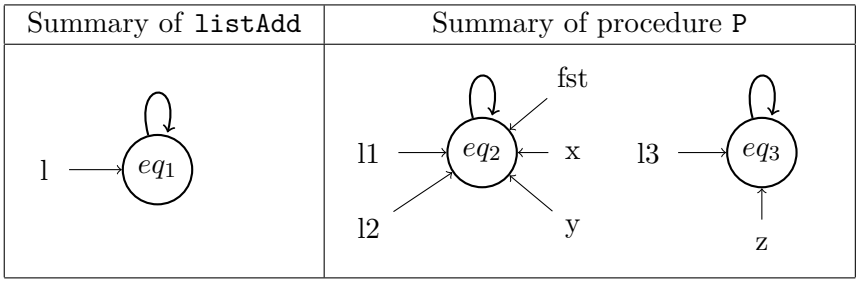


Figure 7.3: Summaries computed by the analysis proposed in [Lattner et al., 2007] for the procedures `listAdd` and `P` shown in Fig. 7.1.

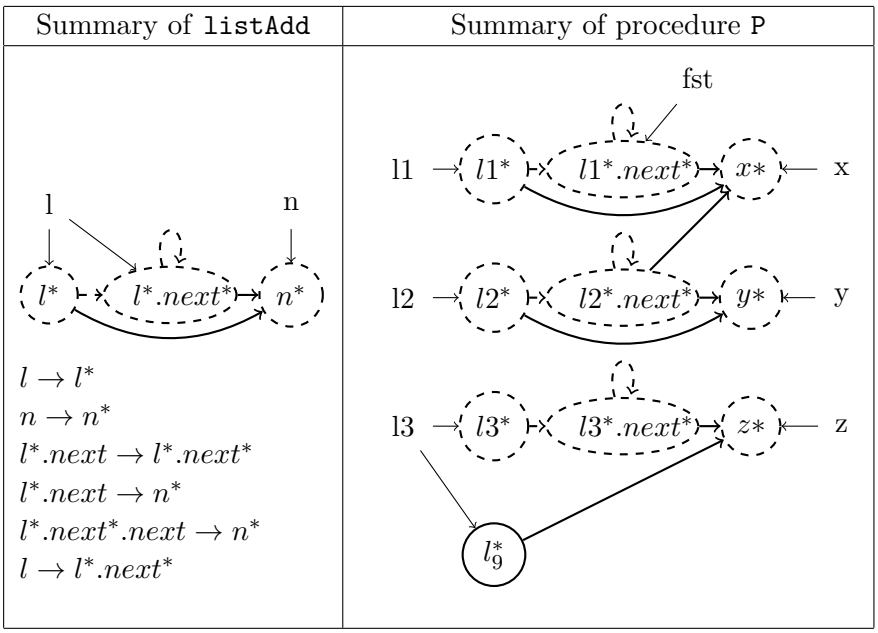


Figure 7.4: Summaries computed by the access-path based analysis proposed in [Cheng and Hwu, 2000] for the procedures `listAdd` and `P` shown in Fig. 7.1.

WSR Analysis Fig. 7.2 shows the summaries computed by WSR analysis for the procedures `listAdd` and `P`. The abstract domain and semantics of the analysis are very similar to those of our framework except that it does not track the happens before relation (i.e, it applies the *Omitting-happen-before* abstraction to the framework). The definitions used by WSR analysis for the parameters of the framework will be clarified in the following discussion.

Consider the summary of the procedure `listAdd` shown at the left side of Fig. 7.2. The two external vertices (shown by dashed edges) n_l and n_n represent the targets of the parameters at the entry the procedure. WSR analysis binds every parameter and global variable p to a vertex with name n_p at the start of a procedure.

The external vertex n_2 represents the targets of the reference `l.next` read at line 2 of `listAdd` procedure. The analysis uses the label of a field-read statement to name the external nodes created while processing the statement. (The analysis would also create an external edge for the `next` field read at line 1, which is not shown in the figure). At the end of the while loop, `l` can be pointing to the initial node n_l or to the nodes read at line 2, namely n_2. The internal edges shown in the figure correspond to the field-write performed at line 2.

Consider the procedure `P`. At all points where `P` invokes `listAdd` i.e, at lines 6, 7, 8 and 10, the summary of `listAdd` will be composed with the transformer graph before the corresponding invocation of `listAdd`. For example, consider the first call to `listAdd` at line 6. The parameter node n_l of the summary of `listAdd` will be mapped to n_{l1}. The external vertex n_2 will be mapped to itself, and the parameter node n_n will be mapped to n_x. The summary of `listAdd` will be translated by applying this mapping and will be composed with the transformer graph before the call.

In the last call to `listAdd` at line 10, while composing the summary of `listAdd` with the transformer graph before the call, the analysis will not add an external edge from n_9 to n_2 because n_9 is a non-escaping node (i.e, a node not reachable from the *prestate*). The transformer graph computed at the end of `P` is shown at the right side of Fig. 7.2. Notice that in the summary of `P`, the `next` fields of the lists `l1` and `l2`

have the same target n_2. This is sound but imprecise as the summary implies that the procedure `P` adds the list nodes $\{x, y, z\}$ to the contents of the lists `l1` and `l2`.

This imprecision is the consequence of not cloning the external vertices while composing the transformer graphs.

Data Structure Analysis Fig. 7.3 shows the summaries computed by the *data structure analysis* (DSA) [Lattner et al., 2007] for the procedures `listAdd` and `P`. DSA does not distinguish between internal and external vertices or edges. Hence, there is only one kind of vertices and edges, which has to be treated as both internal and external. The analysis is also unification based.

The summary for `listAdd` is shown at the left side of Fig. 7.3. To clarify how the analysis came up with this summary we present a step by step walk-through on the procedure `listAdd`. Initially, the analysis will bind `l` and `n` to two different abstract vertices say n_l and n_n. At line 2, on seeing the field-read `l.next`, the analysis will create a new abstract node, say n_2, to represent the target of `l.next`. However, since `l.next` is assigned to `l` in the same statement, the targets of `l` and `l.next` (namely n_l and n_2) will be merged into a single node.

Finally, the field-write at line 3 will result in the merging of the target of `l.next` (which is also now the target of `l`) with `n`. Hence, the summary will have only one abstract node eq_1 that represents the targets of `l`, the objects transitively reachable through the `next` fields of `l`, and the parameter `n`.

The summary of the procedure `P` is shown at the right side of Fig. 7.3. It is obtained by composing the summaries of `listAdd` at each call statement. Consider the first call to `listAdd` at line 6. Unlike WSR analysis, DSA clones the entire summary of the callee by renaming every vertex to a fresh name during summary compositions. The summary of `listAdd` is first cloned by renaming eq_1 to eq_2 and then is composed with the transformer graph before the call. The resulting graph will have a single node eq_2 to which both `l1` and `x` will point to.

A similar operation will be performed during the second call to `listAdd` at line 7. However this time, cloning will not result in ad-

ditional vertices as explained in the following. Say eq_1 is renamed to eq_4 in the cloned summary of `listAdd`. The vertex eq_4 represents the target of `x` (in addition to representing other objects such as the targets of `l2`, `l2.next` etc.). But, the transformer graph before the call will already have a node eq_2 representing the target of `x`. Therefore, eq_4 and eq_2 will be merged in the composed transformer graph as they both represent the target of `x`. After this statement, eq_2 represents the targets of `l1`, `x`, objects reachable via their next fields, as well as the target of `l2`.

The third call to `listAdd(l2, y)` at line 8 will result in the merging of eq_2 and the target of `y`. This time also the vertex cloned from the callee summary (a copy of eq_1) would be merged with eq_2 because they represent the targets of `l2`. The final call to `listAdd(l3,z)` will add a new vertex eq_3 to the composed graph as a result of cloning and composing the summary of `listAdd`. The node eq_3 represents the object newly allocated by the procedure as well as the initial targets of `z` and `l3` at the start of the procedure.

Observe that in spite of using less precise transformer graphs compared to WSR analysis, DSA identifies that the procedure `P` adds only `x` and `y` to the contents of `l1` and `l2`. This is better than the results of WSR analysis but is still imprecise. However, flow-insensitivity and the lack of distinction between internal and external vertices prevents the analysis from determining that `l3` is a new singleton list containing only `z`.

Access-path-based Analysis Fig. 7.4 shows the summaries computed by the analysis proposed in [Cheng and Hwu, 2000] for the procedures `listAdd` and `P`. We will refer to this analysis as *access-path-based analysis* (APA) as it summarizes procedures using a set of points-to relations between access-paths.

The access-paths used by the analysis are a sequence of field dereferences starting from a parameter or a global variable or a statement label (that corresponds to an object allocation site). Access-paths that end with a star ("*") denote objects and those without a suffix "*" denote pointers (or references). In an access-path of the form $\delta.f$, δ

always ends with a star i.e, it represents an object. In the analysis, if `l` is a parameter of a procedure, l^* is used to represent the initial target of the parameter `l`, $l^*.next^*$ is used to represent the initial target of `l.next` and so on. The objects that are newly allocated are denoted using their statement labels. [1]

The analysis also bounds the lengths of the access-paths to keep the abstract domain finite. A bounded access-path δ represent a (possibly infinite) set of access-paths whose prefix is δ. In Fig. 7.4, we assume that the length of the access-paths are bounded to 2.

The summaries computed by the analysis can be represented as transformer graphs. The vertices in the transformer graphs are named using access-paths that denote objects. An access-path $l^*.next^*$, in reality, denotes an external edge from the vertex l^*, which represents the initial target of `l`, to $l^*.next^*$, which represents the initial target of `l.next`. A points-to relation from $l^*.next$ to n^* corresponds to an internal edge from l^* to n^* labelled by the field `next`. (See the *Accesspath-based-naming* strategy of section 6 for a formal description.)

The transformer graph representation of the summaries of APA are shown in Fig. 7.4. We also show the actual points-to relations computed by the analysis for the procedure `listAdd`. Readers might find it easier to compare the transformer graph representations of the summaries with the summaries computed by the other instances.

Consider the summary computed by APA for the procedure `listAdd` (left side of Fig. 7.4). Its transformer graph representation clarifies that the summary for the procedure is semantically equivalent to WSR summary shown at the left side of Fig. 7.2. They differ only in the names of the vertices. Consider the summary for the procedure `P` shown at the right side of Fig. 7.4. The summary precisely identifies the contents of the lists `l1` and `l2`, which is in contrast with the results of WSR analysis. This increased precision is the consequence of using a naming strategy for nodes that makes it necessary to clone the external vertices during summary composition. The analysis maintains

[1] [Cheng and Hwu, 2000] denote the field references in access-paths using field offsets instead of field names, since the analysis targets C programs where distinct field names can denote the same offset within a record.

the invariant that the targets read from a vertex l^* on a field f are always named $l^*.f^*$. In other words, an external edge from l^* on a field f always ends at $l^*.f^*$. To maintain this invariant, during summary composition the targets of external edges are renamed so that they are consistent with the names of their source vertices. However, being flow-insensitive the analysis does not identify the contents of the list `l3` precisely.

It turns out that when this analysis is formulated as an instance of the framework it closely resembles WSR analysis in most aspects except for the naming of nodes (see section 7.2.4). This is an interesting result considering that the original formulation of this analysis as proposed in [Cheng and Hwu, 2000] bears little similarity to WSR analysis. Moreover, we were able to identify some bugs in the analysis proposed in [Cheng and Hwu, 2000] by comparing their semantics with the transfer functions of the instantiation of our framework that uses the same naming strategy.

Flow-Aware Analysis We now discuss the summaries computed by the analysis proposed in [Buss et al., 2008], referred to as *flow-aware analysis* (FAA), for the procedure `listAdd` and `P`. The summaries computed by FAA are not depicted pictorially since they are very similar to the summaries computed by WSR analysis. The critical difference between FAA and the other analyses discussed earlier manifest when there is aliasing in the context in which the summaries are used.

For example, consider the procedure `Q` shown in Fig. 7.1. The procedure first adds a list node `z` to the parameter list `l` and then invokes the procedure `P` passing the list `l` to `l1` and `l2` parameters. In this context, `l1` and `l2` alias. The procedure `P` invokes the procedure `first`, which reads the targets of the `next` field of the list `l` via the reference `l1`. The procedure `P` subsequently adds the nodes `x` and `y` to the list `l` via the reference `l2`. In this case, the summaries obtained for `Q` by every other instance of the framework discussed earlier will conservatively infer that `fst` could be pointing to `x`, `y` or `z`, though clearly, `x` and `y` are added to the list `l` only after the call to `first`.

This is because every analysis discussed earlier do not track the relative ordering between internal and external edges in their summaries. Therefore, they cannot identify that the external edge representing the read of the `next` field of the list `l1` by the `first` procedure precedes the writes performed on the `next` field of the list `l2` by the `listAdd` procedure. The flow-aware analysis stores an approximation of the happens before ($\rightsquigarrow$) relation and hence can decipher that `fst` can only be pointing to `z`.

The preceding discussion highlights that the four analyses actually have varying levels of precision and scalability. Moreover, some of them use apparently very different representations for summaries. In the sequel, we present a succinct formalization of the analyses as instances of the framework.

7.1.1 Language Specific Extensions of the Analyses

Each of the analyses that we formalize have extensions specific to the languages that they analyse. The WSR analysis was designed to analyse Java programs and the rest of the analyses were designed to analyse C-like programs. To analyse C programs, it is necessary to handle C language specific features like address-of operators, pointer arithmetic, type casts, unions and function pointers. For analysing Java programs, we need extensions for handling virtual method calls, static fields and so on.

Formalizing the semantics of the analyses for the language-specific features requires non-trivial extensions to the abstract domain and transfer functions of our framework. But, in most cases, they are conceptually straight-forward. For example, the framework can be extended to handle non-record type, pointer-valued memory locations by allowing unlabelled edges in the transformer graph e.g, like $\langle u, v\rangle$ to capture that the memory location u may-point-to v.

For a more non-trivial example consider the address-of (&) operator. To support this operation, we need to distinguish between *lvalues* (i.e, addresses) and *rvalues* (i.e, memory locations) of variables and fields of objects. The mapping σ_{in} and σ should be considered as mappings from lvalues of variables to their rvalues. Every edge $\langle u, f, v\rangle$ that

indicates that u points-to v on field f should be split into two edges $\langle u, f, r_f \rangle$, $\langle r_f, v \rangle$, where r_f denotes the rvalue of the field f. The transfer functions can be easily adapted to this extended abstract domain. The only significant change required is that reading a variable or an object's field should be considered as reading the targets of its rvalue. Hence, it should result in the creation of external edges. Similarly, writing a variable or an object's field should be considered as writing to its rvalue. Hence, it should result in the creation of internal edges. The semantics of a statement of the form $v_1 = \&v_2$ could then be modelled as adding an internal edge from the rvalue of v_1 (given by $\sigma(v_1)$) to the rvalue of v_2 (given by $\sigma(v_2)$), which indicates that v_1 points-to v_2.

All the analyses except DSA assume that the targets of function pointer calls and virtual method calls are known during the bottom-up summary computation phase. They either employ a separate conservative call-graph analysis to estimate the call-graph, or invoke a top-down phase that propagates the points-to information from callers to callees, and the bottom-up summary computation phase iteratively, until the summaries converge to a fix-point. Hence, they do not require any extensions to the transfer functions for handling dynamic dispatch. On the other hand, DSA has a custom mechanism for handling function pointer calls which is discussed while formalizing the analysis in section 7.2.3.

We do not formalize the language specific extensions of the analyses in this article for the following two reasons. Firstly, we believe that formalizing these features is a distraction from the focus of this article which is to understand the analyses at a conceptual level and generalize them. The additional formalism that would be required for the extensions may make it difficult to understand the analyses, and to appreciate their similarities and differences. Secondly, most of the language specific extensions are rather straight-forward to devise once the core abstractions of the analyses are well understood.

7.2 Formal Definitions of the Instances

7.2.1 WSR Analysis

WSR analysis can be considered as an instance of our framework that applies the following specializations to the base abstract semantics $(\mathcal{A}_{\mathcal{G}}, \mathcal{F}_{\mathcal{G}})$.

$$(\mathcal{A}_{WSR}, \mathcal{F}_{WSR}) = \begin{cases} (\mathcal{A}_{\mathcal{G}}, \mathcal{F}_{\mathcal{G}}) & \\ \left.\begin{array}{l} +\textit{Creation-site-naming} \\ +\textit{Lazy-garbage-collection} \end{array}\right] & (instantiations) \\ \left.\begin{array}{l} +\textit{Partial-eager-simplification} \\ +\textit{Omitting-}\sigma_{in} \end{array}\right] & (restrictions) \\ +\textit{Omitting-happen-before} & (abstractions) \end{cases}$$

7.2.2 Flow-Aware Analysis

Buss *et al.* present a modular pointer analysis called as flow-aware analysis in [Buss et al., 2008] and [Buss et al., 2010]. Their abstract representation of the state transformers, referred to as *assign fetch graphs*, is very similar to the transformer graphs discussed in this article. They refer to internal edges as *assign edges* and external edges as *fetch edges*. We express the flow-aware analysis as an instance of our framework that performs the following specializations.

$$(\mathcal{A}_{FA}, \mathcal{F}_{FA}) = \begin{cases} (\mathcal{A}_{\mathcal{G}}, \mathcal{F}_{\mathcal{G}}) & \\ \left.\begin{array}{l} +\textit{Creation-site-naming} \\ +\textit{Lazy-garbage-collection} \\ +\textit{Lazy-simplification} \end{array}\right] & (instantiations) \\ +\textit{Omitting-}\sigma_{in} & (restrictions) \\ +\textit{approximate-happens-before} & (abstractions) \end{cases}$$

The analysis tracks an approximation of the $\rightsquigarrow$ relation using a total order (which is described in sections 6.1 and 8.4 of [Buss et al., 2010]). Being aware of the dependence (or flow) between the external and internal edges, the analysis can be more precise in some scenarios

especially when there is aliasing in the calling context (as illustrated in section 7.1).

The total order is constructed by numbering the external and internal edges in the order in which they are created (or re-encountered) during the analysis. The edges with smaller numbers are assumed to happen before those with higher numbers. The disadvantage of this strategy is that it introduces spurious orderings between the reads and writes that happen along the two branches of an if-then-else statement.

Caveats [Buss et al., 2010] informally discusses the *Simplify* operation in section 8.1 "computing summaries". A mapping akin to *Incl* (used by the *Simplify* operation) is discussed in detail in section 4.1 "Pointer Alias Analysis". Unfortunately, since section 8.1 in [Buss et al., 2010] is not sufficiently detailed, we are unable to decipher the precise definition of *Simplify*. The description indicates that their analysis translates the internal edges on a vertex w to those in $Incl(w)$. However, it is unclear if the external edges are translated as well, which is necessary for correctness. Another ambiguity is that in [Buss et al., 2010] it is stated that the analysis "deletes everything from a summary of a procedure that the callers could not see". Intuitively, this seems to correspond to the *removeNonEscaping* and $GC_{\mathcal{G}}$ operations that delete non-escaping vertices and associated edges from the summary. Due to these ambiguities the above instantiation may not precisely capture all the aspects of the analysis presented in [Buss et al., 2010].

7.2.3 Data Structure Analysis

[Lattner et al., 2007], [Lattner and Adve, 2005a] present a modular, unification-based pointer analysis called as data structure analysis (DSA). The analysis is also capable of summarizing procedures in presence of function pointer calls. Indirect calls, such as function pointer calls, virtual method calls and higher-order function calls, pose a major challenge for bottom-up analyses. The targets of indirect calls encountered during an analysis of a procedure may depend on the context in which the procedure is invoked.

DSA stores the indirect calls encountered during the analysis of a procedure in the summaries computed for the procedure. These summaries are propagated to the callers of the procedure as in a typical bottom-up analysis. Eventually, sufficient context information becomes available for resolving some indirect calls to their targets. At this point, the analysis instantiates the summaries of the targets of the indirect calls (that are resolvable) and drops the indirect calls from the summaries.

This feature of DSA is non-trivial to understand and formalize, but is orthogonal to the focus of this article. (For interested readers, [Madhavan et al., 2012] presents a generic mechanism for lifting modular heap analysis approaches for first-order programs to higher-order programs which is inspired by the approach of DSA. The abstract semantics formalized in this section when used in conjunction with the approach discussed in [Madhavan et al., 2012] may serve as a formalization of this feature.) Below we focus on the abstract transfer functions of DSA modulo the handling of indirect calls.

DSA as an instance of the framework

$$
\begin{aligned}
&(\mathcal{A}_{DSA}, \mathcal{F}_{DSA}) = \\
&\quad \left\{ \begin{array}{ll}
(\mathcal{A}_{\mathcal{G}}, \mathcal{F}_{\mathcal{G}}) & \\
\quad + \textit{Creation-site-naming-cloning} \\
\quad + \textit{Lazy-simplification} & (\textit{instantiations}) \\
\quad + \textit{Lazy-garbage-collection} \\
\quad + \textit{No-internal-external-distinction} \\
\quad + \textit{Flow-insensitivity} & (\textit{abstractions}) \\
\quad + \textit{Unification}
\end{array} \right.
\end{aligned}
$$

The analysis performs several abstractions to the base abstract semantics $\mathcal{F}_{\mathcal{G}}$. It turns out that in the presence of these abstractions the transformer graphs can be reduced to triples of the form $(\mathsf{V}, \mathsf{E}, \sigma)$, which is deceptively similar to an abstract shape graph though it represents an abstract state transformer. Due to these abstractions, the transfer functions can be simplified. In particular, the *Simplify* operation does

not add any additional edges but only removes edges and vertices when applied in conjunction with GC_c. The analysis makes up for some of the losses in precision by using a precise naming strategy that clones the abstract objects along all acyclic call-paths.

The analysis does not distinguish between external and internal edges and vertices. Therefore, there is no distinction between edges created due to field-reads versus those created due to field-writes. Nevertheless, the analysis tracks the vertices that are *escaping* by associating a boolean flag (called *complete* flag) with the vertices. Vertices that have the complete flag set to true are *non-escaping* vertices. Though [Lattner et al., 2007] informally states that the vertices marked as complete are analogous to the internal vertices of WSR analysis, in reality they correspond to *non-escaping* vertices.

[Lattner et al., 2007] defines the abstract semantics of a (direct) call statement S as a function *resolveCallee* that invokes two functions *cloneGraphInto* and *resolveArguments* (Figure 4 of [Lattner et al., 2007]). The function *cloneGraphInto* clones the vertices in the callee transformer graph, and inlines the cloned transformer graph into the caller transformer graph. This corresponds to the transfer function of a call statement described in Chapter 5.

The function *resolveArguments* merges the targets of every argument in the caller transformer graph with the corresponding parameter nodes of the callee transformer graph, and also transitively merges the targets of their out-edges. This corresponds to the definition of the parameter *NodesToMerge* for a call statement presented in Fig. 6.1 that results in a unification-based analysis.

7.2.4 Access-path-based Modular Analysis

[Cheng and Hwu, 2000] proposes a modular pointer analysis technique that uses access-paths to name abstract objects. The analysis summarizes procedures using a set of points-to relations between access paths. We will refer to the analysis as access-path-based analysis (APA). Below we present a specialization of the framework that is very similar to APA.

$$(\mathcal{A}_{AP}, \mathcal{F}_{AP}) = \begin{cases} (\mathcal{A}_{\mathcal{G}}, \mathcal{F}_{\mathcal{G}}) & \\ \left.\begin{array}{l} +\textit{Accesspath-based-naming} \\ +\textit{Lazy-garbage-collection} \end{array}\right] & (\textit{instantiations}) \\ \left.\begin{array}{l} +\textit{Partial-eager-simplification} \\ +\textit{Omitting-}\sigma_{in} \end{array}\right] & (\textit{restrictions}) \\ +\textit{Flow-insensitivity} & (\textit{abstractions}) \end{cases}$$

As discussed in section 6, when the *Accesspath-based-naming* strategy is used, the nodes names can be used to determine if a vertex is internal or external, and also to reconstruct the set of external edges. Hence, the internal and external vertex components, and the external edge component can be omitted from the transformer graphs. The specializations used by APA (presented above) results in the omission of the happens before and σ_{in} components from the transformer graphs. (Recall that the flow-insensitivity abstraction omits the happens before component as well.) In essence, the transformer graphs used by APA can be reduced to pairs of the form (IE, σ). A pair (IE, σ) corresponds to a points-to relation between LAP_{r+1} and RAP_r, where r is the bound on the length of the access-paths . This is precisely the representation used by [Cheng and Hwu, 2000] for transformer graphs. They define the abstract domain as $2^{LAP_{r+1} \times RAP_r}$.

The transfer functions given by $\mathcal{F}_{AP}$ can also be simplified using the condensed (paired) representation of the transformer graphs. (We elide formal details for brevity). The abstract semantics presented in [Cheng and Hwu, 2000] is similar to the simplified form of the transfer functions.

Caveats The above instantiation has two differences compared to the formulation presented in [Cheng and Hwu, 2000]. Firstly, in the analysis described in [Cheng and Hwu, 2000], the objects newly created by a procedure are initially named using their allocation site label (like in the *Accesspath-based-naming* strategy). Later, in the summary computed for the procedure every label (which denotes newly allocated objects)

is renamed to an access-path called *extended-access-path*, which is obtained from a path starting at a parameter or global variable and ending at the label of interest. This is an odd feature of the analysis. This is sound, but may result in significant loss of precision as an internal vertex (denoting objects allocated inside a procedure) is converted to an external vertex (that possibly denotes objects read from the prestate). [Cheng and Hwu, 2000] states that the reason for performing this is to ensure that the vertices corresponding to newly allocated objects are given different names in different calling contexts. A more precise way of achieving this would be to clone such vertices in all (or selected) calling contexts similar to DSA.

Secondly, we find that the semantics of [Cheng and Hwu, 2000] has some differences (which are possible bugs) compared to $\mathcal{F}_{AP}$. The transfer function $\mathcal{F}_{AP}$ presented above performs partial simplification of transformer graphs which is presented in Appendix A. With this specialization, the transfer functions of field-read and procedure call statements create external edges that start only from escaping vertices, which is an optimization compared to the generic abstract semantics of the framework.

However, the semantics described in [Cheng and Hwu, 2000] pushes this further and does not create an external edge from a vertex when there exists an internal edge starting at the vertex. The formalism presented in [Cheng and Hwu, 2000] creates a new access-path $\delta.f^*$, that corresponds to an external edge starting from δ, iff δ does not point to any node on field f (see definition 2.3 in section 2.1 of [Cheng and Hwu, 2000]). This means that their semantics will not create an external edge on an escaping node δ if there exists an internal edge starting from the node. This is unsound. This we believe is a bug in the algorithm of [Cheng and Hwu, 2000].

The formalization presented above is provably correct, addresses these problems and yet remains close to the original formulation of the analysis.

8

Experimental Results

8.1 Implementation, Benchmarks and Metrics

In this chapter, we present an experimental evaluation of the framework using our open source, modular heap analysis tool *Seal* ([Madhavan et al., 2011], [Madhavan et al., 2012]), which is available at `seal.codeplex.com`. *Seal* uses *Microsoft Phoenix Compiler Framework* and can analyze real-world *Microsoft .NET* applications implemented in the $C^\sharp$ programming language.

Seal is an implementation of the framework discussed in this article. In addition, it also has several extensions for handling advanced features of the $C^\sharp$ language. Most importantly, *Seal* implements the approach described in [Madhavan et al., 2012] for handling higher-order features of $C^\sharp$ such as virtual method calls and lambda expressions.

A few client analyses, such as Purity and Side-effects Analysis, Information Flow Analysis, Escape Analysis, Call-graph Analysis, are also integrated into *Seal*. The client analyses are implemented as post-processing phases that analyze the summaries of the procedures computed by the bottom-up summarization phase. Each client analysis requires tracking some additional information in the summaries. For instance, the Side-effects Analysis requires tracking the vertices and

Benchmark (Abbv.)	**Loc**	**Description**
DocX (*doc*)	10K	Library for manipulating Word files
AvalonDock (*ad*)	18K	Library for docking windows
Facebook API (*fb*)	21K	Library for integrating with Facebook
Dynamic data display (*ddd*)	25K	Real-time data visualization tool
TestApi (*test*)	25K	Library for testing tool development
SharpMap (*sm*)	26K	Geospatial application framework
Quickgraph (*qg*)	34K	Graph data structures and algorithms
SharpDevelop - NRefactor (*ref*)	43K	Code refactoring engine
Craig's utility (*cul*)	56K	Collection of .NET utility methods
PDFsharp (*pdf*)	96K	Library for processing PDFs

Figure 8.1: Benchmarks used in the experimental evaluation. All benchmarks are popular, open source $C^\sharp$ projects hosted at `www.codeplex.com`.

their fields that are modified by a procedure in the summary of the procedure. *Seal* analyzes not only procedures but also libraries (DLLs) modularly. Given a DLL, it summarizes the procedures in a DLL using transformer graphs, stores the summaries in a database, and uses the summaries while analyzing the clients of the library. In our experiments, the DLLs implementing the core $C^\sharp$ library, namely, *mscorlib.dll*, *system.dll* and *system.core.dll* were analyzed once and for all, and the computed summaries were used during the analysis of each benchmark.

Seal can be unsound in some cases. It does not soundly handle calls to native methods (which are methods implemented in C or other unmanaged languages), graphical user interface (GUI) methods, reflection, and concurrency. We minimize the unsoundness by using stubs for commonly used library methods that are not analyzable by *Seal*.

To evaluate the practical usefulness of our framework, we analyze real-world $C^\sharp$ libraries shown in Fig. 8.1, using different configurations for the parameters of the framework. We measure the impact of each configuration on the summaries computed for the procedures, and also

on the results of three client analyses: *Purity and Side-effects Analysis*, *Escape Analysis* and *Call-graph Analysis*. In the rest of the chapter, we explain the various configurations used in our experiments, the metrics used to compare the configurations, and the results for each configuration.

Metrics used in the Experimental Evaluation The following are the metrics we use to compare the precision and scalability of different configurations of the framework. The total time taken by the bottom-up summary computation phase. We denote this as *Time* in the experimental results presented shortly.

The number of vertices and edges in the summary transformer graphs of procedures. This is a measure of scalability of the analysis. Analyses that compute smaller summaries are more scalable. In the experimental results presented shortly we denote this metric as *Size*. The results show, for each benchmark, the average and the maximum size of the summaries computed by the analysis.

The average out-degree of the vertices in the summary transformer graphs, which is the ratio of edges to the vertices in the transformer graphs. This is a measure of sparsity of the summaries. Summaries that are sparse are generally more precise as they identify more references as non-aliasing. We denote this metric as *Deg.* in the results presented later. The results show, for each benchmark, the average sparsity of the summaries computed by the analysis and their *mean absolute deviation*.

We use the following metrics to measure the precision of the Purity and Side-effects Analysis.

The number of procedures that are *pure*. A procedure is pure if it does not write to pre-state memory locations (for more details see [Salcianu, 2001]). In the presence of indirect calls, a procedure by itself may not write to prestate locations but the indirect calls invoked by the procedure may modify the prestate locations. Since the targets of the indirect calls may depend on the calling context, the purity of the procedure is also dependent on the calling context. We call such procedures as *conditionally pure*. We include such procedures also in this metric.

The number of *side-effects* of a procedure, which are the prestate memory locations modified by the procedure. It is possible for procedures to have unbounded number of side-effects. For example, a procedure that appends an element to a linked list `lst` passed as a parameter may modify the locations: `lst.next`, `lst.next.next` and so on. One way to represent side-effects of a procedure is using regular expressions over access-paths. For example, the side-effects of the list append procedure could be represented as $\texttt{lst.(next)}^+$. But, with this representation comparing side-effects across different configurations requires checking inclusion and equivalence of regular expressions, which is quite expensive. Therefore, in our experiments, we use an under-approximation of the side-effects of a procedure described below.

For every vertex that is modified in the transformer graph, we choose one access-path starting from each parameter that the modified vertex corresponds to. In the above example, if there are there are two modified nodes in the transformer graph one corresponding to `lst.next` and other corresponding to $\texttt{lst.next.(next)}^+$. Then we display the two access-paths: `lst.next` and `lst.next.next` as side-effects of the procedure.

In the results presented shortly, we show the average number of (under-approximate) side-effects per impure procedure and also their *mean absolute deviation*. We refer to this metric as *SE*.

We measure the precision of the Escape Analysis using the following metric. The total number of non-escaping (or local) allocation sites. An allocation site is local only if during every invocation of the procedure containing the allocation site, no object created by the allocation site has an handle that is live after the execution of the procedure. We refer to this metric as *LA* in the results presented in the next section.

Like many top-down heap analyses, *Seal* constructs a call-graph on-the-fly during the bottom-up summary computation phase (see [Madhavan et al., 2012]). We evaluate the precision of the call-graph computed by *Seal* for a given DLL using the following metric abbreviated as *CGE*. The number of edges in the call-graph computed for the given DLL. A call-graph is more precise if it has fewer edges.

8.2 Evaluation of the Configurations of the Framework

We now describe the various configurations used in the experimental evaluation and discuss the results they produce. Since the parameters of the framework can be defined arbitrarily, exhaustively evaluating all possible configurations of the framework is not practically feasible. Therefore, we adopt the following approach in this evaluation. We first present the results of one of the most tested and scalable configurations of the tool which we refer to as the *base configuration*. This configuration was also used in the experimental evaluation of [Madhavan et al., 2012]. We then turn on one *specialization* of the framework and evaluate the change in the results relative to the base configuration. This would provide an idea of the importance of the specialization and the criticality of the parameter that is affected by the specialization.

All evaluations presented in this section were carried out on a system with 2 core, 2.6 GHz Intel Core i5 processor, with 8 GB RAM, running Windows 8 operating system.

Base Configuration In this configuration, the abstract vertices are named using their creation site labels (*Creation-site-naming*). The configuration uses flow-insensitive analysis and incorporates partial eager simplification of transformer graphs (*Partial-eager-simplification*) in the transfer functions of the statements. Moreover, a node merging strategy is applied to the transformer graphs computed at the exit point of a procedure. The node merging strategy used ensures that every out-going external edge from a vertex is labelled by a unique field, and similarly, every out-going internal edge is also labelled by a unique field. This is one of the most scalable configurations.

Fig. 8.2 shows the results of the analyzing the benchmarks in the base configuration. The figure shows all the metrics discussed earlier for each benchmark.

One Level Cloning This configuration employs a context-sensitive naming strategy. Every node has a label, which is the label of the creation site, and a context. The context of a node at a program point is the label of the first call statement in the call-path that leads to the

	Time	**Size** avg (max)	**Deg.** avg (dev)	**Pure**	**SE** avg (dev)	**LA**	**CGE**
doc	1m55s	84.8 (697)	1.1 (1.1)	392	9.6 (9)	743	4234
ad	18m30s	88 (2059)	1 (1)	704	11.5 (15.4)	1267	7482
fb	2m17s	55.7 (774)	0.8 (0.7)	1978	9.4 (10.9)	976	7028
ddd	5m4s	50.5 (1965)	0.5 (0.3)	1622	3.9 (4.1)	2950	8366
test	7m13s	52.8 (1636)	0.7 (0.5)	709	8.5 (10.2)	1063	6836
sm	1m15s	25.2 (580)	0.5 (0.3)	1052	3.3 (2.8)	747	4588
qg	1m43s	30.8 (959)	0.5 (0.3)	2368	3.1 (3.1)	663	6360
ref	24m31s	76.7 (5274)	0.8 (0.6)	2037	8.8 (9.7)	1115	43160
cul	5m20s	30.6 (1654)	0.5 (0.3)	2774	3.7 (4.2)	2183	12981
pdf	52m39s	89.9 (2659)	0.9 (0.7)	1751	13.9 (17.2)	3482	12721

Figure 8.2: Results of running *Seal* on the benchmarks shown in Fig. 8.1 in the base configuration.

	%Δ **Time**	%Δ **Size** avg (max)	%Δ **Deg.** avg (dev)	%Δ **Pure**	%Δ **SE** avg (dev)	%Δ **LA**	%Δ **CGE**
doc	-3.4	-10 (-4.7)	-23.4 (-33.8)	5.6	-49 (-51.5)	2.4	-10.5
ad	-64.4	-55.5 (-75.1)	-36.5 (-53.7)	0.7	-72.3 (-85.3)	3.4	-29.9
fb	-13.5	-23.3 (18.3)	-29.2 (-46.6)	0.4	-30.1 (-34.1)	9.9	-15.6
ddd	-36.3	-24 (-6.9)	-8.4 (-17.6)	0.1	-18.5 (-28.3)	0.4	-27.2
test	625.9	-6.2 (-11.2)	-13.6 (-26.1)	1.3	-40.9 (-51.3)	1.9	-13.8
sm	-16.5	14.8 (8.3)	-3.6 (-7.9)	0.9	3.2 (7.2)	0.8	-16.5
qg	-11.8	-2.1 (5.3)	-6.7 (-15)	0.3	-12.5 (-18.8)	2.4	-17.7
cul	38.3	-0.7 (17.2)	-7 (-17.1)	0.7	-8.7 (-10.8)	3.9	-20
pdf	-17.7	-8 (-11.1)	-22.9 (-37.7)	0.2	-8.8 (-10.2)	0.2	-7.5

Figure 8.3: Results obtained with one level cloning of abstract nodes. The values shown in the figure are relative to the values of the base configuration shown in Fig. 8.2.

creation site of the node from that program point. Note that unlike the *Creation-site-naming-cloning* strategy discussed earlier, this strategy does not use the entire call-path as the context but only the label of the first call-site in the call-path.

For example, consider a procedure `A` that calls procedure `B` which in turn calls procedure `C` twice. The abstract nodes created by the two calls to procedure `C` will have distinct names in procedure `B`, but would be indistinguishable in procedure `A` under this naming scheme. We refer to this naming strategy as *one level cloning*. This configuration is similar to the base configuration in every other aspect.

Fig. 8.3 shows the results obtained using this configurations for each benchmark. All numbers in the figure (including those in parentheses) show the percentages by which the values of the metrics differ compared to the base configuration. Percentages that are negative indicate a decrease in the value of the corresponding metric relative to the base configuration. For example, in this configuration, the analysis of the benchmark *doc* took 3.4% lesser time compared to the base configuration, whereas the analysis of benchmark *test* took almost 625% more time compared to the base configuration. The analysis did not scale to the benchmark *ref* under this configuration within a fixed time limit of 90 minutes and hence is omitted from the results. Note that though this configuration uses a context-sensitive naming strategy it sometimes results in significantly faster running times and more compact summaries for benchmarks like *pdf*.

The results for the *sparsity* metric show that this configuration results in sparser summaries compared to the base configuration across all benchmarks. This is expected since cloning increases the context-sensitivity of the analysis.

The increase in the precision of the summaries positively influences the results of the client analyses as shown by the remaining columns in the Fig. 8.3. In particular, more methods are identified as pure and fewer side-effects are inferred for the impure methods. More allocation sites are identified as non-escaping and the call-graph has significantly fewer edges compared to the base configuration. Readers might observe that for the benchmark *sm* the figure indicates an increase in

	%Δ **Time**	%Δ **Size** avg (max)	%Δ **Deg.** avg (dev)	%Δ **Pure**	%Δ **SE** avg (dev)	%Δ **LA**	%Δ **CGE**
doc	-8.2	6.2 (6.9)	-3.9 (-7.2)	0	-0.9 (-1.4)	0	-3.4
ad	-3.1	0.5 (20.1)	-10.3 (-18.9)	0	-3.2 (-5.2)	0	-1
fb	13.2	5 (-11.2)	-13.2 (-27.3)	0	0.1 (0.1)	8.9	-1.4
ddd	71.7	27.2 (127.6)	-2.9 (-6.4)	0.1	-11.9 (-18.7)	0.1	-6.8
sm	157.6	229.4 (2604)	-3 (-6.5)	0.1	6 (10.1)	0	-0.2
qg	8.6	1.6 (15)	-2.3 (-5.3)	0	-4.4 (-7.2)	0	-6.5
cul	-2.1	31.4 (129.1)	-2.4 (-6.8)	0	5.7 (8.6)	0	-3.9

Figure 8.4: Results obtained with full cloning of abstract nodes. The values shown in the figure are relative to the values of the configuration: *One Level Cloning* shown in Fig. 8.3.

the number of side-effects. However, on manual inspection, we found that this was because of the duplicates in the listing of side-effects and do not correspond to an actual increase in the side-effects. The under-approximation heuristic that we use to list side-effects did not identify the duplicates as they were syntactically different.

Full Cloning This configuration performs full cloning of abstract nodes like in the case of DSA, i.e. it uses the *Creation-site-naming-cloning* strategy. It is similar to the base configuration in every other aspect. Fig. 8.4 shows the results obtained with this configuration. All numbers in the figure (including those in parentheses) show the percentages by which the values increase or decrease with respect to the

One Level Cloning configuration. Under this configuration, the analysis timed out on three benchmarks: *test*, *ref* and *pdf*. Hence, they are omitted from the Fig. 8.4.

Notice that for most benchmarks this configuration increases the summary sizes compared to *One Level Cloning*. In the case of the *sm* benchmark the increase in the summary sizes is quite dramatic (almost 26 times). In contrast, the decrease in average degree of vertices (i.e, the increase in sparsity) of the summaries is only marginal.

There is also almost no increase in the number of pure methods. However, there is some reduction in the number of side-effects per procedure in some benchmarks. (The increase in side-effects for a couple of benchmarks were due to the issue of duplicates mentioned while discussing *one level cloning*.) The Escape Analysis doesn't benefit across the board from the increased context-sensitivity. The number of local allocation sites increases significantly for the *fb* benchmark, but remains almost unchanged for the remaining benchmarks. However, the Call-graph analysis does benefit from the increase in context-sensitivity. In particular, the number of call-graph edges decreases across all benchmarks.

On the whole, the results of the *One Level Cloning* and *Full Cloning* configurations indicate the following trends. (a) There is a huge improvement in the precision of the summaries as well as that of the client analyses while switching from a context-insensitive naming strategy to a context-sensitive naming strategy. (b) However, using a very precise context-sensitive naming strategy does not benefit some applications, but instead hampers the scalability of the analysis quite significantly.

The bottom line is that choosing the right level of context-sensitivity in the naming of abstract nodes should be one of the main concerns while designing a modular heap analysis for a particular application.

Flow-sensitive Configuration This configuration performs a flow-sensitive analysis but otherwise uses the same abstractions as the base configuration. Fig. 8.5 shows the results obtained with this configuration. All numbers in the figure (including those in parentheses) show

	%Δ **Time**	%Δ **Size** avg (max)	%Δ **Deg.** avg (dev)	%Δ **Pure**	%Δ **SE** avg (dev)	%Δ **LA**	%Δ **CGE**
doc	49.9	-1 (-12.1)	-0.5 (-0.4)	0	-0.4 (-0.6)	0.7	-2.1
ad	90.2	-1.9 (0)	-4.1 (-4.7)	0.4	-3.5 (-4.4)	0.5	-3.3
fb	21.4	-3.5 (-0.6)	-4.7 (-7.3)	0	-2 (-2.3)	0	-1.8
ddd	-3.3	-1.2 (-1.1)	-1.4 (-2.8)	0.1	-1.2 (-0.3)	0	-0.1
test	110.8	-3.9 (-0.9)	-2.7 (-5)	0	-4.1 (-2.9)	0.2	-1.8
sm	87.3	-7 (-8.6)	-2.2 (-5)	0.3	-8 (-5.1)	0.1	-2
qg	-10.1	-5.1 (-9.2)	-4.6 (-10.6)	0.2	-5.3 (-4.4)	0	-8.5
ref	25.3	-0.6 (-0.5)	-0.7 (-1.1)	0	-0.8 (-0.7)	0.1	-3.4
cul	13.4	-2 (1.4)	-1.9 (-4.6)	0.2	-3.6 (-5.2)	0.5	-1.9
pdf	-17.4	-9.9 (-5.5)	-10.6 (-18.1)	0.2	-1.5 (-1.4)	-0.9	-9.8

Figure 8.5: Results obtained using a flow-sensitive analysis. The values shown in the figure are relative to the values of the base configuration.

the differences (in percentage) compared to the results of the base configuration shown in Fig. 8.2. The data highlights that the flow-sensitive analysis results in a marked increase in the precision when compared to the base configuration, especially in the number of call-graph edges and average side-effects per procedure.

It may be somewhat surprising to the readers that flow-sensitivity does not significantly slow down the analysis. Even though there is a two fold increase in the analysis time for some benchmarks, all of the benchmarks except *pdf* complete well within 30 minutes. The analysis of the *pdf* benchmark actually runs faster. This is mainly because of our efficient implementation of the flow-sensitive analysis. Recall that the transfer functions of the framework perform only weak updates on the abstract vertices. Hence, only variables are strongly updated during a flow-sensitive analysis.

We exploit this property in the implementation. We maintain a single copy of the portion of the transformer graph that is weakly updated, and use separate copies of the variable to abstract node mapping (σ) at each program point. However, this may result in some loss of flow-sensitivity, since the heap-effects along one branch of an if-then-else statement may percolate into the other branch. Nevertheless, it greatly reduces the overhead of copying the transformer graphs at each program point.

Readers might notice that in the *pdf* benchmark the number of local allocation sites (LA) actually decreases. Our investigations reveal that this is due to increased merging of nodes in the summaries of a few procedures compared to the flow-insensitive configuration. Unfortunately, since the benchmark is huge and also because its analysis takes a long time to complete, we were unable to precisely determine the reason for this aberration.

Happens Before Configuration This configuration performs a flow-sensitive analysis and additionally tracks an approximation of the happens before relation in the transformer graphs. The happens before relation is approximated by a total order using the idea presented by *flow-aware analysis* [Buss et al., 2010], which is briefly explained in

	%Δ **Time**	%Δ **Size** avg (max)	%Δ **Deg.** avg (dev)	%Δ **Pure**	%Δ **SE** avg (dev)	%Δ **LA**	%Δ **CGE**
doc	24.1	0	0	0	0	0	0
ad	19	0	0	0	0	0	0
fb	2.7	0 (-0.1)	0	0	0	0	0
ddd	26.3	0	0 (-0.1)	0	0	0	0
test	45.1	-0.1 (0)	-0.1 (-0.1)	0	-0.1 (0)	0	0
sm	-24.5	-0.1 (0)	-0.3 (-0.8)	0	-0.5 (-0.8)	0	0
qg	12.4	-0.1 (0)	-0.1 (-0.3)	0	0	0	-0.1
ref	-21.9	0	0	0	0	0	0
cul	17.6	0	0 (-0.1)	0	0	0	0
pdf	62.6	-0.1 (0)	0 (0.1)	0	-0.4 (-0.5)	0	0

Figure 8.6: Results obtained using a flow-sensitive analysis and with the tracking of the happens before relation. The values shown in the figure are relative to the values of the flow-sensitive configuration.

	%Δ **Time**	%Δ **Size** avg (max)	%Δ **Deg.** avg (dev)	%Δ **Pure**	%Δ **SE** avg (dev)	%Δ **LA**	%Δ **CGE**
doc	83.6	167.5 (719.1)	8.4 (11.1)	5.6	-56.6 (-60.4)	3.1	-10.6
fb	-12.2	79.1 (123.8)	-18 (-29.6)	0.4	-30.2 (-34.1)	2.6	-14.5
sm	24.4	173.5 (720)	30.6 (62.6)	0.8	4.2 (16.9)	0.3	-13

Figure 8.7: Results obtained without node merging. The values shown in the figure are relative to the results of the base configuration.

section 7.2.2. We use this approximation since the overhead of tracking the happens before relation precisely becomes prohibitively expensive as the number of edges in the transformer graphs increase.

Fig. 8.6 shows the results obtained with this configuration relative to the results of the flow-sensitive configuration shown in Fig. 8.5. The results indicate that this configuration results in slower running times in most cases compared to the plain flow-sensitive analysis, but only results in marginal increase in precision. This suggests that omitting the happens before relation from the transformer graphs is perhaps a good trade-off for certain applications. However, note that even this marginal increase in precision may be critical for certain applications like program verification.

Impact of Node Merging To measure the impact of the node merging abstraction, we evaluate the benchmarks under two variations of the base configuration . The first variation, called as *base minus node-merging*, does not employ a node-merging strategy at any point in the analysis. The second configuration, referred to as *quasi-unification*, aggressively merges nodes in the transformer graphs during the analysis of procedures. Note that the base configuration itself applies a lossy node merging strategy at the exit points of the procedures.

The results for the configuration without node merging is shown in Fig. 8.7. Only three benchmarks were analyzable within a time limit of one and a half hours with this configuration. The reason for this is evident from the values for the summary size metric, which highlights that the sizes of the summaries increase by an order of magnitude in the absence of node merging. However, this configuration does improve the precision of the summaries as indicated by the precision of the client analyses.

Fig. 8.8 shows the results of analyzing the benchmarks under the *quasi-unification* configuration. This configuration applies the node merging strategy used by the base configuration not only to the transformer graphs resulting at the exit point of a procedure but also to the transformer graphs that results before and after the call-statements encountered during the analysis of a procedure. The node merging strategy used ensures that every external (or internal) edge starting from a vertex is labeled by a unique field. This is almost like performing a unification based analysis, and hence the name quasi-unification.

The results shown in Fig. 8.8 indicate that the analysis runs slightly faster on most benchmarks under this configuration. There is a marginal increase in the average degree of the summaries implying that the summaries are becoming denser. The loss of precision in Purity and Side-effects Analysis is minimal except for the benchmark *doc*. However, this configuration worsens the results of the Escape Analysis and Call-graph Analysis for most benchmarks.

The results of these two configurations indicate that node merging may result in loss of precision in the client analyses. However, it cannot be completely dispensed with as the analysis without node merging hardly scales to large real-world applications. This leads us to the conclusion that node merging should be carefully fine tuned to suit the precision-scalability needs of an application.

	%Δ **Time**	%Δ **Size** avg (max)	%Δ **Deg.** avg (dev)	%Δ **Pure**	%Δ **SE** avg (dev)	%Δ **LA**	%Δ **CGE**
doc	27	2.8 (0.4)	4 (4.7)	-3.1	13.7 (8.5)	-7.1	0.5
ad	12.1	0	0.3 (0.3)	0	0.1 (0)	-1.1	2.8
fb	4.9	-0.4 (2.3)	1.4 (2)	0	1.2 (1.5)	-0.2	5.3
ddd	-15.9	0	0.1 (0.2)	0	0	0	0.1
test	-27.2	0.7 (-0.7)	1.7 (3.4)	0	0.3 (0.1)	-0.6	0.6
sm	-12.1	-0.4 (0.7)	-0.2 (-0.5)	0	0.2 (0.4)	-0.4	0.1
qg	-32.1	0	0.1 (0)	0	-0.5 (-0.9)	-0.2	0
ref	-31	-0.6 (-0.2)	2 (3.9)	0	2.7 (3.8)	-0.6	2.2
cul	10.7	0.2 (0)	0.5 (0.8)	0	0	-0.1	0.3
pdf	-6.8	0.1 (0.3)	0.4 (0.7)	0	-0.1 (-0.2)	-3.8	0.4

Figure 8.8: Results obtained with quasi-unification configuration. The values shown in the figure are relative to the results of the base configuration.

9

Related Work and Conclusion

Related Work Pointer and heap analysis techniques have been studied extensively in the past few decades. Much of the research on pointer analysis has been directed toward top-down analyses. Some of the recent works include [Smaragdakis et al., 2014], [Marron et al., 2012], [Hardekopf and Lin, 2011], [Smaragdakis et al., 2011], [Lhoták and Chung, 2011], [Liang and Naik, 2011], [De and D'Souza, 2012], [Zhang et al., 2014].

Modular heap analysis techniques have also been a subject of active research, but have attracted lesser attention. Some of the important works include [Chatterjee et al., 1999], [Whaley and Rinard, 1999], [Cheng and Hwu, 2000], [Liang and Harrold, 2001], [Nystrom et al., 2004], [Salcianu and Rinard, 2005], [Lattner et al., 2007], [Buss et al., 2008], [Jeannet et al., 2010], [Dillig et al., 2011], [Madhavan et al., 2011], [Madhavan et al., 2012], [Kneuss et al., 2013], [Mangal et al., 2014].

Parametric frameworks for heap analyses with tunable precision and scalability have also been explored by prior works. However, almost all of these works target top-down heap analyses. A pioneering work in this space is TVLA [Sagiv et al., 1999]. TVLA is a parametric abstract interpretation. It has been used to formalize a number of heap

analyses, and has been applied extensively to verify deep properties of imperative programs. The DOOP heap analysis framework ([Bravenboer and Smaragdakis, 2009], [Smaragdakis and Bravenboer, 2010]) provides a declarative language for expressing points-to analyses. In this framework, a points-to analysis is expressed as a set of *datalog* rules which is then solved using an optimized datalog engine. The burden of proving the analysis correct still rests with the developer of the analysis. However, the framework relieves the developer from having to perform low-level performance optimizations. DOOP has been used to implement a number of context-sensitive points-to analyses.

Other parametric points-to analyses include k-CFA style context- and object-sensitive analyses such as the analysis proposed in [Smaragdakis et al., 2011], [Liang and Naik, 2011], [Zhang et al., 2014], the *Paddle* pointer analysis of the SOOT program analysis framework [Lhoták, 2006], the pointer analysis of the WALA program analysis framework [WALA]. The context-sensitivity of these analyses can be varied by adjusting the parameter k.

To our knowledge there is no prior work on parametric frameworks for modular heap analysis, especially those that are capable of expressing existing modular analyses as instances. One of the main reasons for this is the apparent complexity of these analyses. Below we briefly describe some of the related modular heap analysis approaches that were not described earlier.

[Jeannet et al., 2010] proposed an approach for using the abstract shape graphs of TVLA to represent abstract graph transformers (using a double vocabulary), which is used for modular interprocedural analysis. Their approach can be used to perform bottom-up or top-down interprocedural analysis. However, as duly noted by the authors, when used in a fully bottom-up fashion (as in our framework) the approach may enumerate a large number of input shape configurations.

Rinetzky *et al.* [Rinetzky et al., 2005] present a tabulation-based approach to interprocedural heap analysis of cutpoint-free programs which imposes certain restrictions on aliasing. While our framework computes a procedure summary that can be reused at any callsite, the tabulation approach may analyze a procedure multiple times, but

reuses analysis results at different callsites if the "input heap" is the same. However, there are interesting similarities and connections between the Rinetzky *et al.* approach to merging "graphs" from the callee and the caller and the transformer graph composition of our framework.

Modular approaches have also been explored by separation logic based shape analysis techniques such as [Calcagno et al., 2009], [Gulavani et al., 2009]. They compute Hoare triples, which correspond to conditional summaries: summaries which are valid only in states that satisfy the precondition of the Hoare triple. These summaries typically incorporate significant "non-aliasing" conditions in the precondition.

Conclusion and Future Work In this article, we proposed a framework for modular heap analysis that creates context-independent summaries for procedures but avoids enumerating possible configurations of the input heap. We presented our framework as a parametric abstract interpretation and established the correctness and termination properties. We showed that the framework subsumes at least four existing modular heap analyses.

We presented an experimental evaluation of the framework using our implementation *Seal*. We evaluated the framework by analyzing ten real-world $C^\sharp$ applications with six different configurations for the parameters. We used three client analyses to gauge the precision and scalability of the analysis. The results highlight that the framework can be instantiated to obtain analyses with varying levels of precision and scalability.

Parametric heap analysis tools such as *Seal* are quite useful as they enable fine tuning of heap analyses to suit a particular application. They also provide a platform for developing analyses that can dynamically alter their precision and scalability during an execution, like the analyses proposed in [Smaragdakis et al., 2014] and [Liang and Naik, 2011].

The framework proposed in this article has two limitations: (a) it does not perform strong updates on heap allocated objects, and (b) it does not support path-sensitivity. The program representation of transformer graph presented in this article hints at a potential approach

to address both these challenges. Recall that every internal edge in the transformer graph corresponds to a non-deterministic write which may or may not execute. In order to support strong updates, we need to allow unconditional writes in the programs representing transformer graphs. Hence, the transformer graphs have to be enriched with two types of internal edges: *may* and *must* internal edges. Moreover, the node naming strategy must be enriched so that it distinguishes between *singleton* nodes that represent a single concrete object and *summary* nodes that represent two or more concrete objects, akin to TVLA-style shape analyses. With these extensions it is possible to perform strong updates on singleton nodes.

Path-sensitivity can also be incorporated into the transformer graphs by associating internal and external edges in the transformer graphs with predicates that capture the conditions under which an edge holds. We plan to explore these approaches in the future.

Appendices

A

Simplified Transformer Graphs

This section formalizes the partial simplification restriction informally described in section 6.2.1.

The Domain $\mathcal{A}_{\mathcal{I}}$ Let $\mathcal{A}_{\mathcal{I}}$ denote the set of all transformer graphs $\tau = (\mathsf{EV}, \mathsf{EE}, \sigma_{in}, \mathsf{IV}, \mathsf{IE}, \sigma, \rightsquigarrow)$ that satisfy the following condition:

$$\langle u, f, w \rangle \in \mathsf{EE} \implies u \in \mathit{Escaping}(\tau).$$

The join operator $\sqcup_{co}$ is closed under this domain $\mathcal{A}_{\mathcal{I}}$, i.e. given any $\tau_1, \tau_2 \in \mathcal{A}_{\mathcal{I}}$, $\tau_1 \sqcup_{co} \tau_2 \in \mathcal{A}_{\mathcal{I}}$. Therefore, $(\mathcal{A}_{\mathcal{I}}, \sqsubseteq_{co}, \sqcup_{co})$ forms a sub-lattice of the lattice $(\mathcal{A}_{\mathcal{G}}, \sqsubseteq_{co}, \sqcup_{co})$.

Transformer graphs belonging to $\mathcal{A}_{\mathcal{I}}$ do not contain any external edges with a non-escaping source vertex. Such edges are not essential: e.g., given any transformer $\tau \in \mathcal{A}_{\mathcal{G}}$, the *Simplify* operation presented earlier returns an equivalent transformer that has no such edges. In other words, $\mathit{Simplify}(\tau) \in \mathcal{A}_{\mathcal{I}}$.

The Abstract Semantics $\mathcal{F}_{\mathcal{I}}$

We now present an adaptation of $\mathcal{F}_{\mathcal{G}}$ (the abstract semantic functions) to ensure that they always produce elements of $\mathcal{A}_{\mathcal{I}}$. The abstract se-

$$
\begin{array}{l}
\mathcal{F}_{\mathcal{I}}(v_1 = v_2.f)(\tau) = \\
\quad \text{let } A = \{n \mid \exists n_1 \in \sigma(v_2), \langle n_1, f, n\rangle \in \mathsf{IE}\} \\
\quad \text{let } B = \sigma(v_2) \cap \mathit{Escaping}(\tau) \\
\quad \text{if } (B = \emptyset) \\
\quad \text{then } (\mathsf{EV}, \mathsf{EE}, \sigma_{in}, \mathsf{IV}, \mathsf{IE}, \sigma[v_1 \mapsto A], \rightsquigarrow) \\
\quad \text{else} \\
\quad\quad \text{let } g = \lambda u. \\
\quad\quad\quad (\exists x.\langle u, f, x\rangle \in \mathsf{EE}) \rightarrow \bigcup_{\langle u,f,x\rangle \in \mathsf{EE}} \mathit{Load}_S(u, f, x) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathit{Load}_S(u, f) \\
\quad\quad \text{let } \mathsf{EV}_{new} = \bigcup_{u \in B} g(u) \\
\quad\quad \text{let } \mathsf{EE}_{new} = \bigcup_{u \in B} \{u\} \times f \times g(u) \\
\quad\quad \text{let } \rightsquigarrow_{new} = \{(ie, ee) \mid ie \in \mathsf{IE}, ee \in \mathsf{EE}_{new}\}) \\
\quad\quad (\mathsf{EV} \cup \mathsf{EV}_{new}, \mathsf{EE} \cup \mathsf{EE}_{new}, \sigma_{in}, \\
\quad\quad\quad\quad \mathsf{IV}, \mathsf{IE}, \sigma[v_1 \mapsto A \cup \mathsf{EV}_{new}], \rightsquigarrow \cup \rightsquigarrow_{new})
\end{array}
$$

Figure A.1: The abstract semantics of field-read statement that incorporates the *Simplify* operation partially.

mantic function $\mathcal{F}_{\mathcal{G}}(S)$ shown in Figures 5.1 and Fig. 5.2) is closed with respect to $\mathcal{A}_{\mathcal{I}}$ for most types of statements S. We need to adapt the definition of $\mathcal{F}_{\mathcal{G}}(S)$ only for field-read statements and call statements. One way of doing this would be to apply *Simplify* as the last step (to the transformer graph produced by $\mathcal{F}_{\mathcal{G}}(S)$). Instead, we utilize a *partial simplification* which is more efficient.

Recall that the *Simplify* operation used by $(\mathcal{A}_{\mathcal{G}}, \mathcal{F}_{\mathcal{G}})$ removes unnecessary external vertices (and edges) from a transformer graph but also adds new edges (that were implicitly represented in the original graph). The partial simplification we utilize here removes all external edges from non-escaping nodes, but avoids adding some of the edges

added by *Simplify*. We integrate the partial simplification with the abstract semantic function as it is simpler and more efficient. Furthermore, we exploit the fact that the input transformer graph is already partially-simplified. We denote this variant of the abstract semantics as $\mathcal{F}_{\mathcal{I}}$.

Abstract Semantics of Field-Read Statements

Fig. A.1 formally defines the modified semantics of a field-read statement. Let S denote a field-read statement $v_1 = v_2.f$. The transfer function $\mathcal{F}_{\mathcal{I}}(S)$ is is similar to $\mathcal{F}_{\mathcal{G}}(S)$ (shown in Fig. 5.2) but has two important differences: (a) new external edges are added only to the escaping nodes in $\sigma(v_2)$. If there are no escaping nodes in $\sigma(v_2)$ (i.e, when $B = \emptyset$), no external edges are added. (b) $\sigma(v_1)$ includes the targets of internal edges of $v_2.f$ (the set A). The transfer function $\mathcal{F}_{\mathcal{I}}(S)$ is equivalent to $\mathit{Simplify}_S(\mathcal{F}_{\mathcal{G}}(S)(\tau))$, and hence equivalent to $\mathcal{F}_{\mathcal{G}}(S)(\tau)$. This is formally stated in Lemma A.1. We now explain how the transfer function partially incorporates the simplify operation.

Consider the definition of $\mathcal{F}_{\mathcal{G}}$ for a field-read statement S shown in Fig. 5.2. Applying $\mathit{Simplify}_S$ on $\mathcal{F}_{\mathcal{G}}(S)(\tau)$ will match the newly added external edges EE_{new} to the internal edges starting from vertices in $\sigma(v_2)$ labelled by field f, which is given by the set $\{\langle n_1, f, n\rangle \in \mathsf{IE} \mid n_1 \in \sigma(v_2)\}$. Hence, after simplify, $\sigma(v_1)$ will include the set A. In comparison, $\mathcal{F}_{\mathcal{I}}(S)$ proactively adds the vertices in A to $\sigma(v_1)$. Indeed, *Simplify* may add more vertices to $\sigma(v_1)$ besides the vertices in the set A.

When the domain invariant holds, it can be shown that for the vertices belonging to the set $\sigma(v_2) \setminus B$ which are the non-escaping vertices in $\sigma(v_2)$, all targets on field f are included in the set A (see lemma A.1). Since $\mathcal{F}_{\mathcal{I}}(S)$ has proactively added the vertices in the set A to $\sigma(v_1)$, the external edges on these nodes are redundant and hence are not added by $\mathcal{F}_{\mathcal{I}}(S)$.

In essence, $\mathcal{F}_{\mathcal{I}}(S)$ partially incorporates the *Simplify* operation. However, it does not fully incorporate the *Simplify* operation and so the resulting transformer graph may not be syntactically identical to the result of $\mathit{Simplify}(\mathcal{F}_{\mathcal{G}}(S))$, yet it has the same concrete image.

$$x \in \mathsf{EV}_2 \setminus range(\sigma_{in2}) \Rightarrow x \in \eta_I(x) \tag{A.1}$$

$$x \in \sigma_{in2}(\mathtt{X}) \Rightarrow \sigma_1(\mathtt{X}) \subseteq \eta_I(x) \tag{A.2}$$

$$x \in \mathsf{IV}_2 \Rightarrow Alloc_S(x) \in \eta_I(x) \tag{A.3}$$

$$\langle u, f, x\rangle \in \mathsf{EE}_2, a \in \eta_I(u) \Rightarrow Load_S(a, f, x) \in \eta_I(x) \tag{A.4}$$

$$\langle u, f, v\rangle \in \mathsf{EE}_2, u' \in \eta_I(u), \langle u', f, v'\rangle \in \mathsf{IE}_1 \Rightarrow v' \in \eta_I(v) \tag{A.5}$$

$$\left\{ \begin{array}{c} \langle u, f, v\rangle \in \mathsf{EE}_2, \langle u', f, v'\rangle \in \mathsf{IE}_2, \\ \eta_I(u) \cap \eta_I(u') \neq \emptyset, \\ \langle u, f, v\rangle \rightsquigarrow_2 \langle u', f, v'\rangle \end{array} \right\} \Rightarrow \eta_I(v') \subseteq \eta_I(v) \tag{A.6}$$

Figure A.2: The definition of the function $\eta_I[\![\tau_1, \tau_2]\!]$ used by the composition operation. The transformer graphs $\tau_1 = (\mathsf{EV}_1, \mathsf{EE}_1, \sigma_{in1}, \mathsf{IV}_1, \mathsf{IE}_1, \sigma_1, \rightsquigarrow_1)$ and $\tau_2 = (\mathsf{EV}_2, \mathsf{EE}_2, \sigma_{in2}, \mathsf{IV}_2, \mathsf{IE}_2, \sigma_2, \rightsquigarrow_2)$.

Abstract Semantics of Call Statements

Consider a call statement S. The semantics of the call statement is defined using the composition operation. Here, we redefine the composition operation by partially incorporating the *Simplify* operation as in the case of field-read statement. We define $\tau_2\langle\!\langle\tau_1\rangle\!\rangle_I$ as

$$removeNonEscaping(Append(\tau_1, \tau_2, \eta_I)),$$

where *Append* is defined and discussed in section 5.2.1, and *removeNonEscaping* is defined in section 5.3.1. The function η_I is defined in Fig. A.2.

As shown in Fig. A.2, η_I has all the rules of $\eta[\![\tau_2, \tau_1]\!]$ (defined in section 5.2.1), namely (A.1)–(A.4). In addition to it, it is augmented with rules borrowed from the *Simplify* operation. The net result is that η_I partially computes the mapping *Incl* used by the *Simplify* operation.

The composition operation invokes *removeNonEscaping* without performing a full blown *Simplify* operation. This is possible because of the domain invariant. Lemma A.2 establishes the correctness of this operation by proving equivalence to $Simplify(\tau_2\langle\!\langle\tau_1\rangle\!\rangle_S)$.

Correctness, Equivalence and Termination of $\mathcal{F}_{\mathcal{I}}$

Lemma A.1. Let S be a field-read statement of the form $v_1 = v_2.f$ and $\tau \in \mathcal{A}_{\mathcal{I}}$. The transfer functions: $\mathcal{F}_{\mathcal{I}}(S)$, $\mathcal{F}_{\mathcal{G}}(S)$, and $\mathcal{F}_{\mathcal{G}}(S)$ composed with *Simplify*, are all equivalent. That is,

$$\gamma_T(\mathcal{F}_{\mathcal{G}}(S)(\tau)) = \gamma_T(\mathit{Simplify}(\mathcal{F}_{\mathcal{G}}(S)(\tau))) = \gamma_T(\mathcal{F}_{\mathcal{I}}(S)(\tau))$$

Lemma A.2. Let $\tau_1, \tau_2 \in \mathcal{A}_{\mathcal{I}}$ and let S be a call statement. The composition operation $\langle\!\langle\rangle\!\rangle_I$ is equivalent to the base composition operation ($\langle\!\langle\rangle\!\rangle$), and also to $\langle\!\langle\rangle\!\rangle$ composed with *Simplify*.

$$\gamma_T(\tau_2\langle\!\langle\tau_1\rangle\!\rangle_S) = \gamma_T(\mathit{Simplify}_S(\tau_2\langle\!\langle\tau_1\rangle\!\rangle_S)) = \gamma_T(\tau_2\langle\!\langle\tau_1\rangle\!\rangle_I).$$

Lemma A.3. If $\tau \in \mathcal{A}_{\mathcal{I}}$ then

a. $\mathcal{F}_{\mathcal{I}}(v_1 = v_2.f)(\tau)$ is monotonic w.r.t $\sqsubseteq_{co}$.

b. The composition operation $\tau_1\langle\!\langle\tau_2\rangle\!\rangle_I$, and the abstract semantics of the call statement are monotonic w.r.t $\sqsubseteq_{co}$.

Theorem A.4. For any instantiation $(\mathcal{A}, \mathcal{F}_{\mathcal{A}})$ of $(\mathcal{A}_{\mathcal{I}}, \mathcal{F}_{\mathcal{I}})$ satisfying Assumption 5.10

a. There exists an instantiation of $(\mathcal{A}_{\mathcal{G}}, \mathcal{F}_{\mathcal{G}})$ semantically equivalent to $(\mathcal{A}, \mathcal{F}_{\mathcal{A}})$ and vice versa. Therefore, $(\mathcal{A}, \mathcal{F}_{\mathcal{A}})$ is correct.

b. $(\mathcal{A}, \mathcal{F}_{\mathcal{A}})$ will terminate since the transfer functions are monotonic.

B

The Node Merging Abstraction

In this section we formalize the node merging abstraction informally described in section 6.3.1.

Transformer Graph Embedding We say that a transformer graph τ_1 can be embedded in τ_2, denoted $\tau_1 \preceq \tau_2$, iff there exists a homomorphism h from the vertices τ_1 to τ_2 such that the following holds. (We use subscripts $_1$ and $_2$ to denote components of τ_1 and τ_2.)

$$x \in \mathsf{IV}_1 \Rightarrow h(x) \in \mathsf{IV}_2$$
$$x \in \mathsf{EV}_1 \Rightarrow h(x) \in \mathsf{EV}_2$$
$$\langle x, f, y\rangle \in \mathsf{IE}_1 \Rightarrow \langle h(x), f, h(y)\rangle \in \mathsf{IE}_2$$
$$\langle x, f, y\rangle \in \mathsf{EE}_1 \Rightarrow \langle h(x), f, h(y)\rangle \in \mathsf{EE}_2$$
$$\forall v \in \mathit{Vars}.\ \{h(x) \mid x \in \sigma_1(v)\} \subseteq \sigma_2(v)$$
$$\forall v \in \mathit{Vars}.\ \{h(x) \mid x \in \sigma_{in1}(v)\} \subseteq \sigma_{in2}(v)$$
$$(\langle u, f, w\rangle, \langle x, g, y\rangle) \in \rightsquigarrow_1 \Rightarrow (\langle h(u), f, h(w)\rangle, \langle h(x), g, h(y)\rangle) \in \rightsquigarrow_2$$

We use $\tau_1 \preceq_h \tau_2$ to denote that the function h induces an embedding from τ_1 to τ_2. (Unlike the embedding notions used by TVLA [Sagiv et al., 1999], here h is not required to be surjective.)

Node merging produces an embedding. Assume that we are given a function ξ representing an equivalence relation on the nodes of a transformer graph τ. ξ maps a node in τ to the representative of its equivalence class. We define the transformer graph $\xi(\tau)$ to be the transformer graph obtained by replacing every node u by a unique representative of its equivalence class in every component of τ (we formally define $\xi(\tau)$ later).

Lemma B.1. The embedding operation satisfies the following:

a. $\preceq$ is a pre-order. (i.e., it is reflexive and transitive).

b. γ_T is monotonic with respect to $\preceq$: $\forall \tau_a, \tau_b \in \mathcal{A}_\mathcal{G}$,
$\tau_a \preceq \tau_b \Rightarrow \gamma_T(\tau_a) \sqsubseteq_c \gamma_T(\tau_b)$

c. $\tau \preceq \xi(\tau)$.

Assume that we wish to replace a transformer graph τ by a graph $\xi(\tau)$ at some point during the analysis (perhaps by incorporating this into one of the abstract operations). Our earlier correctness argument still remains valid (since if $f \sim \tau_1 \preceq \tau_2$, then $f \sim \tau_2$). (The proof is essentially a simple inductive proof: see Prop. 4.3 of Cousot and Cousot [1992]).

However, this optimization impacts the termination argument because node merging is not monotonic w.r.t containment ordering. Indeed, our initial implementation of the optimization did not terminate for one program because the computation ended up with a cycle of equivalent, but different, transformers (in the sense of having the same concretization).

Informally, we get around this problem by refining the analysis to ensure that once two nodes are chosen to be merged together, they are always merged together in all subsequent steps. This approach guarantees termination. We formalize this approach in the sequel. The formalization adds an equivalence relation on nodes (representing the nodes currently merged together) to the abstract domain and updates the transformers accordingly.

The Equivalence Relations Domain We represent an equivalence relation on N_a (the set of abstract nodes) using a function $\xi : N_a \mapsto N_a$ that maps every node in an equivalence class to the (unique) representative of the equivalence class. Let $\mathcal{E} \subseteq N_a \mapsto N_a$ be the set of such functions representing the equivalence relations on N_a. Given a function $\xi \in \mathcal{E}$ it is straight forward to construct the equivalence relation it represents (denote as $\simeq_\xi$), $\simeq_\xi = \{(x, y) \mid \xi(x) = \xi(y)\}$. For every $\xi \in \mathcal{E}$ let $\xi^{-1} : N_a \mapsto 2^{N_a}$ denote the inverse function that maps the representative of an equivalence class to the set of nodes it represents.

Let $\leq$ be a partial order on $\mathcal{E}$ corresponding to the refinement order of equivalence relations, i.e., $\xi_1 \leq \xi_2$ iff $\simeq_{\xi_1} \subseteq \simeq_{\xi_2}$. Let $\sqcup$ denote the corresponding join operator: for all $\xi_1, \xi_2 \in \mathcal{E}$, $\xi_1 \sqcup \xi_2$ is the function representing the equivalence relation on N_a that is a superset of $\simeq_{\xi_1}$ and $\simeq_{\xi_2}$.

Lemma B.2. $(\mathcal{E}, \leq, \sqcup)$ is a join semi-lattice. The least element of $(\mathcal{E}, \leq)$ is $\lambda x.x$.

The Abstract Domain

Define an abstract domain $\tilde{\mathcal{A}}$ as the set of pairs $(\tau, \xi) \in \mathcal{A}_\mathcal{G} \times \mathcal{E}$ satisfying the following condition:

$$x \in (\mathsf{EV} \cup \mathsf{IV}) \implies \xi(x) = x$$

where $\tau = (\mathsf{EV}, \mathsf{EE}, \sigma_{in}, \mathsf{IV}, \mathsf{IE}, \sigma, \rightsquigarrow)$. The above condition implies that in every $(\tau, \xi) \in \tilde{\mathcal{A}}$, the vertices in τ are the representatives of the equivalence classes of $\simeq_\xi$.

The equivalence classes are kept around in the abstract state along with transformer graphs to remember the nodes that were merged (and replaced by their representatives) so that these nodes are merged again in the future if they show up.

Let $\tau \in \mathcal{A}$ and let $\mu : N_a \mapsto 2^{N_a}$ be any function. Let $\mu(\tau)$ be the transformer graph obtained by replacing v by $\mu(v)$ in every component of τ. Formally, $\mu(\tau)$ is defined as: $(\hat{\mu}(\mathsf{EV}), \hat{\mu}(\mathsf{EE}), \mu(\sigma_{in}), \hat{\mu}(\mathsf{IE}), \hat{\mu}(\mathsf{IV}), \mu(\sigma), \bigcup_{ie \rightsquigarrow ee} \mu(ie) \times \mu(ee))$, where $\mu(\langle x, f, y \rangle) = \mu(x) \times \{f\} \times \mu(y)$, $\mu(\sigma) = \lambda \mathtt{x}.\hat{\mu}(\sigma(\mathtt{x}))$, and $\hat{\mu}(S) = \bigcup_{x \in S} \mu(x)$.

Depending on the definition of μ, $\mu(\tau)$ may collapse multiple vertices in τ into a single vertex or blow-up a vertex in τ to a set of vertices. For instance, given a function $\xi \in \mathcal{E}$, $\xi(\tau)$ will replace every vertex in τ by the representative of the equivalence class it belongs to in $\simeq_\xi$. (To be mathematically precise, in $\xi(\tau)$ we interpret ξ as a function from $N_a \mapsto 2^{N_a}$.) To the contrary, $\xi^{-1}(\tau)$ will replace the representative of an equivalence class by the vertices it represents.

Define a partial order $\leq_m$ on $\tilde{\mathcal{A}}$ as follows: $(\tau_1, \xi_1) \leq_m (\tau_2, \xi_2)$ iff $\xi_1 \leq \xi_2$ and $\tau_1 \preceq_{\xi_2} \tau_2$. Let $\sqcup_m$ denote the join operation with respect to $\leq_m$. We define the concretization function $\gamma_m : \tilde{\mathcal{A}} \mapsto \mathcal{C}$ as $\gamma_m(\tau, \xi) = \gamma_T(\tau)$.

Node Merging

With every edge and vertex of the control flow graph we associate a node merging operation $NM_i : \tilde{\mathcal{A}} \mapsto \tilde{\mathcal{A}}$, where i is a statement or vertex of the control flow graph, defined as follows:

$$
\begin{aligned}
&NM_i(\tau, \xi) = \\
&\quad \text{let } \xi' = \mathit{NodesToMerge}_i(\tau) \\
&\quad \text{let } \xi'' = \xi \sqcup \xi' \\
&\quad (\xi''(\tau), \xi'')
\end{aligned}
$$

The node merging operation is parameterized by a function $\mathit{NodesToMerge}_i$. It returns an equivalence relation whose equivalence classes specify the nodes that have to be merged together. The following results hold for any arbitrary definition of $\mathit{NodesToMerge}_i$.

Lemma B.3. If $(\tau, \xi) \in \tilde{\mathcal{A}}$ and $(\tau', \xi') = NM_i(\tau, \xi)$ then $\tau \preceq \tau'$

For ensuring termination we require that $\mathit{NodesToMerge}_i$ is monotonic with respect to the embedding operation i.e, for all $\tau_1, \tau_2 \in \mathcal{A}$, if $\tau_1 \preceq \tau_2$ then $\mathit{NodesToMerge}_i(\tau_1) \leq \mathit{NodesToMerge}_i(\tau_2)$. This property will be satisfied if the nodes are chosen for merging based on the properties of the transformer graph that are preserved by the embedding, e.g. based on reachability of nodes.

Semantics Equations

$$\vartheta_v = (\tau_{id}, \lambda x.x) \qquad v \text{ is an entry vertex} \tag{B.1}$$

$$\vartheta_v = \sqcup_m(\{\vartheta_{u,v} \mid u \xrightarrow{S} v\}) \qquad v \text{ is not an entry vertex} \tag{B.2}$$

$$\vartheta_{u,v} = \mathcal{F}_{\tilde{\mathcal{A}}}(S)(\vartheta_u) \qquad u \xrightarrow{S} v, \text{ S is not a call statement} \tag{B.3}$$

$$\vartheta_{u,v} = \mathcal{F}_{\tilde{\mathcal{A}}}(S)(\vartheta_u, \vartheta_{exit(Q)}) \qquad u \xrightarrow{S} v, \text{ S is a call to Q} \tag{B.4}$$

Correctness preserving operations

$$NM_i \in \tilde{\mathcal{A}} \mapsto \tilde{\mathcal{A}} \qquad i \text{ is a statement or a vertex}$$

Figure B.1: The abstract semantics that incorporates the node merging operation.

The Abstract Semantics

The abstract semantics equations that incorporate the node merging operation are shown in Figure B.1. Let S be a primitive statement. $\mathcal{F}_{\tilde{\mathcal{A}}}(S) : \tilde{\mathcal{A}} \mapsto \tilde{\mathcal{A}}$ is defined as:

$$\mathcal{F}_{\tilde{\mathcal{A}}}(S)(\tau, \xi) = (\xi(\mathcal{F}_{\mathcal{G}}(S)(\xi^{-1}(\tau))), \xi)$$

If S is a call to procedure Q, then

$$\mathcal{F}_{\tilde{\mathcal{A}}}(S)((\tau_r, \xi_r), (\tau_e, \xi_e))) = \quad \text{let } \xi' = \xi_r \sqcup \xi_e \\ (\xi'(\mathcal{F}_{\mathcal{G}}(S)({\xi_r}^{-1}(\tau_r), {\xi_e}^{-1}(\tau_e))), \xi')$$

In simple words, the abstract transfer function $\mathcal{F}_{\tilde{\mathcal{A}}}(S)$ first constructs a blown-up transformer graph $\xi^{-1}(\tau)$ by reintroducing all the vertices/edges that may have been merged in τ. (Note that $\xi^{-1}(\tau)$ and τ are semantically equivalent i.e, have the same concrete image as each embed in the other.) It then applies the base semantics $\mathcal{F}_{\mathcal{G}}(S)$, and collapses the merged vertices/edges again by applying ξ on the resulting transformer graph.

The application of ξ^{-1} to the input transformer graph is necessary in order to ensure the monotonicity of $\mathcal{F}_{\tilde{\mathcal{A}}}$ with respect to $\leq_m$. This is a fall out of the fact that for any arbitrary definition of the parameters we are only guaranteed that $\mathcal{F}_{\mathcal{G}}(S)$ is monotonic with respect to the

containing ordering $\sqsubseteq_{co}$. However, for many practical instantiations of the parameter, the transfer function $\mathcal{F}_{\mathcal{G}}(S)$ would be monotonic with respect to $\preceq_{\xi}$ (for any $\xi \in \mathcal{E}$). In such cases, as an optimization we can eliminate the application of ξ^{-1} from the above definition and obtain a more efficient semantics.

Notice that when no nodes are merged during the analysis, the above semantics is equivalent to $(\mathcal{A}_{\mathcal{G}}, \mathcal{F}_{\mathcal{G}})$, since the *NM* operation at every statement and vertex reduces to an identity function and the equivalence relation remains unchanged (from the initial identify relation) throughout the analysis.

Correctness and Termination of the Node Merging Abstraction

The correctness of the semantics $(\tilde{\mathcal{A}}, \mathcal{F}_{\tilde{\mathcal{A}}})$ is straightforward to establish. The ξ component of the abstract state is not relevant for proving correctness (see the definition of γ_m). The τ component of the abstract state is updated by the transfer functions $\mathcal{F}_{\tilde{\mathcal{A}}}$ by applying the base transfer function $\mathcal{F}_{\mathcal{G}}$, followed and preceded by the application of ξ and ξ^{-1}, respectively (for some $\xi \in \mathcal{E}$). However, the application of ξ or ξ^{-1} on a transformer graph is correctness preserving because τ can be embedded in both $\xi(\tau)$ and $\xi^{-1}(\tau)$. Hence, $\mathcal{F}_{\tilde{\mathcal{A}}}$ is correct. Below we formalize the termination of the abstract semantics $\mathcal{F}_{\tilde{\mathcal{A}}}$.

Assumption B.4. *NodesToMerge*$_i$ terminates on all inputs and is monotonic w.r.t the embedding operation $\preceq$.

Lemma B.5. If the Assumptions B.4 and5.10 hold,

a. For every statement S, $\mathcal{F}_{\tilde{\mathcal{A}}}(S)$ is monotonic w.r.t $\leq_m$.

b. For every vertex of the control flow graph or statement i, NM_i is monotonic with respect to $\leq_m$.

Theorem B.6. Every instantiation of $(\tilde{\mathcal{A}}, \mathcal{F}_{\tilde{\mathcal{A}}})$ satisfying assumptions B.4 and 5.10 terminates.

References

Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, pages 243–262, 2009.

Marcio Buss, Daniel Brand, Vugranam C. Sreedhar, and Stephen A. Edwards. Flexible pointer analysis using assign-fetch graphs. In *SAC*, pages 234–239, 2008.

Marcio Buss, Daniel Brand, Vugranam C. Sreedhar, and Stephen A. Edwards. A novel analysis space for pointer analysis and its application for bug finding. *Sci. Comput. Program.*, 75(11):921–942, 2010.

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300, 2009.

Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant context inference. In *POPL*, pages 133–146, 1999.

Ben-Chung Cheng and Wen-Mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *PLDI*, pages 57–69, 2000.

Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

Patrick Cousot and Radhia Cousot. Modular static program analysis. In *CC*, pages 159–178, 2002.

Manuvir Das. Unification-based pointer analysis with directional assignments. In *PLDI*, pages 35–46, 2000.

Arnab De and Deepak D'Souza. Scalable flow-sensitive pointer analysis for java with strong updates. In *ECOOP*, pages 665–687, 2012.

Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI*, pages 567–577, 2011.

Bhargav S. Gulavani, Supratik Chakraborty, Ganesan Ramalingam, and Aditya V. Nori. Bottom-up shape analysis. In *SAS*, pages 188–204, 2009.

Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *CGO*, pages 289–298, 2011.

Bertrand Jeannet, Alexey Loginov, Thomas Reps, and Mooly Sagiv. A relational approach to interprocedural shape analysis. *ACM Trans. Program. Lang. Syst.*, 32:5:1–5:52, 2010.

Etienne Kneuss, Viktor Kuncak, and Philippe Suter. Effect analysis for programs with callbacks. In *VSTTE*, pages 48–67, 2013.

Jens Knoop and Bernhard Steffen. The interprocedural coincidence theorem. In *CC*, pages 125–140, 1992.

Chris Lattner and Vikram Adve. *Macroscopic Data Structure Analysis and Operations*. PhD thesis, University of Illinois at Urbana-Champaign, 2005a.

Chris Lattner and Vikram S. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI*, pages 129–142, 2005b.

Chris Lattner, Andrew Lenharth, and Vikram S. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI*, pages 278–289, 2007.

Ondrej Lhoták. *Program Analysis Using Binary Decision Diagrams*. PhD thesis, 2006.

Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *POPL*, pages 3–16, 2011.

Donglin Liang and Mary Jean Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *SAS*, pages 279–298, 2001.

Percy Liang and Mayur Naik. Scaling abstraction refinement via pruning. In *PLDI*, pages 590–601, 2011.

Ravichandhran Madhavan, Ganesan Ramalingam, and Kapil Vaswani. Purity analysis: An abstract interpretation formulation. In *SAS*, pages 7–24, 2011.

Ravichandhran Madhavan, G. Ramalingam, and Kapil Vaswani. Modular heap analysis for higher-order programs. In *SAS*, pages 370–387, 2012.

Ravi Mangal, Mayur Naik, and Hongseok Yang. A correspondence between two approaches to interprocedural analysis in the presence of join. In *ESOP*, pages 513–533, 2014.

Mark Marron, Ondrej Lhoták, and Anindya Banerjee. Programming paradigm driven heap analysis. In *CC*, pages 41–60, 2012.

Erik M. Nystrom, Hong-Seok Kim, and Wen mei W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *SAS*, pages 165–180, 2004.

Prakash Prabhu, Ganesan Ramalingam, and Kapil Vaswani. Safe programmable speculative parallelism. In *PLDI*, pages 50–61, 2010.

Noam Rinetzky, Mooly Sagiv, and Eran Yahav. Interprocedural shape analysis for cutpoint-free programs. In *SAS*, pages 284–302, 2005.

Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, pages 105–118, 1999.

Alexandru D. Salcianu. Pointer analysis and its applications for java programs. Master's thesis, Massachusetts institute of technology, 2001.

Alexandru D. Salcianu and Martin C. Rinard. Purity and side effect analysis for java programs. In *VMCAI*, pages 199–215, 2005.

Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–234, 1981.

Yannis Smaragdakis and Martin Bravenboer. Using datalog for fast and easy program analysis. In *Datalog*, pages 245–251, 2010.

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *POPL*, pages 17–30, 2011.

Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: context-sensitivity, across the board. In *PLDI*, page 50, 2014.

Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41, 1996.

WALA. T. J. Watson libraries for program analysis. URL `https://github.com/wala/WALA`.

John Whaley and Martin C. Rinard. Compositional pointer and escape analysis for java programs. In *OOPSLA*, pages 187–206, 1999.

Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. On abstraction refinement for program analyses in datalog. In *PLDI*, page 27, 2014.