Modular Lattices for Compositional Interprocedural Analysis

Ghila Castelnuovo, Tel-Aviv University
Mayur Naik, Georgia Institute of Technology
Noam Rinetzky, Mooly Sagiv, Tel-Aviv University
Hongseok Yang, University of Oxford

Abstract

Interprocedural analyses are *compositional* when they compute over-approximations of procedures in a bottom-up fashion. These analyses are usually more scalable than top-down analyses which compute a different procedure summary for every calling context. However, compositional analyses are rare in practice, because it is difficult to develop such analyses with enough precision. In this paper, we establish a connection between a restricted class of compositional analyses and so called *modular lattices*, which require certain associativity between the lattice join and meet operations. Our connection provides sufficient conditions for building a compositional analysis that is as precise as a top-down analysis.

We developed a compositional version of the connection pointer analysis by Ghiya and Hendren which is slightly more conservative than the original top-down analysis in order to meet our modularity requirement. We implemented and applied our compositional connection analysis to real-world Java programs. As expected, the compositional analysis scales much better than the original top-down version. The top-down analysis times out in the largest two of our five programs, and the loss of precision due to the modularity requirement in the remaining programs ranges only between 2-5%.

1. Introduction

Scaling program analysis to large programs is an ongoing challenge for program verification. Typical programs include many relatively small procedures. Therefore, a promising direction for scalability is analyzing each procedure in isolation, using pre-computed summaries for called procedures and computing a summary for the analyzed procedure. Such analyses are called *bottom-up interprocedural analysis* or *compositional analysis*. Notice that the analysis of the procedure itself need not be compositional and can be costly. Indeed, bottom-up interprocedural analyses have been found to scale well [3, 5, 8, 14, 21].

The theory of bottom-up interprocedural analysis has been studied in [7]. In practice, designing and implementing a bottom-up interprocedural analysis is challenging for several reasons: it requires accounting for all potential calling contexts of a procedure in a sound and precise way; the summary of the procedures can be quite large leading to infeasible analyzers; and it may be costly to instantiate procedure summaries. An example of the challenges underlying bottom-up interprocedural analysis is the unsound original formulation of the compositional pointer analysis algorithm in [21]. A corrected version of the algorithm was subsequently proposed in [19] and recently proven sound in [15] using abstract interpretation. In contrast, top-down interprocedural analysis [6, 17, 20] is much better understood and has been integrated into existing tools such as SLAM [1], Soot [2], WALA [9], and Chord [16].

This paper contributes to a better understanding of bottom-up interprocedural analysis. Specifically, we attempt to characterize the cases under which bottom-up and top-down interprocedural analyses yield the same results. To guarantee scalability, we limit the discussion to cases in which bottom-up and top-down analyses use the same underlying abstract domains.

We use *connection analysis* [11], which was developed in the context of parallelizing sequential code, as a motivating example of our approach. Connection analysis is a kind of pointer analysis that aims to prove that two references can never point to the same weakly-connected heap component, and thus ignores the direction of pointers. Despite its conceptual simplicity, connection analysis is flow- and context-sensitive, and the effect of program statements is non-distributive. In fact, the top-down interprocedural connection analysis is exponential, and indeed our experiments indicate that this analysis scales poorly.

Main Contributions. The main results of this paper can be summarized as follows:

- We formulate a sufficient condition on the effect of commands on abstract states that guarantees bottom-up and top-down interprocedural analyses will yield the same results. The condition is based on lattice theory. Roughly speaking, the idea is that the abstract semantics of primitive commands and procedure calls and returns can only be expressed using meet and join operations with constant elements, and that elements used in the meet must be *modular* in a lattice theoretical sense [13].
- We formulate a variant of the connection analysis in a way that satisfies the above requirements. The main idea is to over-approximate the treatment of variables that point to null in all program states that occur at a program point.
- We implemented two versions of the top-down interprocedural connection analysis for Java programs in order to measure the extra loss of precision of our over-approximation. We also implemented the bottom-up interprocedural analysis for Java programs. We report empirical results for five benchmarks of sizes 15K–310K bytecodes. The original top-down analysis times out in over six hours on the largest two benchmarks. For the remaining three benchmarks, only 2-5% of precision was lost by our bottom-up analysis due to the modularity requirement compared to the original top-down version.

This work is based on the master thesis of [4] which contains additional experiments, elaborations, and proofs.

2. Informal Explanation

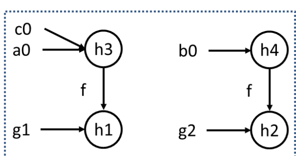
This section presents the use of modular lattices for compositional interprocedural program analyses in an informal manner.

2.1 A Motivating Example

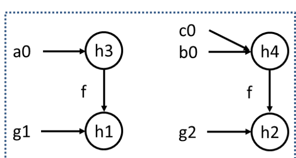
Fig. 1 shows a schematic artificial program illustrating the potential complexity of interprocedural analysis. The main procedure invokes procedure $\mathtt{p_0}$, which invokes $\mathtt{p_1}$ with an actual parameter $\mathtt{a_0}$ or $\mathtt{b_0}$. For every $1 \leq i \leq n$, procedure $\mathtt{p_i}$ either assigns $\mathtt{a_i}$ with formal parameter $\mathtt{c_{i-1}}$ and invokes procedure $\mathtt{p_{i+1}}$ with an actual parameter $\mathtt{a_i}$ or assigns $\mathtt{b_i}$ with formal parameter $\mathtt{c_{i-1}}$ and invokes procedure $\mathtt{p_{i+1}}$ with an actual parameter $\mathtt{b_i}$. Procedure $\mathtt{p_n}$ either assigns $\mathtt{a_n}$ or $\mathtt{b_n}$ with formal parameter $\mathtt{c_{n-1}}$. Fig. 2 depicts the two

```
// a0,...,an,b0,...,bn,g1, and g2 are static variables

static main() {
  g1 = new h1; g2 = new h2; a0 = new h3; b0 = new h4;
  a0.f = g1; b0.f = g2;
  p0();
}

p0() {if(*) p1(a0) else p1(b0)}

p1(c0) {
  if(*) {a1 = c0; p2(a1)} else  {b1 = c0; p2(b1)} }

p2(c1) {
  if(*) {a2 = c1; p3(a2)} else  {b2 = c1; p3(b2)} }
...
pn-1(cn-2) {
    if(*) {an-1 = cn-2; pn(an-1)}
    else  {bn-1 = cn-2; pn(bn-1)}
}

pn(cn-1) if(*) {an = cn-1 else bn = cn-1}
```

Figure 1. Example program.



$$d_{conn} = \{\{\mathtt{g}_1, \mathtt{a_0}, \mathtt{c_0}\}, \{\mathtt{g}_2, \mathtt{b_0}\}, \{\mathtt{a_1}\}, \{\mathtt{b_1}\}, \ldots, \{\mathtt{a_n}\}, \{\mathtt{b_n}\}\}$$
$$d'_{point} = \{\langle \mathtt{a_0}, \mathtt{h_3}\rangle, \langle \mathtt{b_0}, \mathtt{h_4}\rangle, \langle \mathtt{c_0}, \mathtt{h_3}\rangle, \langle \mathtt{h_3}, \mathtt{f}, \mathtt{h_1}\rangle, \langle \mathtt{h_4}, \mathtt{f}, \mathtt{h_2}\rangle, \langle \mathtt{g}_1, \mathtt{h_1}\rangle, \langle \mathtt{g}_2, \mathtt{h_2}\rangle\}$$



$$d_{conn} = \{\{\mathtt{g}_1, \mathtt{a_0}\}, \{\mathtt{g}_2, \mathtt{b_0}, \mathtt{c_0}\}, \{\mathtt{a_1}\}, \{\mathtt{b_1}\}, \ldots, \{\mathtt{a_n}\}, \{\mathtt{b_n}\}\}$$
$$d'_{point} = \{\langle \mathtt{a_0}, \mathtt{h_3}\rangle, \langle \mathtt{b_0}, \mathtt{h_4}\rangle, \langle \mathtt{c_0}, \mathtt{h_4}\rangle, \langle \mathtt{h_3}, \mathtt{f}, \mathtt{h_1}\rangle, \langle \mathtt{h_4}, \mathtt{f}, \mathtt{h_2}\rangle, \langle \mathtt{g}_1, \mathtt{h_1}\rangle, \langle \mathtt{g}_2, \mathtt{h_2}\rangle\}$$

Figure 2. Concrete states at the entry of procedure $\mathtt{p_1}$ (see Fig. 1) and the corresponding connection and points-to abstractions.

concrete states that can occur when the procedure $\mathtt{p_1}$ is invoked. There are two different concrete states corresponding to the then- and the else-branch in $\mathtt{p_0}$.

2.2 Connection Analysis and Points-to Analysis

Connection Analysis. We say that two heap objects are *connected* in a state when we can reach from one object to the other by following fields forward or backward. Two variables are *connected* when they point to connected heap objects.

The goal of the connection analysis is to soundly estimate connection relationships between variables. The abstract states d of the analysis are families $\{X_i\}_{i \in I}$ of disjoint sets of variables. Two variables x, y are in the same set X_i, which we call a *connection set*, when x and y may be connected. Fig. 2 depicts two abstract states at the entry of procedure $\mathtt{p_1}$. There are two calling contexts for the procedure. In the first one, $\mathtt{a_0}$ and $\mathtt{c_0}$ point to the same heap object, whose `f` field goes to the object pointed to by $\mathtt{g}_1$. In addition to these two objects, there are two further ones, pointed to by $\mathtt{b_0}$ and $\mathtt{g}_2$ respectively, where the `f` field of the object pointed to by $\mathtt{b_0}$ points to the object pointed to by $\mathtt{g}_2$. As a result, there are two connection sets $\{\mathtt{a_0}, \mathtt{g}_1, \mathtt{c_0}\}$ and $\{\mathtt{b_0}, \mathtt{g}_2\}$. The second calling context is similar to the first, except that $\mathtt{c_0}$ is aliased with $\mathtt{b_0}$ instead of $\mathtt{a_0}$. The connection sets are changed accordingly, and they are $\{\mathtt{a_0}, \mathtt{g}_1\}$ and $\{\mathtt{b_0}, \mathtt{g}_2, \mathtt{c_0}\}$. In both cases, the other variables are pointing to `null`, and thus are not connected to any variable.

$$
\begin{aligned}
d_1 &= \{\{\mathtt{g}_1, \mathtt{a_0}, \mathtt{a_1}, \mathtt{a_2}, \ldots, \mathtt{a_{n\text{-}1}}, \mathtt{c_{n\text{-}1}}\}, \{\mathtt{g}_2, \mathtt{b_0}\}, \\
&\quad \{\mathtt{a_n}\}, \{\mathtt{b_1}\}, \ldots, \{\mathtt{b_{n\text{-}1}}\}, \{\mathtt{b_n}\}\} \\
d_2 &= \{\{\mathtt{g}_1, \mathtt{a_0}, \mathtt{a_1}, \mathtt{a_2}, \ldots, \mathtt{b_{n\text{-}1}}, \mathtt{c_{n\text{-}1}}\}, \{\mathtt{g}_2, \mathtt{b_0}\}, \\
&\quad \{\mathtt{a_{n\text{-}1}}\}, \{\mathtt{a_n}\}, \{\mathtt{b_1}\}, \ldots, \{\mathtt{b_n}\}\} \\
&\ldots \\
d_{(2^n-1)} &= \{\{\mathtt{g}_1, \mathtt{a_0}, \mathtt{b_1}, \mathtt{b_2}, \ldots, \mathtt{b_{n\text{-}1}}, \mathtt{c_{n\text{-}1}}\}, \{\mathtt{g}_2, \mathtt{b_0}\}, \\
&\quad \{\mathtt{a_1}\}, \ldots, \{\mathtt{a_{n\text{-}1}}\}, \{\mathtt{a_n}\}, \{\mathtt{b_n}\}\} \\
d_{(2^{n-1}+1)} &= \{\{\mathtt{g}_1, \mathtt{a_0}\}, \{\mathtt{g}_2, \mathtt{b_0}, \mathtt{a_1}, \mathtt{a_2}, \ldots, \mathtt{a_{n\text{-}1}}, \mathtt{c_{n\text{-}1}}\}, \\
&\quad \{\mathtt{a_n}\}, \{\mathtt{b_1}\}, \ldots, \{\mathtt{b_{n\text{-}1}}\}, \{\mathtt{b_n}\}\} \\
&\ldots \\
d_{2^n} &= \{\{\mathtt{g}_1, \mathtt{a_0}\}, \{\mathtt{g}_2, \mathtt{b_0}, \mathtt{b_1}, \mathtt{b_2}, \ldots, \mathtt{b_{n\text{-}1}}, \mathtt{c_{n\text{-}1}}\}, \\
&\quad \{\mathtt{a_1}\}, \ldots, \{\mathtt{a_{n\text{-}1}}\}, \{\mathtt{a_n}\}, \{\mathtt{b_n}\}\}
\end{aligned}
$$

Figure 3. Connection abstraction at the entry of procedure $\mathtt{p_n}$ of the program in Fig. 1.

Points-to Analysis. The purpose of the points-to analysis is to compute points-to relations between variables and objects (which are represented by allocation sites). The analysis expresses points-to relations as a set of tuples of the form $\langle \mathtt{x}, \mathtt{h}\rangle$ or $\langle \mathtt{h_1}, \mathtt{f}, \mathtt{h_2}\rangle$. The pair $\langle \mathtt{x}, \mathtt{h}\rangle$ means that variable `x` may point to an object allocated at the site `h`, and the tuple $\langle \mathtt{h_1}, \mathtt{f}, \mathtt{h_2}\rangle$ means that the `f` field of an object allocated at $\mathtt{h_1}$ may point to an object allocated at $\mathtt{h_2}$. Fig. 2 depicts the abstract states at the entry to procedure $\mathtt{p_1}$. Also in this case, there are two calling abstract contexts for $\mathtt{p_1}$. In one of them, $\mathtt{c_0}$ may point to $\mathtt{h_3}$, and in the other, $\mathtt{c_0}$ may point to $\mathtt{h_4}$.

2.3 Top-Down Interprocedural Analysis

A standard approach for the top-down interprocedural analysis is to analyze each procedure once for each different calling context. This approach often has scalability problems. One of the reasons is the large number of different calling contexts that arise. In the program shown in Fig. 1, for instance, for each procedure $\mathtt{p_i}$ there are two calls to procedure $\mathtt{p_{i+1}}$, where for each one of them, the connection and the points-to analyses compute two different calling contexts for procedure $\mathtt{p_{i+1}}$. Therefore, in both the analyses, the number of calling contexts at the entry of procedure $\mathtt{p_i}$ is 2^i.

Fig. 3 shows the connection-abstraction at the entry of procedure $\mathtt{p_n}$. Each abstract state in the abstraction corresponds to one path to $\mathtt{p_n}$. For example, the first state corresponds to selecting the then-branch in all $\mathtt{p_0}$,...,$\mathtt{p_{n\text{-}1}}$, while the second state corresponds to selecting the then-branch in all $\mathtt{p_0}$,...,$\mathtt{p_{n\text{-}2}}$, and the else-branch in $\mathtt{p_{n\text{-}1}}$. Finally, the last state corresponds to selecting the else-branch in all $\mathtt{p_0}$,...,$\mathtt{p_{n\text{-}1}}$.

2.4 Bottom-Up Compositional Interprocedural Analysis

Bottom-up compositional analyses avoid the explosion of calling context by computing for each procedure a summary which is independent of the input, and instantiating as a function of particular calling contexts. Unfortunately, it is hard to analyze a procedure independently of its calling contexts and at the same time compute a summary that is sound and precise enough. One of the reasons is that the abstract transfer functions may depend on the input abstract state, which is often unavailable for the compositional analysis. For example, in the program in Fig. 1, the abstract transformer for the assignment `ai = ci-1` in the points-to analysis is

$$[\![\mathtt{a_i} = \mathtt{c_{i\text{-}1}}]\!]^\sharp(d) = (d \backslash \{\langle \mathtt{a_i}, z\rangle | z \in \mathsf{Var}\}) \cup \{\langle \mathtt{a_i}, w\rangle | \langle \mathtt{c_{i\text{-}1}}, w\rangle \in d\}\,.$$

Note that the rightmost set depends on the input abstract state d.

2.5 Modular Lattices for Compositional Interprocedural Analysis

This paper formulates a sufficient condition for performing compositional interprocedural analysis using lattices theory. Our condition requires that the abstract domain be a lattice with a so-called *modularity* property, and that the effects of primitive commands (such as assignments) on abstract elements be expressed by applying the $\sqcap$ and $\sqcup$ operations to the input states. If this condition is met, we can construct a bottom-up compositional analysis that summarizes each procedure independently of particular inputs.

DEFINITION 1. *Let $\mathcal{D}$ be a lattice. A pair of elements $\langle d_0, d_1\rangle$ is called* **modular**, *denoted by $d_0 M d_1$, iff*

$$d \sqsubseteq d_1 \text{ implies that } (d \sqcup d_0) \sqcap d_1 = d \sqcup (d_0 \sqcap d_1)$$

An element d_1 is called **right-modular** *if $d_0 M d_1$ holds for all $d_0 \in \mathcal{D}$. $\mathcal{D}$ is called* **modular** *if $d_0 M d_1$ holds for all $d_0, d_1 \in \mathcal{D}$.*

Intuitively, a lattice is **modular** when it satisfies a restricted form of associativity between its $\sqcup$ and $\sqcap$ operations [13]. (Note, for example, that every distributive lattice is modular, but not all modular lattices are distributive.) In our application to the interprocedural analysis, the left-hand side of the equality in Def. 1 represents the top-down computation and the right-hand side corresponds to the bottom-up computation. Therefore, modularity ensures that the results coincide.

Our approach requires that transfer functions of primitive commands be defined by the combination of $- \sqcap d_0$ and $- \sqcup d_1$ for some constant abstract elements d_0 and d_1, independent of the input abstract state where d_0 elements are right-modular. Our encoding of points-to analysis described in Sec. 2.2 does not meet this requirement on transfer functions, because it does not use $\sqcup$ with a constant element to define the meaning of the statement $x = y$. In contrast, in connection analysis the transfer function of the statement $x = y$ is defined by

$$[\![x = y]\!]^\sharp = \lambda d.\, (d \sqcap S_x) \sqcup U_{xy}$$

where S_x, U_{xy} are fixed abstract elements and do not depend on the input abstract state d. In Sec. 4, we formally prove that the connection analysis satisfies both the modularity requirement and the requirement on the transfer functions.

We complete this informal description by illustrating how the two requirements lead to the coincidence between top-down and bottom-up analyses. Consider again the assignment $[\![\mathtt{a_i} = \mathtt{c_{i-1}}]\!]^\sharp$, in the body of some procedure $\mathtt{p_i}$. Let $\{d_k\}_k$ denote abstract states at the entry of $\mathtt{p_i}$, and suppose there is some d such that

$$\forall k : \exists d'_k \sqsubseteq S_{\mathtt{a_i}} : d_k = d \sqcup d'_k.$$

The compositional approach first chooses the input state d, and computes $[\![\mathtt{a_i} = \mathtt{c_{i-1}}]\!]^\sharp(d)$. This result is then adapted to any d_k by being joined with d'_k, whenever this procedure is invoked with the abstract state d_k. This adaptation of the bottom-up approach gives the same result as the top-down approach, which applies $[\![\mathtt{a_i} = \mathtt{c_{i-1}}]\!]^\sharp$ on d_k directly, as shown below:

$$\begin{array}{rcl} [\![\mathtt{a_i} = \mathtt{c_{i-1}}]\!]^\sharp(d) \sqcup d'_k & = & ((d \sqcap S_{\mathtt{a_i}}) \sqcup U_{\mathtt{a_i c_{i-1}}}) \sqcup d'_k \\ & = & ((d \sqcap S_{\mathtt{a_i}}) \sqcup d'_k) \sqcup U_{\mathtt{a_i c_{i-1}}} \\ & = & ((d \sqcup d'_k) \sqcap S_{\mathtt{a_i}}) \sqcup U_{\mathtt{a_i c_{i-1}}} \\ & = & (d_k \sqcap S_{\mathtt{a_i}}) \sqcup U_{\mathtt{a_i c_{i-1}}} \\ & = & [\![\mathtt{a_i} = \mathtt{c_{i-1}}]\!]^\sharp(d_k). \end{array}$$

The second equality uses the associativity and commutativity of the $\sqcup$ operator, and the third holds due to the modularity requirement.

3. Programming Language

Let PComm, G, L, and PName be sets of primitive commands, global variables, local variables, and procedure names, respectively. We use the following symbols to range over these sets:

$$a, b \in \mathsf{PComm}, \quad g \in \mathsf{G}, \quad x, y, z \in \mathsf{G} \cup \mathsf{L}, \quad p \in \mathsf{PName}.$$

We formalize our results for a simple imperative programming language with procedures:

$$\begin{array}{rrcl} \textit{Commands} & C & ::= & \mathtt{skip} \mid a \mid C;C \mid C+C \mid C^* \mid p() \\ \textit{Declarations} & D & ::= & \mathsf{proc}\ p() = \{\mathsf{var}\ \vec{x}; C\} \\ \textit{Programs} & P & ::= & \mathsf{var}\ \vec{g}; C \mid D; P \end{array}$$

A program P in our language is a sequence of procedure declarations, followed by a sequence of declarations of global variables and a main command. Commands contain primitive commands $a \in \mathsf{PComm}$, left unspecified, sequential composition $C; C'$, nondeterministic choice $C + C'$, iteration C^*, and procedure calls $p()$. We use $+$ and * instead of conditionals and while loops for theoretical simplicity: given appropriate primitive commands, conditionals and loops can be easily defined.

Declarations D give the definitions of procedures. A procedure is comprised of a sequence of local variables declarations $\vec{x}$ and a command, which we refer to as the procedure's *body*. Procedures do not take any parameters or return any values explicitly; values can instead be passed to and from procedures using global variables. To simplify presentation, we do not consider mutually recursive procedures in our language; direct recursion is allowed. We denote by C_{body_p} and L_p the body of procedure p and the set of its local variables, respectively.

We assume that L and G are fixed arbitrary finite sets. Also, we consider only well-defined programs where all the called procedures are defined.

Standard Semantics. The standard semantics propagates every caller's context to the callee's entry point and computes the effect of the procedure on each one of them. Formally,

$$[\![p()]\!]^\sharp(d) = [\![\mathsf{return}]\!]^\sharp(([\![C_{\mathsf{body}_p}]\!]^\sharp \circ [\![\mathsf{entry}]\!]^\sharp)(d), d)$$

where C_{body_p} is the body of the procedure p, and

$$[\![\mathsf{entry}]\!]^\sharp : \mathcal{D} \to \mathcal{D} \quad \text{and} \quad [\![\mathsf{return}]\!]^\sharp : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$$

are the functions which represent, respectively, entering and returning from a procedure.

Relational Collecting Semantics. The semantics of our programming language tracks pairs of memory states $\langle \overline{\sigma}, \sigma' \rangle$ coming from some unspecified set Σ of memory states. $\overline{\sigma}$ is the *entry* memory state to the procedure of the executing command (or if we are executing the main command, the memory state at the start of the program execution), and σ' is the *current* memory state. We assume that we are given the meaning $[\![a]\!] : \Sigma \to 2^\Sigma$ of every primitive command, and lift it to sets of pairs $\rho \subseteq \mathcal{R} = 2^{\Sigma \times \Sigma}$ of memory states by applying it in a pointwise manner to the current states:

$$[\![c]\!](\rho) = \{\langle \overline{\sigma}, \sigma' \rangle \mid \langle \overline{\sigma}, \sigma \rangle \in \rho \wedge \sigma' \in [\![c]\!](\sigma)\}\,.$$

The meaning of composed commands is standard:

$$\begin{array}{r} [\![C_1 + C_2]\!](\rho) = [\![C_1]\!](\rho) \cup [\![C_2]\!](\rho) \\ [\![C_1; C_2]\!](\rho) = [\![C_2]\!]([\![C_1]\!](\rho)) \\ [\![C^*]\!](\rho) = \mathsf{leastFix}\, \lambda\rho'.\, \rho \cup [\![C]\!](\rho'). \end{array}$$

The effect of procedure invocations is computed using the auxiliary functions entry, return, $\mathsf{combine}$, and $\cdot|_G$, which we explain below.

$$[\![p()]\!](\rho_c) = [\![\mathsf{return}]\!]([\![C_{\mathsf{body}_p}]\!] \circ [\![\mathsf{entry}]\!](\rho_c), \rho_c), \quad \text{where}$$

$$\llbracket \mathsf{entry} \rrbracket \; : \; \mathcal{R} \to \mathcal{R}$$
$$\llbracket \mathsf{entry} \rrbracket(\rho_c) = \{\langle \sigma_e, \sigma_e \rangle \mid \sigma_e = \sigma_c|_G \wedge \langle \overline{\sigma_c}, \sigma_c \rangle \in \rho_c\}$$

$$\llbracket \mathsf{return} \rrbracket \; : \; \mathcal{R} \times \mathcal{R} \to \mathcal{R}$$
$$\llbracket \mathsf{return} \rrbracket(\rho_x, \rho_c) = \{\mathsf{combine}(\overline{\sigma_c}, \sigma_c, \sigma_x|_G) \mid \langle \overline{\sigma_c}, \sigma_c \rangle \in \rho_c \wedge \langle \overline{\sigma_x}, \sigma_x \rangle \in \rho_x \wedge \sigma_c|_G = \sigma_e|_G\}$$

$$\llbracket \mathsf{combine} \rrbracket \; : \; \Sigma \times \Sigma \times \Sigma \to \mathcal{R} \text{ (assumed to be given)}$$
$$(\,\cdot\,|_G) \; : \; \Sigma \to \Sigma \text{ (assumed to be given)}$$

Function entry computes the relation ρ_e at the entry to the invoked procedure. It removes the information regarding the caller's local variables from the current states σ_c coming from the caller's relation at the call-site ρ_c using function $(\,\cdot\,|_G)$, which is assumed to be given. Note that in the computed relation, the entry state and the calling state of the callee are identical.

Function return computes relation ρ_r, which updates the caller's current state with the effect of the callee. The function computes triples $\langle \overline{\sigma_c}, \sigma_c, \sigma_x|_G \rangle$ out of relations $\langle \overline{\sigma_c}, \sigma_c \rangle$ and $\langle \overline{\sigma_x}, \sigma_x \rangle$ coming from the caller at the calls site and to the callee at the return site. return considers only relations where the global part of the caller's current state matches that of the callee's entry state. Note that at the triple, the middle state, σ_c, contains the values of the caller's local variables, which the callee cannot modify, and the last state, $\sigma_x|_G$, contains the updated state of the global parts of the memory state. Procedure combine combines these two kinds of information and generates the updated relation at the return site.

EXAMPLE 2. *For memory states $\langle s_g, s_l, h \rangle \in \Sigma$ comprised of environments s_g and s_l, giving values to global and local variables, respectively, and a heap h, $(\,\cdot\,|_G)$ and* combine *are defined as*

$$\langle s_g, s_l, h \rangle|_G = \langle s_g, \bot, h \rangle,$$
$$\llbracket \mathsf{combine} \rrbracket(\langle \overline{s_g}, \overline{s_l}, \overline{h} \rangle, \langle s_g, s_l, h \rangle, \langle s'_g, \bot, h' \rangle) = (\langle \overline{s_g}, \overline{s_l}, \overline{h} \rangle, \langle s'_g, s_l, h' \rangle).$$

4. Intraprocedural Analysis using Modularity

In this section we show how the modularity properties of lattice elements can help in the analysis of programs without procedures. (Programs with procedures are handled in Sec. 5.) The main idea is to require that only meet and join operators are used to define the abstract semantics of primitive commands and that the argument of the meet is right-modular. We begin with the connection analysis example, and then describe the general case.

4.1 Intraprocedural Connection Analysis

Abstract Domain. The abstract domain consists of equivalence relations on the variables from $\mathsf{L} \cup \mathsf{G}$ and a minimal element $\bot$. Intuitively, variables belong to different partitions if they never point to connected heap objects (i.e., those that are not connected by any chain of pointers even when the directions of these pointers are ignored). For instance, if there is a program state occurring at a program point *pt* in which $x.f$ and y denote the same heap object, then it must be that x and y belong to the same equivalence class of the analysis result at *pt*. We denote by $\mathsf{Equiv}(\Upsilon)$ the set of equivalence relations over a set Υ. Every equivalence relation on Υ induces a unique partitioning of Υ into its equivalence classes and vice versa. Thus, we use these binary-relation and partition views of an equivalence relation interchangeably throughout this paper.

DEFINITION 3. *A* **partition lattice** *over a set Υ is a 6-tuple $\mathcal{D}_{\mathsf{part}}(\Upsilon) = \langle \mathsf{Equiv}(\Upsilon), \sqsubseteq, \bot_{\mathsf{part}}, \top_{\mathsf{part}}, \sqcup, \sqcap \rangle$.*

- *For any equivalence relations d_1, d_2 in $\mathsf{Equiv}(\Upsilon)$,*
$$d_1 \sqsubseteq d_2 \Leftrightarrow \forall v_1, v_2 \in \Upsilon, v_1 \overset{d_1}{\cong} v_2 \Rightarrow v_1 \overset{d_2}{\cong} v_2,$$
where $v_1 \overset{d_i}{\cong} v_2$ means that v_1 and v_2 are related by d_i.
- *The minimal element $\bot_{\mathsf{part}} = \{\{a\} \mid a \in \Upsilon\}$ is the identity relation, relating each element only with itself.*
- *The maximum element $\top_{\mathsf{part}} = \{\Upsilon\}$ is the complete relation, relating each element to every element in Υ. It defines the partition with only one equivalence class:*
- *The join is defined by $d_1 \sqcup d_2 = (d_1 \cup d_2)^+$, where we take the binary-relation view of equivalence relations d_1 and d_2 and $-^+$ is the transitive closure operation.*
- *The meet is defined by $d_1 \sqcap d_2 = d_1 \cap d_2$. Here again we take the binary-relation view of d_i's.*

$$\begin{array}{lllll}
\llbracket x = \mathtt{null} \rrbracket^\sharp(d) & = & \llbracket y = \mathtt{new} \rrbracket^\sharp(d) & = & d \sqcap S_{\widehat{x}} \\
\llbracket x = y \rrbracket^\sharp(d) & = & \llbracket x = y.f \rrbracket^\sharp(d) & = & (d \sqcap S_{\widehat{x}}) \sqcup U_{\widehat{x}\widehat{y}} \\
\llbracket x.f = y \rrbracket^\sharp(d) & = & d \sqcup U_{\widehat{x}\widehat{y}} & &
\end{array}$$
$$\begin{array}{llll}
\text{where} & S_{\widehat{x}} & = & \{\{\widehat{x}\}\} \cup \{\{z \mid z \in \Upsilon \setminus \{\widehat{x}\}\}\} \\
 & U_{\widehat{x}\widehat{y}} & = & \{\{\widehat{x}, \widehat{y}\}\} \cup \{\{z\} \mid z \in \Upsilon \setminus \{\widehat{x}, \widehat{y}\}\}
\end{array}$$

Table 1. Abstract semantics of primitive commands in the connection analysis for $d \neq \bot$. $\llbracket a \rrbracket^\sharp(\bot) = \bot$ for any command a. $U_{\widehat{x}\widehat{y}}$ is used to merge the connection sets of $\widehat{x}$ and $\widehat{y}$. $S_{\widehat{x}}$ is used to separate $\widehat{x}$ from its current connection set. In Sec. 4.1, $\widehat{x}$ is x and $\widehat{y}$ is y. In Sec. 5, $\widehat{x}$ denotes x' and $\widehat{y}$ denotes y'.

For an element $x \in \Upsilon$, the **connection set** of x in $d \in \mathsf{Equiv}(\Upsilon)$, denoted $[x]$, is the equivalence class of x in d.

Throughout the paper, we refer to an extended partition domain $\mathcal{D} = \mathcal{D}_{\mathsf{part}} \cup \{\bot\}$, which is the result of adding a bottom element $\bot$ to the original partition lattice, where for every $d \in \mathcal{D}_{\mathsf{part}}$, $\bot \sqsubset d$.

Abstract Semantics. Table 1 shows the abstract semantics of primitive commands for the connection analysis.

Assigning `null` or a newly allocated object to a variable x separates x from its connection set. Therefore, the analysis takes the meet of the current abstract state with S_x — the partition with two connection sets $\{x\}$ and the rest of the variables.

The effect of the statement $x = y$ is to separate the variable x from its connection set and to add x to the connection set of y. This is realized by performing a meet with S_x, and then a join with U_{xy} — a partition with $\{x, y\}$ as a connection set and singleton connection sets for the rest of the variables.

The abstraction does not distinguish between the objects pointed to by y and $y.f$. Thus, following [11], we set x to be in the same connection set as y after the assignment $x = y.f$. As a result, the same abstract semantics is used for both $x = y.f$ and $x = y$.

The concrete semantics of $x.f = y$ redirects the f field of the object pointed to by x to the object pointed to by y. The abstract semantics treats this statement in a rather conservative way, performing "weak updates": We merge the connection sets of x and y by joining the current abstract state with U_{xy}.

4.2 Conditionally Compositional Intraprocedural Analysis

DEFINITION 4 (Conditionally Adaptable Functions). *Let $\mathcal{D}$ be a lattice. A function $f : \mathcal{D} \to \mathcal{D}$ is **conditionally adaptable** if it has the form $f = \lambda d.((d \sqcap d_p) \sqcup d_g)$ for some $d_p, d_g \in \mathcal{D}$ and the element d_p is right-modular. We refer to d_p as f's **meet element** and to d_g as f's **join element**.*

We focus on static analyses where the transfer function for every atomic command a is some conditionally adaptable function $\llbracket a \rrbracket^\sharp$. We denote the meet elements of $\llbracket a \rrbracket^\sharp$ by $P\llbracket a \rrbracket^\sharp$. For a command C, we denote by $P\llbracket C \rrbracket^\sharp$ the set of meet elements of primitive subcommands occurring in C.

LEMMA 5. *Let $\mathcal{D}$ be a lattice. Let C be a command which does not contain procedure calls. For every $d_1, d_2 \in \mathcal{D}$ if $d_2 \sqsubseteq d_p$ for every $d_p \in P\llbracket C \rrbracket^\sharp$, then $\llbracket C \rrbracket^\sharp(d_1 \sqcup d_2) = \llbracket C \rrbracket^\sharp(d_1) \sqcup d_2$.*

Lem. 5 can be used to justify compositional summary-based intraprocedural analyses in the following way: Take a command C and an abstract value d_2 such that the conditions of the lemma hold. Computing the abstract value $[\![C]\!]^\sharp(d_1 \sqcup d_2)$ can be done by computing $d = [\![C]\!]^\sharp(d_1)$, possibly caching (d_1, d) in a summary for C, and then adapting the result by joining d with d_2.[1]

LEMMA 6. *The transfer functions of primitive commands in the intraprocedural connection analysis are conditionally adaptable.*

In contrast, and perhaps counter-intuitively, our framework for the interprocedural analysis has non-conditional summaries, which do not have a proviso like $d_2 \sqsubseteq P[\![C]\!]^\sharp$. It achieves this by requiring certain properties of the abstract domain used to record procedures summaries, which we now describe.

5. Compositional Analysis using Modularity

In this section, we define an abstract framework for compositional interprocedural analysis using modularity and illustrate the framework using the connection analysis. To make the material more accessible, we formulate some of the definitions specifically for the connection analysis and defer the general definitions to [4].

The main message is that the meet elements of atomic commands are right-modular and greater than or equal to all the elements in a sublattice of the domain which is used to record the effect of the caller on the callee's entry state. This allows to summarize the effects of procedures in a bottom-up manner, and to get the coincidence between the results of the bottom-up and top-down analyses.

5.1 Partition Domains for Ternary Relations

We first generalize the abstract domain for the *intra*procedural connection analysis described in Sec. 4.1 to the *inter*procedural setting.

Recall that the return operation defined in Sec. 3 operates on triplets of states. For this reason, we use an abstract domain that allows representing ternary relations between program states. We now formulate this for the connection analysis. For every global variable $g \in \mathsf{G}$, $\overline{g}$ denotes the value of g at the entry to a procedure and g' denotes its current value. The analysis computes at every program point a relation between the objects pointed to by global variables at the entry to the procedure (represented by $\overline{\mathsf{G}}$) and the ones pointed to by global variables and local variables at the current state (represented by G' and L', respectively).

For technical reasons, described later, we also use the set $\dot{\mathsf{G}}$ to compute the effect of procedure calls. These sets are used to represent partitions over variables in the same way as in Sec. 4.1. Formally, we define $\mathcal{D} = \mathsf{Equiv}(\Upsilon) \cup \{\perp\}$ in the same way as in Def. 3 of Sec. 4.1 where $\Upsilon = \overline{\mathsf{G}} \cup \mathsf{G}' \cup \dot{\mathsf{G}} \cup \mathsf{L}'$ and

$$\mathsf{G}' = \{g' \mid g \in \mathsf{G}\} \qquad \overline{\mathsf{G}} = \{\bar{g} \mid g \in \mathsf{G}\}$$
$$\dot{\mathsf{G}} = \{\dot{g} \mid g \in \mathsf{G}\} \qquad \mathsf{L}' = \{x' \mid x \in \mathsf{L}\}$$

[1] Interestingly, the notion of condensation in [12] is similar to the implications of Lem. 5 (and to the frame rule in separation logic) in the sense that the join (or $*$ in separation logic) distributes over the transfer functions. However, [12] requires the distribution $S(a + b) = a + S(b)$ hold for every two elements a and b in the domain. Our requirements are less restrictive: In Lem. 5, we require such equality only for elements smaller than or equal to the meet elements of the transfer functions. This is important for handling the connection analysis in which condensation property does not hold. (In addition, the method of [12] is developed for domains for logical programs using completion and requires the refined domain to be compatible with the projection operator, which is specific to logic programs. and be finitely generated [12, Cor. 4.9].)

$$\begin{array}{rcl}
R_X &=& \{\{x \mid x \in X\}\} \cup \{\{x\} \mid x \in \Upsilon \setminus X\} \\
\mathcal{D}_{\mathsf{in}} &=& \{d \in \mathcal{D} \mid d \sqsubseteq R_{\overline{\mathsf{G}}}\} \\
\mathcal{D}_{\mathsf{out}} &=& \{d \in \mathcal{D} \mid d \sqsubseteq R_{\mathsf{G}'}\} \\
\mathcal{D}_{\mathsf{inout}} &=& \{d \in \mathcal{D} \mid d \sqsubseteq R_{\overline{\mathsf{G}} \cup \mathsf{G}'}\} \\
\mathcal{D}_{\mathsf{inoutloc}} &=& \{d \in \mathcal{D} \mid d \sqsubseteq R_{\overline{\mathsf{G}} \cup \mathsf{G}' \cup \mathsf{L}'}\}
\end{array}$$

Table 2. Constant projection element R_X for an arbitrary set X and the sublattices of $\mathcal{D}$ used by the interprocedural relational analysis. R_X is the partition that contains a connection set for all the variables in X and singletons for all the variables in $\Upsilon \setminus X$. Each one of the sublattices represents connection relations in the current state between objects which were pointed to by local or global variables at different stages during the execution.

REMARK 1. *Formally, the interprocedural connection analysis computes an over-approximation of the relational concrete semantics defined in Sec. 3. A Galois connection between the $\mathcal{D}$ and a standard (concrete collecting relational) domain for heap manipulating programs is defined in [4].*

5.2 Triad Partition Domains

We first informally introduce the concept of *Triad Domain*. Triad domains are used to perform abstract interpretation to represent concrete domains and their concrete semantics as defined in Sec. 3. A triad domain $\mathcal{D}$ is a complete lattice which conservatively represents binary and ternary relations (hence the name "triad") between memory states arising at different program points such as the entry point to the caller procedure, the call-site, and the current program point. The analysis uses elements $d \in \mathcal{D}$ to represent ternary relations when computing procedure returns. For all other purposes binary relations are used. More specifically, the analysis makes special use of the *triad sublattices* of $\mathcal{D}$ defined in Table 2, which we now explain.

Each sublattice is used to abstract binary relations between sets of program states arising at different program points. We construct these sublattices by first choosing *projection elements* d_{proj_i} from the abstract domain $\mathcal{D}$, and then defining the sublattice $\mathcal{D}_i$ to be the closed interval $[\perp, d_{\mathsf{proj}_i}]$, which consists of all the elements between $\perp$ and d_{proj_i} according to the $\sqsubseteq$ order (including $\perp$ and d_{proj_i}). Moreover, for every $i \in \{\mathsf{in}, \mathsf{out}, \mathsf{inout}, \mathsf{inoutloc}\}$, we define the projection operation $(\,\cdot\,|_i)$ as follows: $d|_i = d \sqcap d_{\mathsf{proj}_i}$. Note that $d|_i$ is always in $\mathcal{D}_i$.

In the connection analysis, projection elements d_{proj_i} are defined in terms of R_X's in Table 2:

$$d_{\mathsf{proj}_{\mathsf{in}}} = R_{\overline{\mathsf{G}}}, d_{\mathsf{proj}_{\mathsf{out}}} = R_{\mathsf{G}'}, d_{\mathsf{proj}_{\mathsf{inout}}} = R_{\overline{\mathsf{G}} \cup \mathsf{G}'}, d_{\mathsf{proj}_{\mathsf{inoutloc}}} = R_{\overline{\mathsf{G}} \cup \mathsf{G}' \cup \mathsf{L}'}.$$

R_X is the partition that contains a connection set containing all the variables in X and singleton sets for all the variables in $\Upsilon \setminus X$.

Each abstract state in the sublattice $\mathcal{D}_{\mathsf{out}}$ represents a partition on heap objects pointed to by global variables in the current state, such that two such heap objects are grouped together in this partition when they are *weakly connected*, i.e., we can reach from one object to the other by following pointers forward or backward. For example, suppose that a global variable g_1 points to an object o_1 and a global variable g_2 points to an object o_2 at a program point pt, and that o_1 and o_2 are weakly connected. Then, the analysis result will be an equivalence relation that puts g_1' and g_2' in the same equivalence class.

Each abstract state in $\mathcal{D}_{\mathsf{in}}$ represents a partition of objects pointed to by global variables upon the procedure entry where the partition is done according to weakly-connected components.

The sublattice $\mathcal{D}_{\mathsf{inout}}$ is used to abstract relations in the current heap between objects pointed to by global variables upon procedure entry and those pointed to by global variables in the current program point. For example, if at point pt in a procedure p an ob-

$$
\begin{aligned}
\iota_{\mathsf{entry}} &= \bigsqcup_{g\in\mathsf{G}} U_{g'\overline{g}} = \{\{g',\overline{g}\},\{\dot{g}\} \mid g\in\mathsf{G}\} \\
[\![\mathsf{entry}]\!]^{\sharp}(d) &= (d \sqcap R_{\mathsf{G}'}) \sqcup \iota_{\mathsf{entry}} \\
[\![\mathsf{return}]\!]^{\sharp}(d_{\mathsf{exit}}, d_{\mathsf{call}}) &= (f_{\mathsf{call}}(d_{\mathsf{call}}) \sqcup f_{\mathsf{exit}}(d_{\mathsf{exit}} \sqcap R_{\overline{\mathsf{G}}\cup\mathsf{G}'})) \sqcap R_{\overline{\mathsf{G}}\cup\mathsf{G}'\cup\mathsf{L}} \\
[\![p()]\!]^{\sharp}(d) &= [\![\mathsf{return}]\!]^{\sharp}(([\![C_{\mathsf{body}_p}]\!]^{\sharp} \circ [\![\mathsf{entry}]\!]^{\sharp})(d), d) \\
[\![p()]\!]^{\sharp}_{\mathsf{BU}}(d) &= [\![\mathsf{return}]\!]^{\sharp}([\![C_{\mathsf{body}_p}]\!]^{\sharp}(\iota_{\mathsf{entry}}), d)
\end{aligned}
$$

Table 3. The definition of ι_{entry} and the interprocedural abstract semantics for the top-down and bottom-up connection analyses. ι_{entry} is the element that represents the identity relation between input and output, and C_{body_p} is the body of procedure p.

ject is currently pointed to by a global variable g_1 and it belongs to the same weakly connected component as an object that was pointed to by a global variable g_2 at the entry point of p, then the partition at pt will include a connection set with $\overline{g_1}$ and g_2'.

Similarly, the sublattice $\mathcal{D}_{\mathsf{inoutloc}}$ is used to abstract relations in the current heap between objects pointed to by global variables upon procedure entry and global and local variables in the current program point.

5.3 Interprocedural Top Down Triad Connection Analysis

We describe here the abstract semantics for the top-down interprocedural connection analysis. The intraprocedural semantics is shown in Table 1. Notice that there is a minor difference between the semantics of primitive commands for the intraprocedural connection analysis defined in Sec. 4.1 and for the analysis in this section. In the analysis without procedures we use x, whereas in the analysis of this section we use x'.

The abstract meaning of procedure calls in the connection analysis is defined in Table 3. Again, we refer to the auxiliary constant elements R_X for a set X defined in Table 2.

When a procedure is entered, local variables of the procedure and all the global variables $\overline{g}$ at the entry to the procedure are initialized to `null`. This is realized by applying the meet operation with auxiliary variable $R_{\mathsf{G}'}$. Then, each of the $\overline{g}$ is initialized with the current variable value g' using ι_{entry}. The ι_{entry} element denotes a particular state that abstracts the identity relation between input and output states. In the connection analysis, it is defined by a partition containing $\{\overline{g}, g'\}$ connection sets for all global variables g. Intuitively, this stores the current value of variable g into $\overline{g}$, by representing the case where the object currently pointed to by g is in the same weakly connected component as the object that was pointed to by g at the entry point of the procedure.

The effect of returning from a procedure is more complex. It takes two inputs: d_{call}, which represents the partition at the call-site, and d_{exit}, which represents the partition at the exit from the procedure. The meet operation of d_{exit} with $R_{\overline{\mathsf{G}}\cup\mathsf{G}'}$ emulates the nullification of local variables of the procedure. The computed abstract values emulate the composition of the input-output relation of the call-site with that of the return-site. Variables of the form $\dot{g}$ are used to implement a natural join operation for composing these relations. $f_{\mathsf{call}}(d_{\mathsf{call}})$ renames global variables from g' to $\dot{g}$ and $f_{\mathsf{exit}}(d_{\mathsf{exit}})$ renames global variables from $\overline{g}$ to $\dot{g}$ to allow natural join. Intuitively, the old values $\overline{g}$ of the callee at the exit-site are matched with the current values g' of the caller at the call-site. The last meet operation represents the nullification of the temporary values $\dot{g}$ of the global variables.

In [4] we generalize these definitions to generic *triad analyses*.

5.4 Bottom Up Triad Connection Analysis

In this section, we introduce a bottom-up semantics for the connection analysis. Primitive commands are interpreted in the same way as in the top-down analysis. The effect of procedure calls is computed using the function $[\![p()]\!]^{\sharp}_{\mathsf{BU}}(d)$, defined in Table 3, instead of $[\![p()]\!]^{\sharp}(d)$. The two functions differ in the first argument they use when applying $[\![\mathsf{return}]\!]^{\sharp}$: $[\![p()]\!]^{\sharp}_{\mathsf{BU}}(d)$ uses a *constant* value, which is the abstract state at the procedure exit computed when analyzing $p()$ with ι_{entry}. In contrast, $[\![p()]\!]^{\sharp}(d)$ uses the abstract state resulting at the procedure exit when analyzing the call to $p()$ with d.

5.5 Coincidence Result in Connection Analysis

We are interested in finding a sufficient condition on an analysis, for the following equality to hold:

$$\forall d \in \mathcal{D}.\ [\![p()]\!]^{\sharp}_{\mathsf{BU}}(d) = [\![p()]\!]^{\sharp}(d)\,.$$

We sketch the main arguments of the proof, substantiating their validity using examples from the interprocedural connection analysis in lieu of more formal mathematical arguments, given in [4].

5.5.1 Uniform Representation of Entry Abstract States

Any abstract state d arising at the entry to a procedure in the top-down analysis is **uniform**, i.e., it is a partition such that for every global variable g, variables $\overline{g}$ and g' are always in the same connection set. This is a result of the definition of function `entry`, which projects the abstract element at the call-site into the sublattice $\mathcal{D}_{\mathsf{out}}$ and the successive join with the ι_{entry} element. The projection results in an abstract state where all connection sets containing more than a single element are comprised only of primed variables. Then, after joining $d|_{\mathsf{out}}$ with ι_{entry}, each old variable $\overline{g}$ resides in the same partition as its corresponding current primed variable g'.

We point out that the uniformity of the entry states is due to the property of ι_{entry} that its connection sets are comprised of pairs of variables of the form $\{x', \overline{x}\}$. One important implication of this uniformity is that every entry abstract state d_0 to any procedure has a dual representation. In one representation, d is the join of ι_{entry} with some elements $U_{x'y'} \in \mathcal{D}_{\mathsf{out}}$. In the other representation, d is expressed as the join of ι_{entry} with some elements $U_{\overline{x}\overline{y}} \in \mathcal{D}_{\mathsf{in}}$. In the following, we use the function o that replaces relationships among current variables by those among old ones: $o(U_{x'y'}) = U_{\overline{x}\overline{y}}$; and $o(d)$ is the least upper bounds of ι_{entry} and elements $U_{\overline{x}\overline{y}}$ for all x, y such that x' and y' are in the same connection set of d.

5.5.2 Delayed Evaluation of the Effect of Calling Contexts

Elements of the form $U_{\overline{x}\overline{y}}$, coming from $\mathcal{D}_{\mathsf{in}}$, are smaller than or equal to the meet elements of intraprocedural statements. In Lem. 6 of Sec. 4 we proved that the semantics of the connection analysis is conditionally adaptable. Thus, computing the composed effect of any sequence τ of intraprocedural transformers on an entry state of the form $d_0 \sqcup U_{\overline{x}_1\overline{y}_1} \ldots \sqcup U_{\overline{x}_n\overline{y}_n}$ results in an element of the form $d_0' \sqcup U_{\overline{x}_1\overline{y}_1} \ldots \sqcup U_{\overline{x}_n\overline{y}_n}$, where d_0' results from applying the transformers in τ on d_0. Using the observation we made in Sec. 5.5.1, this means that we can represent any abstract element d resulting at a call-site as $d = d_1 \sqcup d_2$, where d_1 is the effect of τ on ι_{entry} and $d_2 \in \mathcal{D}_{\mathsf{in}}$ is a join of elements of the form $U_{\overline{x}\overline{y}} \in \mathcal{D}_{\mathsf{in}}$:

$$d = d_1 \sqcup U_{\overline{x}_1\overline{y}_1} \ldots \sqcup U_{\overline{x}_n\overline{y}_n}\,. \tag{1}$$

5.5.3 Counterpart Representation for Calling Contexts

Because of the previous reasoning, we can now assume that any abstract value at the call-site to a procedure $p()$ is of the form $d_1 \sqcup d_3$, where $d_3 \in \mathcal{D}_{\mathsf{in}}$ and it is a join of elements of form $U_{\overline{x}\overline{y}}$.

For each $U_{\overline{x}\overline{y}}$, the entry state resulting from analyzing $p()$ when the calling context is $d_1 \sqcup U_{\overline{x}\overline{y}}$ is either identical to the one resulting from d_1 or can be obtained from d_1 by merging two of its connection sets. Furthermore, the need to merge occurs only if there are variables w' and z' such that w' and $\overline{x}$ are in one of the connection sets of d_1 and z' and $\overline{y}$ are in another. This means that the effect of

$U_{\overline{xy}}$ on the entry state can be expressed via primed variables:

$$d_1 \sqcup U_{\overline{xy}} = d_1 \sqcup U_{w'z'} \,.$$

This implies that if the abstract state at the call-site is $d_1 \sqcup d_3$, then there is an element $d'_3 \in \mathcal{D}_{\mathsf{out}}$ such that

$$(d_1 \sqcup d_3)|_{\mathsf{out}} = d_1|_{\mathsf{out}} \sqcup d'_3 \tag{2}$$

We refer to the element $d'_3 \in \mathcal{D}_{\mathsf{out}}$, which can be used to represent the effect of $d_3 \in \mathcal{D}_{\mathsf{in}}$ at the call-site as d_3's *counterpart*, and denote it by $\widehat{d_3}$.

5.5.4 Representing Entry States with Counterparts

The above facts imply that we can represent an abstract state d at the call-site as

$$d = d_1 \sqcup d_3 \sqcup d_4, \tag{3}$$

where $d_3, d_4 \in \mathcal{D}_{\mathsf{in}}$. d_3 is a join of the elements of the form $U_{\overline{xy}}$ such that $\overline{x}$ and $\overline{y}$ reside in d_1 in different partitions, which also contain current (primed) variables, and thus possibly affect the entry state; d_4 is a join of all the other elements $U_{\overline{xy}} \in \mathcal{D}_{\mathsf{in}}$, which are needed to represent d in this form, but either $\bar{x}$ or $\overline{y}$ resides in the same partition in d_1 or one of them is in a partition containing only old variables. As explained in the previous paragraph, there is an element $d'_3 = \widehat{d_3}$ that joins elements of the form $U_{x'y'}$ such that

$$(d_1 \sqcup d_3)|_{\mathsf{out}} = (d_1 \sqcup d'_3)|_{\mathsf{out}} \tag{4}$$

and

$$d = d_1 \sqcup d_3 \sqcup d_4 = d_1 \sqcup d'_3 \sqcup d_4 \,. \tag{5}$$

Thus, after applying the entry's semantics, we get that abstract states at the entry point of procedure are always of the form

$$[\![\mathsf{entry}]\!]^\sharp(d) = (d_1 \sqcup d'_3)|_{\mathsf{out}} \sqcup \iota_{\mathsf{entry}} \tag{6}$$

where d'_3 represents the effect of $d_3 \sqcup d_4$ on partitions containing current variables g' in d_1. Because $U_{x'y'} \sqsubseteq R_{G'}$ and d'_3 joins elements of form $U_{x'y'}$, the *modularity of the lattice* gives that

$$(d_1 \sqcup d'_3)|_{\mathsf{out}} \sqcup \iota_{\mathsf{entry}} = (d_1|_{\mathsf{out}} \sqcup d'_3) \sqcup \iota_{\mathsf{entry}}$$

This implies that every state d_0 at an entry point to a procedure is of the following form:

$$d_0 = \iota_{\mathsf{entry}} \sqcup \underbrace{(U_{x'y'} \ldots \sqcup U_{x'_l y'_l})}_{d_1|_{\mathsf{out}}} \sqcup \underbrace{(U_{x'_{l+1} y'_{l+1}} \ldots \sqcup U_{x'_n y'_n})}_{d'_3} \,.$$

Using the dual representation of entry state, we get that

$$\iota_{\mathsf{entry}} \sqcup U_{x'_1 y'_1} \ldots \sqcup U_{x'_n y'_n} = \iota_{\mathsf{entry}} \sqcup o(U_{x'_1 y'_1} \ldots \sqcup U_{x'_n y'_n})$$

and thus the form of a state d_0 at an entry point to a procedure is

$$d_0 = \iota_{\mathsf{entry}} \sqcup U_{\overline{x}_1 \overline{y}_1} \sqcup \ldots \sqcup U_{\overline{x}_n \overline{y}_n} \tag{7}$$

5.5.5 Putting It All Together

We now show that the interprocedural connection analysis can be done compositionally. Intuitively, the effect of the caller's calling context can be carried over procedure invocations. Alternatively, the effect of the callee on the caller's context can be adapted unconditionally for different caller's calling contexts.

We sketch here an outline of the proof for case $C = p()$ using the connection analysis domain. The proof goes by induction on the structure of the program. In Eq.3 we showed that every abstract value that arises at the call-site is of the form $d_1 \sqcup d_3 \sqcup d_4$, where $d_3, d_4 \in \mathcal{D}_{\mathsf{in}}$. Thus, we show that

$$[\![C]\!]^\sharp(d_1 \sqcup d_3 \sqcup d_4) = [\![C]\!]^\sharp(d_1) \sqcup d_3 \sqcup d_4 \,. \tag{8}$$

Say we want to compute the effect of invoking $p()$ on abstract state d according to the top-down abstract semantics.

$$[\![p()]\!]^\sharp(d) \quad = \quad [\![\mathsf{return}]\!]^\sharp\big(\big(([\![C_{\mathsf{body}_p}]\!]^\sharp \circ [\![\mathsf{entry}]\!]^\sharp)(d)\big), d\big)$$

First, let's compute the first argument to $[\![\mathsf{return}]\!]^\sharp$.

$$\begin{array}{ll} & ([\![C_{\mathsf{body}_p}]\!]^\sharp \circ [\![\mathsf{entry}]\!]^\sharp)(d) \\ = & [\![C_{\mathsf{body}_p}]\!]^\sharp([\![\mathsf{entry}]\!]^\sharp(d_1 \sqcup d_3 \sqcup d_4)) \\ = & [\![C_{\mathsf{body}_p}]\!]^\sharp(((d_1 \sqcup d_3 \sqcup d_4)|_{\mathsf{out}}) \sqcup \iota_{\mathsf{entry}}) \\ = & [\![C_{\mathsf{body}_p}]\!]^\sharp(((d_1 \sqcup d_3)|_{\mathsf{out}}) \sqcup \iota_{\mathsf{entry}}) \\ = & [\![C_{\mathsf{body}_p}]\!]^\sharp((d_1)|_{\mathsf{out}} \sqcup d'_3 \sqcup \iota_{\mathsf{entry}}) \\ = & [\![C_{\mathsf{body}_p}]\!]^\sharp((d_1)|_{\mathsf{out}} \sqcup o(d'_3) \sqcup \iota_{\mathsf{entry}}) \\ = & [\![C_{\mathsf{body}_p}]\!]^\sharp((d_1)|_{\mathsf{out}} \sqcup \iota_{\mathsf{entry}}) \sqcup o(d'_3) \end{array} \tag{9}$$

The first equalities are mere substitutions based on observations we made before. The last one comes from the induction assumption.

When applying the $\mathtt{return}$ semantics, we first compute the natural join and then remove the temporary variables. Hence, we get

$$(f_{\mathsf{call}}(d_1 \sqcup d_3 \sqcup d_4) \sqcup f_{\mathsf{exit}}([\![C_{\mathsf{body}_p}]\!]^\sharp((d_1)|_{\mathsf{out}} \sqcup \iota_{\mathsf{entry}}) \sqcup o(d'_3)))|_{\mathsf{inoutloc}}$$

Let's first compute the result of the inner parentheses.

$$\begin{array}{ll} & f_{\mathsf{call}}(d_1 \sqcup d_3 \sqcup d_4) \sqcup f_{\mathsf{exit}}([\![C_{\mathsf{body}_p}]\!]^\sharp((d_1)|_{\mathsf{out}} \sqcup \iota_{\mathsf{entry}}) \sqcup o(d'_3)) \\ = & f_{\mathsf{call}}(d_1 \sqcup d'_3 \sqcup d_4) \sqcup f_{\mathsf{exit}}([\![C_{\mathsf{body}_p}]\!]^\sharp((d_1)|_{\mathsf{out}} \sqcup \iota_{\mathsf{entry}}) \sqcup o(d'_3)) \\ = & f_{\mathsf{call}}(d'_3) \sqcup f_{\mathsf{call}}(d_1 \sqcup d_4) \sqcup \\ & f_{\mathsf{exit}}(o(d'_3)) \sqcup f_{\mathsf{exit}}([\![C_{\mathsf{body}_p}]\!]^\sharp((d_1)|_{\mathsf{out}} \sqcup \iota_{\mathsf{entry}})) \end{array} \tag{10}$$

The first equality is by the definition of d'_3 and the last equality is by the isomorphism of the renaming operations f_{call} and f_{exit}.

Note, among the join arguments, $f_{\mathsf{exit}}(o(d'_3))$ and $f_{\mathsf{call}}(d'_3)$. Let's look at the first element. $o(d'_3)$ replaces all the occurrences of $U_{x'y'}$ in d'_3 with $U_{\overline{xy}}$. f_{exit} replaces all the occurrences of $U_{\overline{xy}}$ in $o(d'_3)$ with $U_{\dot{x}\dot{y}}$. Thus, the first element is

$$U_{\dot{x}_1 \dot{y}_1} \sqcup \ldots \sqcup U_{\dot{x}_n \dot{y}_n}$$

which is the result of replacing in d'_3 all the occurrences of $U_{x'y'}$ with $U_{\dot{x}\dot{y}}$. Consider the second element. f_{exit} replaces all occurrences of $U_{x'y'}$ in d'_3 with $U_{\dot{x}\dot{y}}$. Thus, also the second element is

$$U_{\dot{x}_1 \dot{y}_1} \sqcup \ldots \sqcup U_{\dot{x}_n \dot{y}_n}$$

Thus, we get that

$$(10) = f_{\mathsf{call}}(d'_3) \sqcup f_{\mathsf{call}}(d_1 \sqcup d_4) \sqcup f_{\mathsf{exit}}([\![C_{\mathsf{body}_p}]\!]^\sharp((d_1)|_{\mathsf{out}} \sqcup \iota_{\mathsf{entry}}))$$

Moreover, f_{call} is isomorphic and by Eq.5

$$= f_{\mathsf{call}}(d_3 \sqcup d_1 \sqcup d_4) \sqcup f_{\mathsf{exit}}([\![C_{\mathsf{body}_p}]\!]^\sharp((d_1)|_{\mathsf{out}} \sqcup \iota_{\mathsf{entry}}))$$

Remember (Eq.1) that d_3 and d_4 are both of form

$$U_{\overline{x}_1 \overline{y}_1} \sqcup \ldots \sqcup U_{\overline{x}_n \overline{y}_n}$$

and that $f_{\mathsf{call}}(d)$ only replaces g' occurrences in d; thus

$$f_{\mathsf{call}}(U_{\overline{x}_1 \overline{y}_1} \sqcup \ldots \sqcup U_{\overline{x}_n \overline{y}_n}) = U_{\overline{x}_1 \overline{y}_1} \sqcup \ldots \sqcup U_{\overline{x}_n \overline{y}_n}$$

Finally, we get

$$\begin{array}{l} = f_{\mathsf{call}}(d_1) \sqcup f_{\mathsf{exit}}([\![C_{\mathsf{body}_p}]\!]^\sharp((d_1)|_{\mathsf{out}} \sqcup \iota_{\mathsf{entry}}) \sqcup (d_3 \sqcup d_4) \\ = [\![p()]\!]^\sharp(d_1) \sqcup d_3 \sqcup d_4 \end{array}$$

5.5.6 Precision Coincidence

We combine the observations we made to informally show the coincidence result between the top-down and the bottom-up semantics. According to Eq.3, every state d at a call-site can be represented as

$d = d_1 \sqcup d_3 \sqcup d_4$, where $d_3, d_4 \in \mathcal{D}_{\mathsf{in}}$

$$\begin{aligned} [\![p()]\!]^\sharp(d) &= [\![\mathsf{return}]\!]^\sharp([\![C_{\mathsf{body}_p}]\!]^\sharp([\![\mathsf{entry}]\!]^\sharp(d)), d) \\ &= [\![\mathsf{return}]\!]^\sharp([\![C_{\mathsf{body}_p}]\!]^\sharp(d_1|_{\mathsf{out}} \sqcup \iota_{\mathsf{entry}} \sqcup d_3'), d) \\ &= [\![\mathsf{return}]\!]^\sharp([\![C_{\mathsf{body}_p}]\!]^\sharp(\iota_{\mathsf{entry}} \sqcup o(d_1|_{\mathsf{out}}) \sqcup o(d_3')), d) \quad (11) \end{aligned}$$

The second equivalence is by Eq.6, and the second equivalence is because $d_1|_{\mathsf{out}}, d_3' \in \mathcal{D}_{\mathsf{out}}$ and

$$\iota_{\mathsf{entry}} \sqcup o(d_1|_{\mathsf{out}}) \sqcup o(d_3') = \iota_{\mathsf{entry}} \sqcup d_1|_{\mathsf{out}} \sqcup d_3'$$

We showed that for every $d = d_1 \sqcup d_3 \sqcup d_4$, such that $d_3, d_4 \in \mathcal{D}_{\mathsf{in}}$,

$$[\![C]\!]^\sharp(d_1 \sqcup d_3 \sqcup d_4) = [\![C]\!]^\sharp(d_1) \sqcup d_3 \sqcup d_4$$

for any command C. Therefore, since $o(d_3'), o(d_1|_{\mathsf{out}}) \in \mathcal{D}_{\mathsf{in}}$,

$$(11) = [\![\mathsf{return}]\!]^\sharp([\![C_{\mathsf{body}_p}]\!]^\sharp(\iota_{\mathsf{entry}}) \sqcup o(d_1|_{\mathsf{out}}) \sqcup o(d_3'), d_1 \sqcup d_3 \sqcup d_4)\,.$$

By Eq.5, $d_1 \sqcup d_3 \sqcup d_4 = d_1 \sqcup d_3' \sqcup d_4$ Thus, we can remove $o(d_3')$ because $f_{\mathsf{exit}}(o(d_3'))$ will be redundant in the natural join of the $[\![\mathsf{return}]\!]^\sharp$ operator. Using a similar reasoning, we can remove $f_{\mathsf{exit}}(o(d_1|_{\mathsf{out}}))$, since $f_{\mathsf{call}}(d_1|_{\mathsf{out}}) \sqsubseteq f_{\mathsf{call}}(d_1)$. Hence, finally,

$$(11) = [\![\mathsf{return}]\!]^\sharp([\![C_{\mathsf{body}_p}]\!]^\sharp(\iota_{\mathsf{entry}}), d_1 \sqcup d_3 \sqcup d_4) = [\![p()]\!]^\sharp{}_{\mathsf{BU}}(d)\,.$$

6. Experimental evaluation

In this section, we evaluate the effectiveness of our approach in practice using the connection analysis for concreteness. We implemented three versions of this analysis: the original top-down version from [11], our modified top-down version, and our modular bottom-up version that coincides in precision with the modified top-down version. We next briefly describe these three versions.

The original top-down connection analysis does not meet the requirements described in Sec. 5, because the abstract transformer for destructive update statements $x.f = y$ depends on the abstract state; the connection sets of x and y are not merged if x or y points to null in all the executions leading to this statement. We therefore conservatively modified the analysis to satisfy our requirements, by changing the abstract transformer to always merge x's and y's connection sets. Our bottom-up modular analysis that coincides with this modified top-down analysis operates in two phases. The first phase computes a summary for every procedure by analyzing it with an input state ι_{entry}. The summary over-approximates relations between all possible inputs of this procedure and each program point in the body of the procedure. The second phase is a chaotic iteration algorithm which propagates values from callers to callees using the precomputed summaries, and is similar to the second phase of the interprocedural functional algorithm of [18, Figure 7].

We implemented all three versions of connection analysis described above using Chord [16] and applied them to five Java benchmark programs whose characteristics are summarized in Table 4. They include two programs (grande2 and grande3) from the Java Grande benchmark suite and two (antlr and bloat) from the DaCapo benchmark suite. We excluded programs from these suites that use multi-threading, since our analyses are sequential. Our larger three benchmark programs are commonly used in evaluating pointer analyses. All our experiments were performed using Oracle HotSpot JRE 1.6.0 on a Linux machine with Intel Xeon 2.13 GHz processors and 128 Gb RAM.

We next compare the top-down and bottom-up approaches in terms of precision (Sec. 6.1) and scalability (Sec. 6.2). We omit the modified top-down version of connection analysis from further evaluation, as we found its performance difference from the original top-down version to be negligible, and since its precision is identical to our bottom-up version (in principle, due to our coincidence result, as well as confirmed in our experiments).

6.1 Precision

We measure the precision of connection analysis by the size of the connection sets of pointer variables at program points of interest. Each such pair of variable and program point can be viewed as a separate *query* to the connection analysis. To obtain such queries, we chose the parallelism client proposed in the original work on connection analysis [11], which demands the connection set of each dereferenced pointer variable in the program. In Java, this corresponds to variables of reference type that are dereferenced to access instance fields or array elements. More specifically, our queries constitute the base variable in each occurrence of a getfield, putfield, aload, or astore bytecode instruction in the program. The number of such queries for our five benchmarks are shown in the "# of queries" column of Table 5. To avoid counting the same set of queries across benchmarks, we only consider queries in application code, ignoring those in JDK library code. This number of queries ranges from around 0.6K to over 10K for our benchmarks.

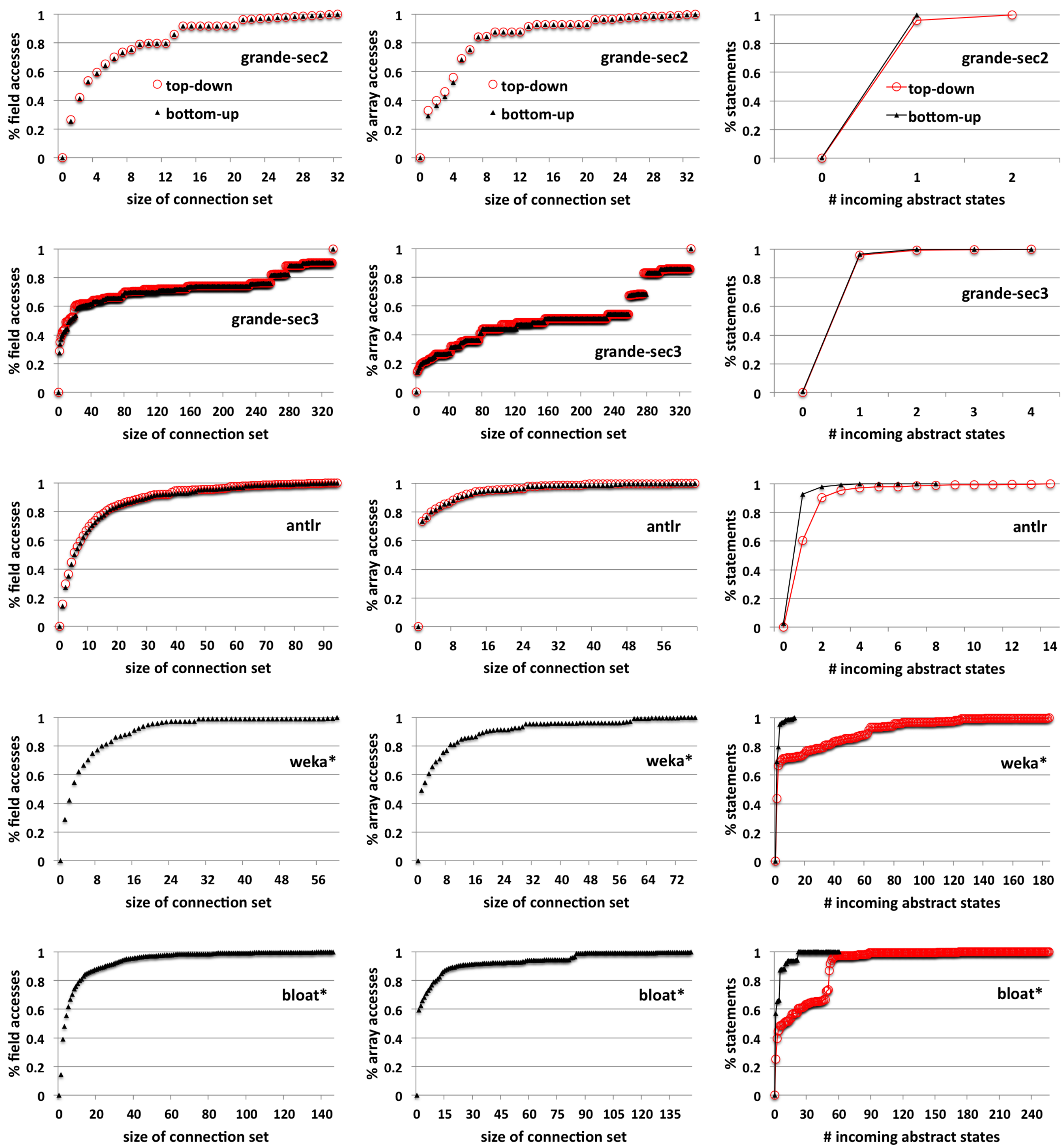
A precise answer to a query $x.f$ (a field access) or *x[i]* (an array access) is one that is able to disambiguate the object pointed to by x from objects pointed to by another variable y. In the connection analysis abstraction, x and y are disambiguated if they are not connected. We thereby measure the precision of connection analysis in terms of the size of the connection set of variable x, where a more precise abstraction is one where the number of other variables connected to x is small. To avoid gratuitously inflating this size, we perform intra-procedural copy propagation on the intermediate representation of the benchmarks in Chord.

Fig. 4 provides a detailed comparison of precision, based on the above metric, of the top-down and bottom-up versions of connection analysis, separately for field access queries (column (a)) and array access queries (column (b)). Each graph in columns (a) and (b) plots, for each distinct connection set size (on the X axis), the fraction of queries (on the Y axis) for which each analysis computed connection sets of equal or smaller size. Graphs marked (*) indicate where the sizes of connection sets computed by the top-down analysis are not plotted because the analysis timed out after six hours. This happens for our two largest benchmarks (weka and bloat). The graphs for the remaining three benchmarks (grande2, grande3, and antlr) show that the precision of our modular bottom-up analysis closely tracks that of the original top-down analysis: the points for the bottom-up and top-down analyses, denoted ▲ and ○, respectively, overlap almost perfectly in each of the six graphs. The ratio of the connection set size computed by the top-down analysis to that computed by the bottom-up analysis on average across all queries is 0.977 for grande2, 0.977 for grande3, and 0.952 for antlr. While we do not measure the impact of this precision loss of 2-5% on a real client, we note that for our largest two benchmarks, the top-down analysis does not produce any useful result.

6.2 Scalability

Table 5 compares the scalability of the top-down and bottom-up analyses in terms of three different metrics: running time, memory consumption, and the total number of computed abstract states. As noted earlier, the bottom-up analysis runs in two phases: a summary computation phase followed by a summary instantiation phase. The above data for these phases is reported in separate columns of the table. On our largest benchmark (bloat), the bottom-up analysis takes around 50 minutes and 873 Mb memory, whereas the top-down analysis times out after six hours, not only on this benchmark but also on the second largest one (weka).

The "# of abstract states" columns provide the sum of the sizes of the computed abstractions in terms of the number of abstract states, including only incoming states at program points of queries (in the "queries" sub-column), and incoming states at all program points, including the JDK library (in the "total" sub-column). Col-



(a) Precision comparison for field accesses. (b) Precision comparison for array accesses. (c) Scalability comparison.

Figure 4. Comparison of the precision and scalability of the original top-down and our modular bottom-up versions of connection analysis. Each graph in columns (a) and (b) shows, for each distinct connection set size (on the X axis), the fraction of queries (on the Y axis) for which the analyses computed connection sets of equal or smaller size. This data is missing for the top-down analysis in the graphs marked (*) because this analysis timed out after six hours on those benchmarks. For the remaining benchmarks, the near perfect overlap in the points plotted for the two analyses indicates very minor loss in precision of the bottom-up analysis over the top-down analysis. Column (c) compares scalability of the two analyses in terms of the total number of abstract states computed by them. Each graph in this column shows, for each distinct number of incoming abstract states computed at each program point (on the X axis), the fraction of program points (on the Y axis) with equal or smaller number of such states. The numbers for the top-down analysis in the graphs marked (*) were obtained at the instant of timeout. These graphs clearly show the blow-up in the number of states computed by the top-down analysis over the bottom-up analysis.

	description	# classes		# methods		# bytecodes	
		app only	total	app only	total	app only	total
grande2	Java Grande kernels	17	61	112	237	8,146	13,724
grande3	Java Grande large-scale applications	42	241	231	1,162	27,812	75,139
antlr	Parser and translator generator	116	358	1,167	2,400	128,684	186,377
weka	Machine-learning library for data-mining tasks	62	530	575	3,391	40,767	223,291
bloat	Java bytecode optimization and analysis tool	277	611	2,651	4,699	194,725	311,727

Table 4. Benchmark characteristics. The "# of classes" column is the number of classes containing reachable methods. The "# of methods" column is the number of reachable methods computed by a static 0-CFA call-graph analysis. The "# of bytecodes" column is the number of bytecodes of reachable methods. The "total" columns report numbers for *all* reachable code, whereas the "app only" columns report numbers for only application code (excluding JDK library code).

	# of queries	Bottom-Up analysis						Top-Down analysis			
		summary computation		summary instantiation							
		time	memory	time	memory	# of abstract states		time	memory	# of abstract states	
						queries	total			queries	total
grande2	616	0.6 sec	78 Mb	0.9 sec	61 Mb	616	1,318	1 sec	37 Mb	616	3,959
grande3	4,236	43 sec	224 Mb	1:21 min	137 Mb	4,373	8,258	1:11 min	506 Mb	4,354	27,232
antlr	5,838	16 sec	339 Mb	30 sec	149 Mb	6,207	21,437	1:23 min	1.1 Gb	8,388	79,710
weka	2,205	46 sec	503 Mb	2:48 min	228 Mb	2,523	25,147	$>$ 6 hrs	26 Gb	5,694	688,957
bloat	10,237	3:03 min	573 Mb	30 min	704 Mb	36,779	131,665	$>$ 6 hrs	24 Gb	139,551	962,376

Table 5. The number of queries to connection analysis and three metrics comparing the scalability of the original top-down and our modular bottom-up versions of the analysis on those queries: running time, memory consumption, and number of incoming abstract states computed at program points of interest. These points include only query points in the "query" sub-columns and all points in the "total" sub-columns. All three metrics show that the top-down analysis scales much more poorly than the bottom-up analysis.

umn (c) of Fig. 4 provides more detailed measurements of the latter numbers. The graphs there show, for each distinct number of incoming states computed at each program point (on the X axis), the fraction of program points (on the Y axis) with equal or smaller number of incoming states. The numbers for the top-down analysis in the graphs marked (*) were obtained at the instant of timeout. The graphs clearly show the blow-up in the number of states computed by the top-down analysis over the bottom-up analysis.

7. Conclusions

We show using lattice theory that when an abstract domain has enough right-modular elements to allow transfer functions to be expressed as joins and meets with constant elements—and the elements used in the meet are right-modular—a compositional (bottom-up) interprocedural analysis can be as precise as a top-down analysis. Using the above, we developed a new bottom-up interprocedural algorithm for connection pointer analysis of Java programs. Our experiments indicate that, in practice, our algorithm is nearly as precise as the existing algorithm, while scaling significantly better. In [4] we apply the same technique to derive a new bottom-up analysis for a variant of the copy-constant propagation problem [10]. The algorithm utilizes a sophisticated join to compute the effect of copy statements of the form `x:=y`. Notice that this is not simple under our restrictions since constant values of y are propagated into x. Indeed, we found that designing the right join operator is the key step when using our approach.

Acknowledgments. Noam Rinetzky was supported by the EU project ADVENT, grant number: 308830.

References

[1] T. Ball and S. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *PASTE*, pages 97–103, 2001.

[2] E. Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *ACM SIGPLAN SOAP Workshop*, pages 3–8, 2012.

[3] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6), 2011.

[4] G. Castelnuovo. Modular lattices for compositional interprocedural analysis. Master's thesis, School of Computer Science, Tel Aviv University, 2012.

[5] R. Chatterjee, B. G. Ryder, and W. Landi. Relevant context inference. In *POPL*, pages 133–146, 1999.

[6] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E. Neuhold, editor, *Formal Descriptions of Programming Concepts*, pages 237–277. North-Holland, 1978.

[7] P. Cousot and R. Cousot. Modular static program analysis. In *CC*, pages 159–178, 2002.

[8] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI*, pages 567–5–77, 2011.

[9] J. Dolby, S. Fink, and M. Sridharan. T. J. Watson Libraries for Analysis. `http://wala.sourceforge.net/`, 2006.

[10] C. N. Fischer, R. K. Cytron, and R. J. LeBlanc. *Crafting A Compiler*. Addison-Wesley Publishing Company, USA, 1st edition, 2009.

[11] R. Ghiya and L. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *IJPP*, 24(6):547–578, 1996.

[12] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract domains condensing. *ACM Trans. Comput. Log*, 6(1):33–60, 2005.

[13] G. Grätzer. *General Lattice Theory*. Birkhäuser Verlag, 1978.

[14] B. Gulavani, S. Chakraborty, G. Ramalingam, and A. Nori. Bottom-up shape analysis using lisf. *ACM TOPLAS*, 33(5):17, 2011.

[15] R. Madhavan, G. Ramalingam, and K. Vaswani. Purity analysis: An abstract interpretation formulation. In *SAS*, pages 7–24, 2011.

[16] M. Naik. Chord: A program analysis platform for Java. Available at `http://pag.gatech.edu/chord/`, 2006.

[17] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[18] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *TCS*, 167, 1996.

[19] A. Salcianu and M. Rinard. Purity and side effect analysis for Java programs. In *VMCAI*, pages 199–215, 2005.

[20] M. Sharir and A. Pnueli. Two approaches to interprocedural data

flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7. 1981.

[21] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. In *OOPSLA*, pages 187–206, 1999.

8. Compositional Constant Propagation Analysis

In this section we describe an encoding of a bottom-up interprocedural copy constant propagation analysis as a triad analysis.

8.1 Programming Language

We define a simple programming language which manipulates integer variables. The language is defined according to the requirements of our general framework. (See Sec. 3.) In this section, we assume that programs have only global integer variables $g \in G$ which are initialized to 0. We also assume that the primitive commands $a \in \mathsf{PComm}$ are of the form

`x := c` , `x := y` , and `x := *`,

pertaining to assignments to a variable `x` of a constant value `c`, of the value of a variable `y`, or of an unknown value, respectively. We denote by $\mathsf{K}_P \subset_{fin} \mathcal{N}$ the finite set of constants which appear in a program P. We assume K_P contains 0. We denote by $\mathsf{G}_P \subset_{fin} G$ the finite set of global variables which appear in a program P.

In the following, we assume a fixed arbitrary program P and denote by $\mathsf{K} = \mathsf{K}_P$ the (fixed finite) set of constants that appear in P, and by $\mathsf{G} = \mathsf{G}_P$ the (fixed finite) set of global variables of P.

For technical reasons, explained in Sec. 8.5, we assume that the analyzed program contains a special global variable `t` which is not used directly by the program, but is used only to implement copy assignments of the form `x := y` using the following sequence of assignments

```
t:=y; y:=0; x:=0; y:=t; x:=t; t:=0.
```

(Note that, in particular, we assume that there are no statements of the form `x := x`.)

8.2 Concrete Semantics

8.2.1 Standard Intraprocedural Concrete Semantics

A *standard memory state* $s \in \mathcal{S} = \mathsf{G} \mapsto \mathcal{N}$ maps variables to their integer values. The meaning of primitive commands $a \in \mathsf{AComm}$ is standard, and defined below.

$$\begin{array}{rcl} [\![x := c]\!](s) & = & \{(s[x \mapsto [\![c]\!]])\} \\ [\![x := y]\!](s) & = & \{(s[x \mapsto s(y)])\} \\ [\![x := *]\!](s) & = & \{(s[x \mapsto n]) \mid n \in \mathcal{N}\} \end{array}$$

Note that $[\![a]\!] : \mathcal{S} \to 2^{\mathcal{S}}$ for a primitive a.

8.2.2 Relational Concrete Semantics

An *input-output pair of standard memory states* $r = (s, s') \in \mathcal{R} = \mathcal{S} \times \mathcal{S}$ records the values of variables at the entry to the procedure (s) and at the current state (s'). The meaning of intraprocedural statements in lifted to input-output pairs as described in Sec. 3. The interprocedural semantics is defined, as described in Sec. 3, using the functions $\cdot|_G : \mathcal{S} \to \mathcal{S}$ and $[\![\mathsf{combine}]\!] : \mathcal{S} \times \mathcal{S} \times \mathcal{S} \to \mathcal{R}$, whose meaning is defined below:

$$\begin{array}{rcl} s|_G & = & s \\ [\![\mathsf{combine}]\!](s_1, s_2, s_3) & = & (s_1, s_3) \end{array}$$

Informally, the projection of the state on its global part does not modify the state due to our assumption a state is a mapping of *global* variables to values. For a similar reason, the combination of the caller's input-output pair at the call-site with that of the callee at the exit-site results in a pair of memory states where the first one records the memory at the entry-site of the caller and the second one records the state at the exit-site of the callee.

8.3 Abstract Semantics

Notations. For every global variable $g \in \mathsf{G}$, $\overline{g}$ denotes the value of g at the entry to a procedure and g' denotes its current value. Similarly to the connection analysis, we use an additional set $\dot{\mathsf{G}}$ of variables to compute the effect of procedure calls. We denote by $\upsilon \in \Upsilon = \mathsf{G}' \cup \overline{G} \cup \dot{\mathsf{G}}$ the set of all *annotated variables* which are ranged by a meta variable υ.

We denote by $\zeta \in \mathsf{VAL} = \Upsilon \cup \mathsf{K} \cup \{*\}$ the set of *abstract values* ranged over by ζ. VAL is comprised of annotated variables, constants which appear in the program, and the special value $*$.

8.3.1 Abstract Domain

Let $\mathcal{D}_{map}$ be the set of all maps from variables $\upsilon \in \Upsilon$ to 2^{VAL}

$$\mathcal{D}_{map} = \Upsilon \mapsto 2^{\mathsf{VAL}} .$$

We denote the set $\mathcal{D}_{trans} \subseteq \mathcal{D}_{map}$ of *transitively closed maps* by

$$\mathcal{D}_{trans} = \{\lceil d \rceil \mid d \in \mathcal{D}_{map}\} ,$$

where

$$\lceil d \rceil = \lambda\upsilon \in \Upsilon.\{\upsilon\} \cup \left\{ \zeta \in d(\upsilon_n) \;\middle|\; \begin{array}{c} \exists \upsilon_0, \ldots, \upsilon_n.\, \upsilon_0 = \upsilon \wedge \\ \forall 0 \le i < n.\, \upsilon_{i+1} \in d(\upsilon_i) \end{array} \right\} .$$

Note that a map $d \in \mathcal{D}_{map}$ is *transitively closed*, i.e., $d \in \mathcal{D}_{trans}$, if and only if it associates υ to a set containing υ, i.e., $\upsilon \in d(\upsilon)$, and for any $\upsilon' \in d(\upsilon)$ it holds that $d(\upsilon') \subseteq d(\upsilon)$.

The abstract domain $\mathcal{D}$ of the copy constant propagation analysis is an augmentation of $\mathcal{D}_{trans}$ with an explicit bottom element.

$$\mathcal{D} = \langle \mathcal{D}_{const}, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle, \text{ where}$$

$$\mathcal{D}_{const} = \mathcal{D}_{trans} \cup \{\bot\}$$

$$d_1 \sqsubseteq d_2 \Leftrightarrow d_1 = \bot \vee \forall \upsilon \in \Upsilon.\, d_1(\upsilon) \subseteq d_2(\upsilon)$$

$$\top = \lambda\upsilon \in \Upsilon\,.\mathsf{VAL}$$

$$d_1 \sqcup d_2 = \begin{cases} d_1 & d_2 = \bot \\ d_2 & d_1 = \bot \\ \lceil d_1 \cup d_2 \rceil & \text{otherwise} \end{cases}$$

$$d_1 \sqcap d_2 = \begin{cases} \bot & d_1 = \bot \vee d_2 = \bot \\ d_1 \cap d_2 & \text{otherwise} \end{cases}$$

8.3.2 Abstract Intraprocedural Transformers

The abstract meaning of the primitive intraprocedural statements is defined as follows:

$$\begin{array}{rcl} [\![x := c]\!]^\sharp & = & (\lambda d.d \sqcap S_{x'}) \sqcup U_{x'c} \\ [\![x := y]\!]^\sharp & = & (\lambda d.d \sqcap S_{x'}) \sqcup U_{x'y'} \\ [\![x := *]\!]^\sharp & = & (\lambda d.d \sqcap S_{x'}) \sqcup U_{x'*} \end{array}$$

where

$$S_{x'}(\upsilon) \quad = \quad \lambda\upsilon \in \Upsilon. \begin{cases} \{x'\} & \upsilon = x' \\ \mathsf{VAL} \setminus \{x'\} & \upsilon \neq x' \end{cases}$$

$$U_{x'\zeta} \quad = \quad \lambda\upsilon \in \Upsilon. \begin{cases} \{x', \zeta\} & \upsilon = x' \\ \{\upsilon\} & \upsilon \neq x' \end{cases}$$

In the following, we show that the abstract transfer functions of the copy constant propagation analysis are conditionally adaptable. We first prove a simple lemma that holds for every lattice.

LEMMA 7. *For any lattice $(D, \sqsubseteq)$ and elements $d, d', d_s \in D$ such that $d' \sqsubseteq d_s$ it holds that*

$$d' \sqcup (d \sqcap d_s) \sqsubseteq (d' \sqcup d) \sqcap d_s \,.$$

Proof By the definition of $\sqcup$ it holds that

$$d \sqsubseteq (d' \sqcup d) \quad \text{and} \quad d' \sqsubseteq (d' \sqcup d)\,.$$

By the monotonicity of $\sqcap$, we get that

$$d \sqcap d_s \sqsubseteq (d' \sqcup d) \sqcap d_s \quad \text{and} \quad d' \sqcap d_s \sqsubseteq (d' \sqcup d) \sqcap d_s\,.$$

By the monotonicity and the idempotence of $\sqcup$, we get that

$$(d' \sqcap d_s) \sqcup (d \sqcap d_s) \sqsubseteq (d' \sqcup d) \sqcap d_s\,.$$

By the assumption $d' \sqsubseteq d_s$. Hence, $d' \sqcap d_s = d'$, and it follows that

$$d' \sqcup (d \sqcap d_s) \sqsubseteq (d' \sqcup d) \sqcap d_s\,.$$

□

DEFINITION 8 (General projection and separation elements). *Let $X \subseteq \Upsilon$. We denote by*

$$S_X(\upsilon) = \lambda \upsilon \in \Upsilon. \begin{cases} \{\upsilon\} & \upsilon \in X \\ \mathsf{VAL} \setminus X & \upsilon \notin X \end{cases}$$

the separation element *of X and by*

$$R_X = S_{\Upsilon \setminus X}$$

the projection element *of X.*

LEMMA 9. *For every $X \subseteq \Upsilon$, S_X is right-modular.*

Proof We need to prove that for all $d, d' \in \mathcal{D}$ such that $d' \sqsubseteq S_X$ it holds that

$$(d' \sqcup d) \sqcap S_X = d' \sqcup (d \sqcap S_X)\,.$$

By Lem. 7, it holds that

$$(d' \sqcup d) \sqcap S_X \sqsupseteq d' \sqcup (d \sqcap S_X)\,.$$

Thus it suffices to show that

$$d_1 = (d' \sqcup d) \sqcap S_X \sqsubseteq d' \sqcup (d \sqcap S_X) = d_2\,. \qquad (*)$$

We prove (*) by induction on the size of X.

Base Case. For $|X| = 1$, we get that $S_X = S_{\{\upsilon\}}$ for some $\upsilon \in \Upsilon$. Pick $\upsilon_0 \in \Upsilon$. We need to show that $d_1(\upsilon_0) \subseteq d_2(\upsilon_0)$. There can be two case: either $\upsilon_0 = \upsilon$ or not.

If $\upsilon_0 = \upsilon$, then $d_1(\upsilon_0) \subseteq S_{\{\upsilon'\}}(\upsilon_0) = \{\upsilon_0\}$. By definition of the domain, which includes only transitively closed maps, $\upsilon_0 \in d'(\upsilon_0)$. Hence, $d_1(\upsilon_0) \subseteq \{\upsilon_0\} \subseteq d_2(\upsilon_0)$.

Otherwise, $\upsilon_0 \neq \upsilon$. Pick $\upsilon_1 \in d_1(\upsilon_0)$. By the definition of $S_{\{\upsilon\}}$ and the meet operation, $\upsilon_1 \neq \upsilon$, and again by the definition of the meet operation, $\upsilon_1 \in (d' \sqcup d)(\upsilon_0)$. Thus, there exists a minimal sequence $\zeta_0, \ldots, \zeta_n$ such that $\zeta_0 = \upsilon_0 \wedge \zeta_n = \upsilon_1$ and for all $0 \leq i < n$, $\zeta_{i+1} \in d'(\zeta_i) \cup d(\zeta_i)$.

CLAIM 10. *For all $0 \leq j < n$, $\zeta_j \neq \upsilon$.*

Proof $\zeta_0 = \upsilon' \neq \upsilon$, by the assumption.

Assume that there exists some $0 < j < n$ such that $\zeta_j = \upsilon$. By the minimality of the sequence, $\zeta_{j-1} \neq \upsilon$ and $\zeta_{j+1} \neq \upsilon$, and since $d' \sqsubseteq S_{\{\upsilon\}}$, $\zeta_j = \upsilon \notin d'(\zeta_{j-1})$ and $\zeta_{j+1} \notin d'(\zeta_j) = d'(\upsilon)$ and thus $\zeta_j \in d(\zeta_{j-1})$ and $\zeta_{j+1} \in d(\zeta_j)$. d is transitively closed and from this, $\zeta_{j+1} \in d(\zeta_{j-1})$. Thus, the sequence $\zeta_0, \ldots, \zeta_{j-1}, \zeta_{j+1}, \ldots, \zeta_n$ is a valid sequence for υ', υ'', and this is a contradiction to the minimality of the original sequence.

□

By the claim, we got that for all $0 \leq j < n$,

$$\begin{array}{rcll} \zeta_{i+1} & \in & (d'(\zeta_i) \cup d(\zeta_i)) \setminus \{\upsilon\} & \text{By the previous claim} \\ \Rightarrow \zeta_{i+1} & \in & (d'(\zeta_i) \cup d(\zeta_i)) \cap S_{\{\upsilon\}}(\zeta_i) & \text{By the definition of } S_{\{\upsilon\}} \\ & = & (d'(\zeta_i) \cap S_{\{\upsilon\}}(\zeta_i)) \cup (d(\zeta_i) \cap S_{\{\upsilon\}}(\zeta_i)) & \\ & = & d'(\zeta_i) \cup (d(\zeta_i) \cap S_{\{\upsilon\}}(\zeta_i)) & \text{By } d'(\zeta_i) \subseteq S_{\{\upsilon\}}(\zeta_i) \\ & = & d'(\zeta_i) \cup (d \sqcap S_{\{\upsilon\}})(\zeta_i)\,. & \end{array}$$

Therefore we can create a sequence for υ', υ'', by taking elements only from $d' \cup (d \sqcap S_{\{\upsilon\}})$, and hence $\upsilon'' \in (d' \sqcup (d \sqcap S_{\{\upsilon\}}))(\upsilon')$.

Induction Step. Assume that the induction assumption holds for sets X such that $|X| = n$, we will prove for sets X such that $|X| = n + 1$. Let $\upsilon \in X$ be an arbitrary element. Notice that by its definition

$$S_X = S_{X \setminus \{\upsilon\}} \sqcap S_{\{\upsilon\}}\,.$$

Therefore,

$$\begin{array}{rcl} (d' \sqcup d) \sqcap S_X & = & (d' \sqcup d) \sqcap (S_{X \setminus \{\upsilon\}} \sqcap S_{\{\upsilon\}}) \\ & = & ((d' \sqcup d) \sqcap S_{X \setminus \{\upsilon\}}) \sqcap S_{\{\upsilon\}} \\ & \sqsubseteq & (d' \sqcup (d \sqcap S_{X \setminus \{\upsilon\}})) \sqcap S_{\{\upsilon\}} \\ & = & d' \sqcup ((d \sqcap S_{X \setminus \{\upsilon\}}) \sqcap S_{\{\upsilon\}}) \\ & = & d' \sqcup (d \sqcap (S_{X \setminus \{\upsilon\}} \sqcap S_{\{\upsilon\}})) \\ & = & d' \sqcup (d \sqcap S_X) \end{array}$$

□

LEMMA 11. *The abstract transfer functions of the atomic commands are conditionally adaptable.*

Proof By Lem. 9, $S'_x = S_{\{x'\}}$ is right-modular and all the transfer functions are of form

$$f = \lambda d.((d \sqcap d_p) \sqcup d_g)\,.$$

where $d_p = S_{x'}$ for some $x' \in \Upsilon$.

□

8.4 Soundness of the Top Down Analysis

The soundness of the copy constant propagation analysis is formalized by the concretization function $\gamma : \mathcal{D} \to 2^{\mathcal{S} \times \mathcal{S}}$, where

$$\begin{array}{l} (s, s') \in \gamma(d) \iff \\ \quad \big(\forall \overline{x} \in \overline{\mathsf{G}}.\, (s(x) \in d(\overline{x}) \cap \mathsf{K}) \vee (* \in d(\overline{x}))\big) \bigwedge \\ \quad \big(\forall x' \in \mathsf{G}'.\, s'(x) \in ((d(\overline{x}) \cap \mathsf{K}) \cup \{s(y) \mid \overline{y} \in d(\overline{x}))\}) \vee (* \in d(\overline{x}))\big)\,. \end{array}$$

Intuitively, an input-output pair (s, s') is *conservatively represented* by an abstract element d if and only if (a) the input state maps a variable g to n if n is one of the constants mapped to $\overline{g}$ by d or if $* \in d(\overline{g})$ and (b) the output state maps a variable g to n if n is one of the constants mapped to g' by d, the value of global variable y at the entry state that is mapped to g' by d, or if $* \in d(g')$.

LEMMA 12 (Soundness). *The abstract transformers pertaining to intraprocedural primitive commands, $|_G$, and* combine *over-approximate the concrete ones.*

8.5 Precision Improving Transformations

In Sec. 8.1, we place certain restrictions on the analyzed programs. Specifically, we forbid copy assignments of the form x:=y between arbitrary global variables `x` and `y`, and, instead, require that the value of `y` be copied to `x` through a sequence of assignments that use a temporary variable `t`. In the concrete semantics, our requirements do not affect the values of the program's variables outside of the sequences of intermediate assignments. In the abstract semantics, however, adhering to our requirements can improve the precision of the analysis, as we explain below.

Consider the execution of the sequence of abstract transformers pertaining to the (non deterministic) command x:=y + y:=3 on an abstract state d, in which $3 \notin d(y')$. Applying the abstract transformer $[\![\texttt{x := y}]\!]^\sharp$ to d results in an abstract element d', where $y' \in d(x')$. Applying $[\![\texttt{y := 3}]\!]^\sharp$ to d results in an abstract state d'', where $3 \in d''(y')$. Perhaps surprisingly, in the abstract state $d''' = d' \sqcup d''$, which conservatively represent the possible states after the non-deterministic choice (+), we get that $3 \in d'''(x')$. This is sound, but imprecise. The reason for the imprecision is that our domain includes only transitively closed maps and having $y' \in d'(x')$ results in an undesired correlation between the possible values of x in d''' and that of y in d''. In particular, the assignment of 3 to y is propagated to x in a flow-insensitive manner.

Rewriting the copy assignment using t according to our restrictions breaks such undesired correlations. Consider, for example, the sequence of abstract transformers pertaining to the aforementioned command: `t:=y; y:=0; x:=0; y:=t; x:=t; t:=0`, and apply this sequence to d. In the abstract state $\hat{d}$ arising just before t is assigned 0 we get that $t' \in \hat{d}(x')$ and $t' \in \hat{d}(y')$ but $y' \notin \hat{d}(x')$ and $x' \notin \hat{d}(y')$. Assigning 0 to t breaks the correlation between t and x and y.

8.6 Copy Constant Propagation as a Triad Analysis

8.6.1 Triad Domain

LEMMA 13. *$\mathcal{D}$ is a triad domain.*

Projection Elements

Proof We define the projection elements

$$\begin{array}{rcl} d_{\mathsf{proj}_{\mathsf{in}}} & = & R_{\overline{\mathsf{G}}} \\ d_{\mathsf{proj}_{\mathsf{tmp}}} & = & R_{\dot{\mathsf{G}}} \\ d_{\mathsf{proj}_{\mathsf{out}}} & = & R_{\mathsf{G}'} \end{array}$$

By Lem. 9, $d_{\mathsf{proj}_{\mathsf{in}}}$, $d_{\mathsf{proj}_{\mathsf{tmp}}}$ and $d_{\mathsf{proj}_{\mathsf{out}}}$ are right-modular.

Isomorphism functions We define the renaming functions

$$f^{\Upsilon}_{\mathsf{call}}, f^{\Upsilon}_{\mathsf{exit}}, f^{\Upsilon}_{\mathsf{inout}} \colon \Upsilon \to \Upsilon$$

$$f^{\Upsilon}_{\mathsf{call}}(\tilde{\upsilon}) = \begin{cases} \dot{\upsilon} & \tilde{\upsilon} = \upsilon' \\ \upsilon' & \tilde{\upsilon} = \dot{\upsilon} \\ \tilde{\upsilon} & otherwise \end{cases}$$

$$f^{\Upsilon}_{\mathsf{exit}}(\tilde{\upsilon}) = \begin{cases} \dot{\upsilon} & \tilde{\upsilon} = \overline{\upsilon} \\ \overline{\upsilon} & \tilde{\upsilon} = \dot{\upsilon} \\ \tilde{\upsilon} & otherwise \end{cases}$$

$$f^{\Upsilon}_{\mathsf{inout}}(\tilde{\upsilon}) = \begin{cases} \overline{\upsilon} & \tilde{\upsilon} = \upsilon' \\ \upsilon' & \tilde{\upsilon} = \overline{\upsilon} \\ \tilde{\upsilon} & otherwise \end{cases}$$

Let $f^{\mathsf{VAL}}_{\mathsf{call}}$, $f^{\mathsf{VAL}}_{\mathsf{exit}}$, $f^{\mathsf{VAL}}_{\mathsf{inout}}$ be the renaming function induced on 2^{VAL} and finally let f_{call}, f_{exit}, f_{inout} be the renaming functions induced on $\mathcal{D}$,

$$f_i(d) = \begin{cases} \bot & d = \bot \\ \lambda\upsilon \in \Upsilon.\, f^{\mathsf{VAL}}_i(d({f^{\Upsilon}_i}^{-1}(\upsilon))) & \text{otherwise} \end{cases} \tag{12}$$

where $i \in [\mathsf{call}, \mathsf{inout}, \mathsf{exit}]$.

CLAIM 14.

$$\begin{array}{rlrlrl} f_{\mathsf{call}}(R_{\mathsf{G}'}) & = R_{\dot{\mathsf{G}}}, & f_{\mathsf{call}}(R_{\overline{\mathsf{G}}}) & = R_{\overline{\mathsf{G}}}, & f_{\mathsf{call}}(R_{\dot{\mathsf{G}}}) & = R_{\mathsf{G}'} \\ f_{\mathsf{exit}}(R_{\mathsf{G}'}) & = R_{\mathsf{G}'}, & f_{\mathsf{exit}}(R_{\overline{\mathsf{G}}}) & = R_{\dot{\mathsf{G}}}, & f_{\mathsf{exit}}(R_{\dot{\mathsf{G}}}) & = R_{\overline{\mathsf{G}}} \\ f_{\mathsf{inout}}(R_{\mathsf{G}'}) & = R_{\overline{\mathsf{G}}}, & f_{\mathsf{inout}}(R_{\overline{\mathsf{G}}}) & = R_{\mathsf{G}'}, & f_{\mathsf{inout}}(R_{\dot{\mathsf{G}}}) & = R_{\dot{\mathsf{G}}} \end{array}$$

Proof We prove the claim on f_{call} and $R_{\mathsf{G}'}$. The other cases are symmetric.

$$\begin{array}{rcl} f_{\mathsf{call}}(R_{\mathsf{G}'}) & = & f_{\mathsf{call}}\left(\lambda\upsilon \in \Upsilon.\begin{cases} \{\upsilon\} & \upsilon \in \Upsilon \setminus \mathsf{G}' \\ \mathsf{K} \cup \mathsf{G}' & \upsilon \in \mathsf{G}' \end{cases}\right) \\ & = & \lambda\upsilon \in \Upsilon.\begin{cases} \{\upsilon\} & \upsilon \in \Upsilon \setminus \dot{\mathsf{G}} \\ \mathsf{K} \cup \dot{\mathsf{G}} & \upsilon \in \dot{\mathsf{G}} \end{cases} \\ & = & R_{\dot{\mathsf{G}}} \end{array}$$

□

CLAIM 15. *For all $d \in \mathcal{D}_{\mathsf{out}}$*

$$f_{\mathsf{exit}}(f_{\mathsf{inout}}(d)) = f_{\mathsf{call}}(d)$$

Proof Let $d \in \mathcal{D}_{\mathsf{out}}$ and let $\upsilon \in \Upsilon$. If $d = \bot$ then

$$f_{\mathsf{call}}(d) = f_{\mathsf{exit}}(f_{\mathsf{inout}}(d)) = \bot\,.$$

Otherwise, by Eq.12,

$$f_{\mathsf{call}}(d)(\upsilon) = f^{\mathsf{VAL}}_{\mathsf{call}}(d({f^{\Upsilon}_{\mathsf{call}}}^{-1}(\upsilon)))$$

and

$$\begin{array}{rcl} ((f_{\mathsf{exit}} \circ f_{\mathsf{inout}})(d))(\upsilon) & = & f_{\mathsf{exit}}(f_{\mathsf{inout}}(d))(\upsilon) \\ & = & f^{\mathsf{VAL}}_{\mathsf{exit}}(f_{\mathsf{inout}}(d)({f^{\Upsilon}_{\mathsf{exit}}}^{-1}(\upsilon)) \\ & = & f^{\mathsf{VAL}}_{\mathsf{exit}}(f^{\mathsf{VAL}}_{\mathsf{inout}}(d({f^{\Upsilon}_{\mathsf{inout}}}^{-1}({f^{\Upsilon}_{\mathsf{exit}}}^{-1}(\upsilon)) \\ & = & (f^{\mathsf{VAL}}_{\mathsf{exit}} \circ f^{\mathsf{VAL}}_{\mathsf{inout}})(d({f^{\Upsilon}_{\mathsf{inout}}}^{-1} \circ {f^{\Upsilon}_{\mathsf{exit}}}^{-1})(\upsilon)) \end{array}$$

If $\upsilon \notin \dot{\mathsf{G}}$, then

$${f^{\Upsilon}_{\mathsf{call}}}^{-1}(\upsilon) \notin \mathsf{G}'$$

and

$${f^{\Upsilon}_{\mathsf{inout}}}^{-1} \circ {f^{\Upsilon}_{\mathsf{exit}}}^{-1}(\upsilon) \notin \mathsf{G}'$$

and therefore

$$d({f^{\Upsilon}_{\mathsf{call}}}^{-1}(\upsilon)) = \{{f^{\Upsilon}_{\mathsf{call}}}^{-1}(\upsilon)\}$$

and

$$d({f^{\Upsilon}_{\mathsf{inout}}}^{-1} \circ {f^{\Upsilon}_{\mathsf{exit}}}^{-1}(\upsilon)) = \{{f^{\Upsilon}_{\mathsf{inout}}}^{-1} \circ {f^{\Upsilon}_{\mathsf{exit}}}^{-1}(\upsilon)\}$$

Hence,

$$f_{\mathsf{call}}(d)(\upsilon) = f^{\mathsf{VAL}}_{\mathsf{call}}(\{{f^{\Upsilon}_{\mathsf{call}}}^{-1}(\upsilon)\}) = \{\upsilon\}$$

and

$$(f_{\mathsf{exit}} \circ f_{\mathsf{inout}})(d)(\upsilon) = (f^{\mathsf{VAL}}_{\mathsf{exit}} \circ f^{\mathsf{VAL}}_{\mathsf{inout}})(\{{f^{\Upsilon}_{\mathsf{inout}}}^{-1} \circ {f^{\Upsilon}_{\mathsf{exit}}}^{-1}(\upsilon)\}) = \{\upsilon\}$$

Otherwise, if $\upsilon \in \dot{\mathsf{G}}$, by the definition of the renaming functions

$${f^{\Upsilon}_{\mathsf{call}}}^{-1}(\upsilon) = ({f^{\Upsilon}_{\mathsf{inout}}}^{-1} \circ {f^{\Upsilon}_{\mathsf{exit}}}^{-1})(\upsilon)$$

and by the definition of $\mathcal{D}_{\mathsf{out}}$,

$$d({f^{\Upsilon}_{\mathsf{call}}}^{-1}(\upsilon)) \subseteq G'$$

and therefore again by the definition of the renaming functions

$$f^{\mathsf{VAL}}_{\mathsf{call}}(d({f^{\Upsilon}_{\mathsf{call}}}^{-1}(\upsilon))) = (f^{\mathsf{VAL}}_{\mathsf{exit}} \circ f^{\mathsf{VAL}}_{\mathsf{inout}})(d({f^{\Upsilon}_{\mathsf{call}}}^{-1}(\upsilon)))$$

□

CLAIM 16. *For all $d \in \mathcal{D}_{\mathsf{in}}$*

$$f_{\mathsf{call}}(d) = d$$

Proof By the definition of $\mathcal{D}_{\mathsf{in}}$ and of f_{call}.

□

ι_{entry} ***element*** We define

$\iota_{\mathsf{entry}} = [\upsilon' \mapsto \{\upsilon', \overline{\upsilon}\} \mid \upsilon \in \mathsf{G}] \cup [\overline{\upsilon} \mapsto \{\upsilon', \overline{\upsilon}\} \mid \upsilon \in \mathsf{G}] \cup [\dot{\upsilon} \mapsto \{\dot{\upsilon}\} \mid \upsilon \in \mathsf{G}]$

CLAIM 17. *For every* $d \in \mathcal{D}_{\mathsf{out}}$, $d \sqcup \iota_{\mathsf{entry}} = f_{\mathsf{inout}}(d) \sqcup \iota_{\mathsf{entry}}$.

Proof

$$
\begin{aligned}
d \sqcup \iota_{\mathsf{entry}} &= \lceil d \cup \iota_{\mathsf{entry}} \rceil \\
&= \left\lceil \lambda \upsilon \in \Upsilon.\ \begin{cases} d(g') \cup \{g', \bar{g}\} & \upsilon = g' \in \mathsf{G}' \\ \{\bar{g}\} \cup \{g', \bar{g}\} & \upsilon = \bar{g} \in \overline{\mathsf{G}} \\ \{\dot{g}\} \cup \{\dot{g}\} & \upsilon = \dot{g} \in \dot{\mathsf{G}} \end{cases} \right\rceil \\
&= \lambda \upsilon \in \Upsilon.\ \begin{cases} \{\bar{h}, h' \mid h' \in d(g')\} & \upsilon = g' \in \mathsf{G}' \vee \upsilon = \bar{g} \in \overline{\mathsf{G}} \\ \{\dot{g}\} & \upsilon = \dot{g} \in \dot{\mathsf{G}} \end{cases} \\
&= \left\lceil \lambda \upsilon \in \Upsilon.\ \begin{cases} \{g'\} \cup \{g', \bar{g}\} & \upsilon = g' \in \mathsf{G}' \\ \{\bar{h} \mid h' \in d(g')\} \cup \{g', \bar{g}\} & \upsilon = \bar{g} \in \overline{\mathsf{G}} \\ \{\dot{g}\} \cup \{\dot{g}\} & \upsilon = \dot{g} \in \dot{\mathsf{G}} \end{cases} \right\rceil \\
&= \left\lceil \lambda \upsilon \in \Upsilon.\ \begin{cases} \{g'\} \cup \{g', \bar{g}\} & \upsilon = g' \in \mathsf{G}' \\ f_{\mathsf{inout}}(d)(\bar{g}) \cup \{g', \bar{g}\} & \upsilon = \bar{g} \in \overline{\mathsf{G}} \\ \{\dot{g}\} \cup \{\dot{g}\} & \upsilon = \dot{g} \in \dot{\mathsf{G}} \end{cases} \right\rceil \\
&= f_{\mathsf{inout}}(d) \sqcup \iota_{\mathsf{entry}}
\end{aligned}
$$

□

□