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%.

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.

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.

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 p_0, which invokes p_1 with an actual parameter a_0 or b_0. For every 1 ≤ i ≤ n, procedure p_i either assigns a_i with formal parameter c_{i-1} and invokes procedure p_{i+1} with an actual parameter a_i or assigns b_i with formal parameter c_{i-1} and invokes procedure p_{i+1} with an actual parameter b_i. Procedure p_n either assigns a_n or b_n with formal parameter 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(*) {a3 = c0; p2(a3)} else {b1 = c0; p2(b1)}

    p2(c1) {
        if(*) {a2 = c1; p3(a2)} else {b2 = c1; p3(b2)}
    }

    p0(c0-1) if(*) {a0 = c0-1 else b0 = c0-1}
    ...
}

Figure 1. Example program.

Figure 2. Concrete states at the entry of procedure p1 (see Fig. 1) and the corresponding connection and points-to abstractions.

d_conn = \{\{g1, a0, c0\}, \{g2, b0\}, \{a1\}...\{a_n\}, \{b1\}\}
d_point = \{(a0, h3), (b0, h4), (c0, h3), (h3, f, h1), (h4, f, h2), (g1, h1), (g2, h2)\}

Figure 3. Connection abstraction at the entry of procedure p0 of the program in Fig. 1.

by b0 points to the object pointed to by g2. As a result, there are two connection sets \{a0, g1, c0\} and \{b0, g2\}. The second calling context is similar to the first, except that c0 is aliased with b0 instead of a0. The connection sets are changed accordingly, and they are \{a0, g1\} and \{b0, g2, c0\}. In both cases, the other variables are pointing to nu11, and thus are not connected to any variable.

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 (x, h) or (h, f, h2). The pair (x, h) means that variable x may point to an object allocated at the site h, and the tuple (h, f, h2) means that the f field of an object allocated at h1 may point to an object allocated at h2. Fig. 2 depicts the abstract states at the entry to procedure p1. Also in this case, there are two calling abstract contexts for p1. In one of them, c0 may point to h3, and in the other, c0 may point to h2.

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 p1 there are two calls to procedure p1+1, where for each one of them, the connection and the points-to analyses compute two different calling contexts for procedure p1+1. Therefore, in both the analyses, the number of calling contexts at the entry of procedure p1 is 2^2.

Fig. 3 shows the connection-abstraction at the entry of procedure p0. Each abstract state in the abstraction corresponds to one path to p0. For example, the first state corresponds to selecting the then-branch in all p0, p0+1, while the second state corresponds to selecting the then-branch in all p0, p0+1, and the else-branch in p0+. Finally, the last state corresponds to selecting the else-branch in all p0, p0+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 a0 = c0-1 in the points-to analysis is

\[ [a0 = c0-1]_P(d) = (d \setminus \{a1, z\} z \in Var) U \{\{a0, w\} | \{c0-1, w\} \in d\} \]
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 lattice 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 \( \cap \) and \( \cup \) 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 \( D \) be a lattice. A pair of elements \( (d_0, d_1) \) is called **modular**, denoted by \( d_0 \sqsubseteq d_1 \), if

\[
d
\implies (d \sqcup d_0) \sqcap d_1 = d \sqcup (d_0 \sqcap d_1)
\]

An element \( d_1 \) is called **right-modular** if \( d_0 \sqcap d_1 \) holds for all \( d_0 \in D \). \( D \) is called **modular** if \( d_0 \sqcap d_1 \) holds for all \( d_0, d_1 \in D \).

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 \( \neg \sqcap d_0 \) and \( \neg \sqcup d_1 \) for some constant abstract elements \( d_0 \) and \( d_1 \) independent of the input abstract state \( d_0 \) elements are right-modular. Our encoding of points-to analyses 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

\[
[x = y] = \lambda d. (d \sqcap S_x) \sqcup U_{xy}
\]

where \( S_x \) and \( U_{xy} \) 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 \( [a_i = c_{i-1}] \), in the body of some procedure \( P \). Let \( \{d_k\}_k \) denote abstract states at the entry of \( P \), and suppose there is some \( d \) such that

\[
\forall k : \exists d_k \subseteq S_{a_i} : d_k = d \sqcup d_k'.
\]

The compositional approach first chooses the input state \( d \), and computes \( [a_i = c_{i-1}] \). 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 \( [a_i = c_{i-1}] \) directly, as shown below:

\[
[a_i = c_{i-1}] = (d \sqcap S_{a_i}) \sqcup U_{a_i} \sqcap d_k = (d \sqcap S_{a_i}) \sqcup U_{a_i} \sqcap d_k = (d \sqcup d_k') \sqcap S_{a_i} \sqcup U_{a_i} \sqcap d_k = (d_k \sqcup d_k') \sqcap U_{a_i} \sqcap d_k = [a_i = c_{i-1}] \sqcup (d_k).
\]

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 \( \text{PComm} \), \( \text{G} \), and \( \text{PName} \) be sets of primitive commands, global variables, local variables, and procedure names, respectively. We use the following symbols to range over these sets:

\[
a, b \in \text{PComm}, \quad g \in \text{G}, \quad x, y, z \in \text{G} \cup \text{L}, \quad p \in \text{PName}.
\]

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

**Commands** \( C ::= \text{ask} \mid a \mid C; C \mid C + C \mid C^* \mid p() \)

**Declarations** \( D ::= \text{proc} \ p() = [\text{var} \ x; C] \)

**Programs** \( P ::= \text{var} \ g; C \mid P \)

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 \( a \in \text{PComm} \), left unspecified, sequential composition \( C; C' \), non-deterministic choice \( C + C' \), iteration \( C^* \), and procedure call \( p() \). We use \(+\) and \(\ast\) 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 \( x \) 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 \( \text{body}_p \) and \( L_p \) the body of procedure \( p \) and the set of its local variables, respectively.

We assume that \( \text{L} \) and \( \text{G} \) 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,

\[
[p()]^3(d) = \text{return}^3 \circ (\text{body}_p)^3 \circ \text{entry}^3(d, d)
\]

where \( \text{body}_p \) is the body of the procedure \( p \), and

\[
\text{entry}^3 : D \rightarrow D \quad \text{and} \quad \text{return}^3 : D \times D \rightarrow D
\]

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 \( (\sigma, \sigma') \) coming from some unspecified set \( \Sigma \) of memory states. \( \sigma \) 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 \( \sigma' \) is the **current** memory state. We assume that we are given the meaning \( [a] : \Sigma \rightarrow \Sigma \) of every primitive command, and lift it to sets of pairs \( \sigma \in \mathcal{R} = 2^{\Sigma \times \Sigma} \) of memory states by applying it in a pointwise manner to the current states:

\[
\begin{align*}
&\mathcal{R}([a]) = \{ (\sigma, \sigma') \mid (\sigma, \sigma') \in \mathcal{R} \}.
\end{align*}
\]

The meaning of composed commands is standard:

\[
[C_1 + C_2](\rho) = [C_1](\rho) \cup [C_2](\rho)
\]

\[
[C_1; C_2](\rho) = [C_2][C_1](\rho)
\]

\[
[C_1^*](\rho) = \text{leastFix}(\lambda \rho. \rho \cup [C](\rho)).
\]

The effect of procedure invocations is computed using the auxiliary functions \( \text{return} \), \( \text{entry} \), and \( |C| \), which we explain below.

\[
[p()](\rho_c) = \text{return}(\text{body}_p, \rho_c) \circ \text{entry}(\rho_c, \rho_c),
\]

where
Function entry computes the relation \( \rho_e \) at the entry to the invoked procedure. It removes the information regarding the callee’s local variables from the current state \( \sigma_c \) coming from the caller’s relation at the call-site \( \rho_e \) using function \( \langle \cdot | \cdot \rangle \), 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 \( \rho_r \), which updates the caller’s current state with the effect of the callee. The function computes triples \( \langle \pi_c, \sigma_c, \sigma_e | \cdot \rangle \) out of relations \( \langle \pi_c, \sigma_c \rangle \) coming from the caller at the callsite and to the callee at the return site. It returns only those relations where the global part of the callee’s current state matches that of the callee’s entry state. Note that at the triple, the middle state, \( \sigma_{\epsilon} \), contains the values of the caller’s local variables, which the callee cannot modify, and the last state, \( \sigma_e | \cdot \rangle \), 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 \( h \), \( \langle \cdot | \cdot \rangle \) and combine are defined as:

\[
\langle s_g, s_l, h \rangle | \cdot \rangle = \langle s_g, l, h \rangle,
\]

\[
\text{combine}\left( \langle \pi_c, \sigma_c | \cdot \rangle, \langle s_g, s_l, h \rangle, \langle s'_g, s'_l, h' \rangle \right) = \langle \pi_c, \sigma_c, \sigma_e | \cdot \rangle,
\]

where \( \pi_c \subset \pi \).

**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 \( \mathbb{L} \cup \mathbb{G} \) 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 \( \text{Equiv}(\pi) \) the set of equivalence relations over a set \( \pi \). Every equivalence relation on \( \pi \) induces a unique partitioning of \( \pi \) 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 \( \pi \) is a 6-tuple \( \mathcal{D}_\text{part}(\pi) = (\text{Equiv}(\pi), \sqsubseteq, \sqcup, \sqcap, \sqcup, \sqcap) \).

- For any equivalence relations \( d_1, d_2 \in \text{Equiv}(\pi) \),

  \[
  d_1 \sqsubseteq d_2 \iff \forall \pi, \pi_1 \sqsubseteq \pi_2 \Rightarrow d_1 \sqsubseteq \pi_2,
  \]

  where \( \pi_1 \sqsubseteq \pi_2 \) means that \( \pi_1 \) and \( \pi_2 \) are related by \( d_1 \).

- The minimal element \( \sqcap \) is the identity relation, relating each element only with itself.

**Table 1.** Abstract semantics of primitive commands in the connection analysis for \( d \neq \bot \), \( a \mathbb{L}^{\pi}(\bot) = \bot \) for any command \( a \). \( U_{\mathbb{G}} \) is used to merge the connection sets of \( x \) and \( y \). \( S_{\mathbb{G}} \) is used to separate \( x \) from its current set in Sec. 4.1, \( x \) and \( y \) in Sec. 5. \( \alpha \) denotes x’ and \( y \) denotes y’.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = \text{null} )</td>
<td>( y = \text{new} )</td>
</tr>
<tr>
<td>( x = y )</td>
<td>( x \triangleright y )</td>
</tr>
<tr>
<td>( x.f = y )</td>
<td>( d \sqcup U_{\mathbb{G}} )</td>
</tr>
</tbody>
</table>

where

\[
S_{\mathbb{G}} = \{ \{ x \} \} \cup \{ \{ z \mid z \in \mathbb{Y} \setminus \{ x \} \}
\]

\[
U_{\mathbb{G}} = \{ \{ x, y \} \} \cup \{ \{ z \mid z \in \mathbb{Y} \setminus \{ x, y \} \}
\]

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

Assigning \( x = \text{null} \) 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 \( S_{\mathbb{G}} \) — 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 \( S_{\mathbb{G}} \) and then a join with \( U_{\mathbb{G}} \) — 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 \( y \). As a result, the 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 \( U_{\mathbb{G}} \).

**4.2 Conditionally Compositional Intraprocedural Analysis**

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

We focus on static analyses where the transfer function for every atomic command \( a \) is some conditionally adaptable function \( [a] \). We denote the meet elements of \( [a] \) by \( P[a] \). For a command \( C \), we denote by \( P[C] \) 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 \( d_1 \sqsubseteq d_2 \), then \( [C](d_1 \sqcup d_2) = [C](d_1) \sqcup d_2 \).

Lemma 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 $[C]^d_2(d_1 \cup d_2)$ can be done by computing $d = [C]^d_2(d_1)$, 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 $d_2 \subseteq P[C]^d_2$. 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 intraprocedural connection analysis described in Sec. 4.1 to the interprocedural 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 $g \in \mathcal{G}$, $\overline{g}$ 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 $\mathcal{G}$) and the ones pointed to by global variables and local variables at the current state (represented by $\mathcal{G}'$ and $\mathcal{L}'$, respectively).

For technical reasons, described later, we also use the set $G$ 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 $\mathcal{D} = \text{Equiv}(\mathcal{T}) \cup \{\perp\}$ in the same way as in Def. 3 of Sec. 4.1 where $\mathcal{T} = \mathcal{G} \cup \mathcal{G}' \cup \mathcal{G} \cup L'$ and

\[
\mathcal{G}' = \{g' \mid g \in \mathcal{G}\} \\
\mathcal{G} = \{g \mid g \in \mathcal{G}\} \\
\mathcal{T} = \{\overline{g} \mid g \in \mathcal{G}\} \\
\mathcal{L}' = \{x' \mid x \in \mathcal{L}\}
\]

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 of $S(a + b) = a + S(b)$ 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].)

In this section, we define an abstract framework for compositional interprocedural analysis 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].

\[
R_X = \{\{x \mid x \in X\} \cup \{\{x\} \mid x \in \mathcal{T} \setminus X\}
\]
\[
\mathcal{D}_{in} = \{d \in \mathcal{D} \mid d \subseteq R_X\}
\]
\[
\mathcal{D}_{out} = \{d \in \mathcal{D} \mid d \subseteq R_{G,G'}\}
\]
\[
\mathcal{D}_{inout} = \{d \in \mathcal{D} \mid d \subseteq R_{G,G',L'}\}
\]

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 $\mathcal{T} \setminus X$.

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 $d_{proj}$ from the abstract domain $\mathcal{D}$, and then defining the sublattice $\mathcal{D}_i$ to be the closed interval $[\perp, d_{proj}]$, which consists of all the elements between $\perp$ and $d_{proj}$, according to the $\subseteq$ order (including $\perp$ and $d_{proj}$). Moreover, for every $i \in \{\text{in, out, inout, inoutloc}\}$, we define the projection operation $(\cdot|)_i$ as follows: $d_{i|} = d \cap d_{proj}$. Note that $d_{i|}$ is always in $\mathcal{D}_i$.

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

\[
d_{\text{proj}_{\text{in}}} = R_{G,G'} \cap d_{\text{proj}} \cap R_{G,G'} \cap d_{\text{proj}_{\text{inout}}} = R_{G,G'} \cap d_{\text{proj}_{\text{inout}}} = R_{G,G',L'}\]

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

Each abstract state in the sublattice $\mathcal{D}_{\text{out}}$ 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 $g_1^p$ and $g_2^p$ in the same equivalence class.

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

The sublattice $\mathcal{D}_{\text{inout}}$ 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-
The abstract meaning of procedure calls in the connection analysis is defined in Table 3. Again, we refer to the auxiliary constant \( \iota \) as the element that represents the identity relation between input and output, and \( C_{\text{body},p} \) is the body of procedure \( p \).

When a procedure is entered, local variables of the procedure and all the global variables \( g \) at the entry to the procedure are initialized to null. This is realized by applying the meet operation with auxiliary variable \( R_G \). Then, each of the \( g \) is initialized with the current variable value \( g' \) using \( \iota \). The \( \iota \) 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 \( (g', g) \) connection sets for all global variables \( g \). Intuitively, this stores the current value of variable \( g \) into \( g' \), 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: \( d_{\text{call}} \), which represents the partition at the call-site, and \( d_{\text{exit}} \), which represents the partition at the exit from the procedure. The meet operation of \( d_{\text{exit}} \) with \( R_G \) 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 \( g \) are used to implement a natural join operation for composing these relations. \( j_{\text{call}}(d_{\text{call}}) \) renames global variables from \( g' \) to \( g \) and \( j_{\text{exit}}(d_{\text{exit}}) \) renames global variables from \( g \) to \( g' \) to allow natural join. Intuitively, the old values \( g \) 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 \( g' \) 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 \( [p()]^{BU} \), defined in Table 3, instead of \( [p()]^{T} \). The two functions differ in the first argument they use when applying \( \text{return}^{BU} : [p()]^{BU} \) uses a constant value, which is the abstract state at the procedure exit computed when analyzing \( p() \) with \( \iota \). In contrast, \( [p()]^{T} \) 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:

\[
\forall d \in D. \ [p()]^{BU}(d) = [p()]^{T}(d)
\]

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 \( g \) and \( g' \) are always in the same connection set. This is a result of the definition of function \( \text{return} \), which projects the abstract element at the call-site into the sublattice \( D_{\text{out}} \) and the successive join with the \( \iota \) 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 \( d_{\text{out}} \) with \( \iota \), each old variable \( g \) 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 \( \iota \), that its connection sets are comprised of pairs of variables of the form \( \{g', g\} \). One important implication of this uniformity is that every entry abstract state \( d_{\text{entry}} \) to any procedure has a dual representation. In one representation, \( d \) is the join of \( \iota \) with some elements \( U' \subseteq D_{\text{out}} \). In the other representation, \( d \) is expressed as the join of \( \iota \) with some elements \( U_{\pi_0} \subseteq D_{\pi_0} \). In the following, we use the function \( o \) that replaces relationships among current variables by those among old ones: \( o(U' \subseteq U_{\pi_0}) = U_{\pi_0} \); and \( o(d) \) is the least upper bounds of \( \iota \) and elements \( U_{\pi_0} \) 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 \( U_{\pi_j} \) coming from \( D_{\pi_0} \), 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 \( \tau \) of intraprocedural transformers on an entry state of the form \( d_{\text{entry}} \subseteq D_{\pi_0} \cup U_{\pi_1} \cup U_{\pi_2} \) results in an element of the form \( d_{\text{entry}} \cup U_{\pi_0} \cup U_{\pi_1} \cup U_{\pi_2} \) where \( d_{\text{entry}} \) results from applying the transformers in \( \tau \) on \( d_{\text{entry}} \). Using the observation made in Sec. 5.5.1, this means that we can represent any abstract element \( d_{\text{entry}} \) resulting at a call-site as \( d = d_{\text{entry}} \cup d_{\text{call}} \), where \( d_{\text{call}} \) is the effect of \( \tau \) on \( d_{\text{entry}} \) and \( d_{\text{entry}} \subseteq D_{\pi_0} \) is a join of elements of the form \( U_{\pi_0} \subseteq D_{\pi_0} \):

\[
d = d_{\text{entry}} \cup U_{\pi_1} \cup \ldots \cup U_{\pi_n} \quad .
\]

#### 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 \( d_{\text{call}} \cup d_{\text{exit}} \), where \( d_{\text{call}} \subseteq D_{\pi_0} \) and it is a join of elements of form \( U_{\pi_0} \).

For each \( U_{\pi_0} \), the entry state resulting from analyzing \( p() \) when the calling context is \( d \) is either identical to the one resulting from \( d_{\text{call}} \) or can be obtained from \( d_{\text{call}} \) 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 \( z \) are in one of the connection sets of \( d_{\text{call}} \) and \( z' \) and \( z \) are in another. This means that the effect of
Say we want to compute the effect of invoking \( p() \) on abstract state \( d \) according to the top-down abstract semantics.

\[
[p()]^2(d) = [\text{return}]^2(((C_{\text{body}, p}^\sharp \circ [\text{entry}]^2)(d)), d)
\]

First, let’s compute the first argument to \([\text{return}]^2\).

\[
([C_{\text{body}, p}^\sharp \circ [\text{entry}]^2](d) = [C_{\text{body}, p}^\sharp([\text{entry}])^2](d)
\]

\[
= [C_{\text{body}, p}^\sharp([\text{entry}])^2]((d_1 \sqcup d_3 \sqcup d_4))
\]

\[
= [C_{\text{body}, p}^\sharp]((d_1 \sqcup d_3 \sqcup d_4)) \sqcup \text{tentry}
\]

\[
= [C_{\text{body}, p}^\sharp]((d_1 \sqcup d_3 \sqcup d_4)) \sqcup \text{tentry}
\]

\[
= [C_{\text{body}, p}^\sharp]((d_1 \sqcup d_3 \sqcup d_4)) \sqcup \text{tentry}
\]

\[
= [C_{\text{body}, p}^\sharp]((d_1 \sqcup d_3 \sqcup d_4)) \sqcup \text{tentry}
\]

\[
= [C_{\text{body}, p}^\sharp]((d_1 \sqcup d_3 \sqcup d_4)) \sqcup \text{tentry}
\]

\[
= [C_{\text{body}, p}^\sharp]((d_1 \sqcup d_3 \sqcup d_4)) \sqcup \text{tentry}
\]

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

When applying the return semantics, we first compute the natural join and then remove the temporary variables. Hence, we get

\[
\text{fcall}(d_1 \sqcup d_3 \sqcup d_4) \sqcup \text{fexit}([C_{\text{body}, p}^\sharp][((d_1 \sqcup d_3 \sqcup d_4)) \sqcup \text{tentry}])
\]

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

\[
\text{fcall}(d_1 \sqcup d_3 \sqcup d_4) \sqcup \text{fexit}([C_{\text{body}, p}^\sharp][((d_1 \sqcup d_3 \sqcup d_4)) \sqcup \text{tentry}]) = \text{fcall}(d_1 \sqcup d_3 \sqcup d_4) \sqcup \text{fexit}([C_{\text{body}, p}^\sharp][((d_1 \sqcup d_3 \sqcup d_4)) \sqcup \text{tentry}])
\]

\[
= \text{fcall}(d_1 \sqcup d_3 \sqcup d_4) \sqcup \text{fexit}([C_{\text{body}, p}^\sharp][((d_1 \sqcup d_3 \sqcup d_4)) \sqcup \text{tentry}])
\]

\[
= \text{fcall}(d_1 \sqcup d_3 \sqcup d_4) \sqcup \text{fexit}([C_{\text{body}, p}^\sharp][((d_1 \sqcup d_3 \sqcup d_4)) \sqcup \text{tentry}])
\]

\[
= \text{fcall}(d_1 \sqcup d_3 \sqcup d_4) \sqcup \text{fexit}([C_{\text{body}, p}^\sharp][((d_1 \sqcup d_3 \sqcup d_4)) \sqcup \text{tentry}])
\]

Thus, we get that

\[
\text{fcall}^2(d_1 \sqcup d_3 \sqcup d_4) \sqcup \text{fexit}([C_{\text{body}, p}^\sharp][((d_1 \sqcup d_3 \sqcup d_4)) \sqcup \text{tentry}])
\]

Moreover, \( f\text{call} \) is isomorphic and by Eq.5

\[
\text{fcall}(d_1 \sqcup d_3 \sqcup d_4) \sqcup \text{fexit}([C_{\text{body}, p}^\sharp][((d_1 \sqcup d_3 \sqcup d_4)) \sqcup \text{tentry}])
\]

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

\[
U_{\bar{x}_{\pi_1}, \bar{y}_n} \sqcup \ldots \sqcup U_{\bar{x}_{\pi_n}, \bar{y}_n}
\]

and that \( f\text{call} \) only replaces \( g' \) occurrences in \( d \); thus

\[
\text{fcall}(U_{\bar{x}_{\pi_1}, \bar{y}_n} \sqcup \ldots \sqcup U_{\bar{x}_{\pi_n}, \bar{y}_n}) = U_{\bar{x}_{\pi_1}, \bar{y}_n} \sqcup \ldots \sqcup U_{\bar{x}_{\pi_n}, \bar{y}_n}
\]

Finally, we get

\[
= \text{fcall}(d_1 \sqcup d_3 \sqcup d_4) \sqcup \text{fexit}([C_{\text{body}, p}^\sharp][((d_1 \sqcup d_3 \sqcup d_4)) \sqcup \text{tentry}])
\]

\[
= [p()]^2(d_1 \sqcup d_3 \sqcup d_4)
\]

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
The second equivalence is by Eq. 6, and the second equivalence is for any command
by changing the abstract transformer to always merge
conservatively modified the analysis to satisfy our requirements,
to null in all the executions leading to this statement. We therefore
x
y
and
are not merged if
depends on the abstract
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.

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/f (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
bottom-up analysis 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.

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 (boloat), 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).

\[ d = d_1 \cup d_3 \cup d_4, \text{ where } d_3, d_4 \in D_m \]

\[ [p]_o^2(d) = [\text{return}]_o^2([\text{C.body}]_o^2([\text{entry}]_o^2(d)), d) = [\text{return}]_o^2([\text{C.body}]_o^2(d_1 \cup d_3 \cup d_4), d) = [\text{return}]_o^2([\text{C.body}]_o^2([\text{entry}]_o^2(d_1 \cup o(d_1) \cup d_3) \cup [\text{entry}]_o^2(d_3)), d) \] (11)

The second equivalence is by Eq. 6, and the second equivalence is
because \( d_1 \cup d_3 \cup d_4 \in D_m \) and

\[ [\text{entry}]_o^2(d_1 \cup d_3) \cup o(d_3) = [\text{entry}]_o^2(d_1) \cup o(d_3) \in D_m \]

We showed that for any \( d = d_1 \cup d_3 \cup d_4 \), such that \( d_3, d_4 \in D_m \),

\[ [C]_o^2(d_1 \cup d_3 \cup d_4) = [\text{C}]_o^2(d_1) \cup d_3 \cup d_4 \]

for any command \( C \). Therefore, since \( o(d_3), o(d_1) \in D_m \),

\[ (11) = [\text{return}]_o^2([\text{C.body}]_o^2([\text{entry}]_o^2(d_1) \cup o(d_1) \cup o(d_3) \cup d_3) \]

By Eq. 5, \( d_1 \cup d_3 \cup d_4 = d_1 \cup d_3 \cup d_4 \). Thus, we can remove
\( o(d_3) \) because \( f_{\text{calc}}(o(d_3)) \) will be redundant in the natural join of
the \( \text{return} \) \( _o^2 \) operator. Using a similar reasoning, we can remove
\( f_{\text{exit}}(o(d_1)) \), since \( f_{\text{calc}}(d_1) \subseteq f_{\text{calc}}(d_1) \). Hence, finally,

\[ (11) = [\text{return}]_o^2([\text{C.body}]_o^2([\text{entry}]_o^2(d_1) \cup d_3 \cup d_4) = [p]_o^2([\text{BU}]_o^2(d)) \].

6. Experimental evaluation
In this section, we evaluate the effectiveness of our approach in
practice using the connection analysis for concreteness. We imple-
mented three versions of this analysis: the original top-down ver-
sion 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 \( x \) 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 \( \text{entry} \). 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 de-
scribed above using Chord [16] and applied them to five Java
benchmark programs whose characteristics are summarized in Ta-
ble 4. They include two programs (grande2 and grande3) from the
Java Grande benchmark suite and two (antlr and boloat) 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 evalu-
ating 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 coinci-
dence result, as well as confirmed in our experiments).
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.
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).

<table>
<thead>
<tr>
<th>description</th>
<th># classes</th>
<th># methods</th>
<th># bytecodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>app only</td>
<td>total</td>
<td>app only</td>
</tr>
<tr>
<td>grante2</td>
<td>17</td>
<td>61</td>
<td>112</td>
</tr>
<tr>
<td>grante3</td>
<td>42</td>
<td>241</td>
<td>231</td>
</tr>
<tr>
<td>antlr</td>
<td>116</td>
<td>358</td>
<td>1,167</td>
</tr>
<tr>
<td>weka</td>
<td>62</td>
<td>530</td>
<td>575</td>
</tr>
<tr>
<td>bloat</td>
<td>277</td>
<td>611</td>
<td>2,651</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th># of queries</th>
<th>summary computation time</th>
<th>summary instantiation time</th>
<th># of abstract states total</th>
<th>Top-Down analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>queries</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>time</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td># of abstract states total</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>queries</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>time</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td># of abstract states total</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>queries</td>
</tr>
</tbody>
</table>

7. Conclusions

We show using lattice theory that when an abstract domain has enough right-regular elements to allow function to be expressed as joins and meets with constant elements—and the elements used in the meet are right-regular—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

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 \( g \) which are initialized to 0. We also assume that the primitive commands \( a \in \text{PComm} \) are of the form

\[
\begin{align*}
x &:= c, \\
x &:= y, \\
x &:= \ast,
\end{align*}
\]

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 \( K_P \subseteq \mathcal{N} \) the finite set of constants which appear in a program \( P \). We assume \( K_P \) contains 0. We denote by \( G_P \subseteq \mathcal{G} \) 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 \( K = K_P \) the (fixed finite) set of constants that appear in \( P \), and by \( G = G_P \) 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 \( P \) denoted by \( \{ \} \) \[ \{ \} = \{ x := x \} \]

(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 \( s \in \mathcal{S} = G \rightarrow \mathcal{N} \) maps variables to their integer values. The meaning of primitive commands \( a \in \text{AComm} \) is standard, and defined below.

\[
\begin{align*}
[x := c](s) &= \{(s[x := \{ c \}])\}, \\
[x := y](s) &= \{(s[x := s(y)])\}, \\
[x := \ast](s) &= \{(s[x := n]) \mid n \in \mathcal{N}\}
\end{align*}
\]

Note that \([a] : \mathcal{S} \rightarrow 2^\mathcal{G}\) 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 \( s|_G : \mathcal{S} \rightarrow \mathcal{S} \) and \([\text{combine}] : \mathcal{S} \times \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{R} \), whose meaning is defined below:

\[
\begin{align*}
[s|_G](s_1, s_2, s_3) &= s, \\
[\text{combine}] (s_1, s_2, s_3) &= (s_1, s_3)
\end{align*}
\]

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 \( g \in \mathcal{G}, \bar{g} \) 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 \( \mathcal{G} \) of variables to compute the effect of procedure calls. We denote by \( v \in \mathcal{T} = \mathcal{G}' \cup \mathcal{T} \cup \mathcal{G} \) the set of all annotated variables which are ranged by a meta variable \( \gamma \).

We denote by \( \zeta \in \mathcal{G} \) \( = \mathcal{T} \cup \mathcal{K} \cup \{ \ast \} \) the set of abstract values ranged over by \( \zeta \). \( \mathcal{VAL} \) is comprised of annotated variables, constants which appear in the program, and the special value \( \ast \).

8.3.1 Abstract Domain

Let \( D_{\text{map}} \) be the set of all maps from variables \( v \in \mathcal{T} \) to \( 2^{\mathcal{VAL}} \)

\[
D_{\text{map}} = \mathcal{T} \mapsto 2^{\mathcal{VAL}}.
\]

We denote the set \( D_{\text{trans}} \subseteq D_{\text{map}} \) of transitively closed maps by

\[
D_{\text{trans}} = \{ [d] \mid d \in D_{\text{map}} \},
\]

where

\[
[d] = \lambda v \in \mathcal{T} \{ \{ v \} / v_0 = v \land 0 \leq i < n, v_{i+1} \in d(v_i) \}.
\]

Note that a map \( d \in D_{\text{map}} \) is transitively closed, i.e., \( d \in D_{\text{trans}} \), if and only if it associates \( v \) to a set containing \( v \), i.e., \( v \in d(v) \), and for any \( v' \in d(v) \) it holds that \( d(v') \subseteq d(v) \).

The abstract domain \( \mathcal{D} \) of the copy constant propagation analysis is an augmentation of \( D_{\text{trans}} \) with an explicit bottom element.

\[
\mathcal{D} = (\mathcal{D}_{\text{const}}, \sqcap, \sqcup, \sqsubseteq, \sqsupseteq, \sqsubset, \sqsupset, \sqcap, \sqcup, \sqsubseteq, \sqsupseteq, \sqsubset, \sqsupset), \text{ where}
\]

\[
\begin{align*}
D_{\text{const}} &= D_{\text{trans}} \cup \{ \bot \}, \\
d_1 \sqcap d_2 \Leftrightarrow d_1 = \bot \lor \forall v \in \mathcal{T}, d_1(v) \subseteq d_2(v) \\
\top &= \lambda v \in \mathcal{T} \mathcal{VAL} \\
d_1 \sqcup d_2 &= \begin{cases} 
{d_1} & \text{if } d_1 = \bot \\
{d_2} & \text{if } d_2 = \bot \\
{d_1 \sqcup d_2} & \text{otherwise}
\end{cases} \\
d_1 \sqcap d_2 &= \begin{cases} 
\bot & \text{if } d_1 = \bot \lor d_2 = \bot \\
{d_1 \sqcap d_2} & \text{otherwise}
\end{cases}
\end{align*}
\]

8.3.2 Abstract Intraprocedural Transformers

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

\[
\begin{align*}
[x := c]^2 &= (\lambda d. d \sqcap S_x) \sqcup U_{x'}c, \\
[x := y]^2 &= (\lambda d. d \sqcap S_y) \sqcup U_{x'y'}, \\
[x := \ast]^2 &= (\lambda d. d \sqcap S_x) \sqcup U_{x'x'}
\end{align*}
\]

where

\[
\begin{align*}
S_x(v) &= \lambda v \in \mathcal{T} \{ \{ v \} \} \text{ if } v = x' \\
U_{x'} &= \lambda v \in \mathcal{T} \{ \{ v \} \} \text{ if } v \neq x'
\end{align*}
\]

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 $\sqsubseteq$ 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

$$(d \sqcap d_s) \sqcup (d \sqcap d_s) \subseteq (d' \sqcup d) \sqcap d_s.$$  

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

$$d' \sqcup (d \sqcap d_s) \subseteq (d' \sqcup d) \sqcap d_s.$$  

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

**Induction Step.** Assume that the induction assumption holds for sets $X$ such that $|X| = n$. Let $\nu \in X$ be an arbitrary element. Notice that by its definition

$$S_{\nu'} = S_{\nu' \setminus \nu} \cap S_{\nu}.$$  

Therefore,

$$(d' \sqcup d) \sqcap S_{\nu} = (d' \sqcup d) \sqcap (S_{\nu' \setminus \nu} \cap S_{\nu})$$

$$(d' \sqcup d) \sqcap S_{\nu} = (d' \sqcup d) \sqcap (S_{\nu' \setminus \nu} \cap S_{\nu})$$

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

$$(s,s') \in \gamma(d) \iff \left( \forall \tau \in \tau, (s(\tau) \in d(\tau) \cap K) \lor (s' \in d(\tau)) \right) \land \left( \forall \tau' \in \tau', s'(\tau') \in \left(d(\tau) \cap K \lor s(y) \mid y \in d(\tau') \right) \lor \left(s' \in d(\tau') \right) \right).$$

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 $s \in d(\overline{g})$ 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 $s \in d(\overline{g'})$.

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

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

$$f = \lambda d.((d \sqcap d) \sqcup d_b).$$

where $d_b \in S_{\nu'}$ for some $\nu \in \nu'$.  

8.4 Soundness of the Top Down Analysis

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

$$(s,s') \in \gamma(d) \iff \left( \forall \tau \in \tau, (s(\tau) \in d(\tau) \cap K) \lor (s' \in d(\tau)) \right) \land \left( \forall \tau' \in \tau', s'(\tau') \in \left(d(\tau) \cap K \lor s(y) \mid y \in d(\tau') \right) \lor (s' \in d(\tau')) \right).$$

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 $s \in d(\overline{g})$ 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 $s \in d(\overline{g'})$.

**Lemmata 12.** The abstract transformers pertaining to intraprocedural primitive commands, $\nu$, 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 \( \texttt{x} := \texttt{y} + \texttt{y} := 3 \) on an abstract state \( d \), in which \( 3 \not\in d(\texttt{y}) \). Applying the abstract transformer \([x := y] \) to \( d \) results in an abstract element \( d' \), where \( y' \in d(x') \). Applying \([y := 3] \) to \( d' \) results in an abstract state \( d'' \), where \( 3 \in d''(\texttt{y}) \). Perhaps surprisingly, in the abstract state \( d'' = d' \cup d'' \), which conservatively represent the possible states after the non-deterministic choice (+), we get that \( 3 \in d''(\texttt{x}) \). This is sound, but imprecise. The reason for the imprecision is that our domain includes only transitively closed maps and having \( y' \notin d(\texttt{x}) \) 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: \( \texttt{t} := \texttt{y} ; \quad \texttt{y} := 0 ; \quad \texttt{x} := \texttt{t} ; \quad \texttt{x} := \texttt{t} ; \quad \texttt{t} := 0 \), and apply this sequence to \( d \). In the abstract state \( d \) arising just before \( t \) is assigned \( 0 \), we get that \( \texttt{t}' \in d(\texttt{x}') \) and \( \texttt{t}' \in d(\texttt{y}') \) but \( \texttt{y}' \notin d(\texttt{x}') \) and \( \texttt{x}' \notin d(\texttt{y}') \). 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.** \( D \) is a triad domain.

**Projection Elements**

**Proof** We define the projection elements

\[
\begin{align*}
\text{proj}_{\text{in}} &= R_G \quad \text{proj}_{\text{out}} = \text{proj}_{\text{out}} = R_G
\end{align*}
\]

By Lem. 9, \( \text{proj}_{\text{in}}, \text{proj}_{\text{out}} \) and \( \text{proj}_{\text{out}} \) are right-modular.

**Isomorphism functions** We define the renaming functions

\[
\begin{align*}
\text{call}^\uparrow(x) &= \begin{cases}
\text{null} & \text{if} \quad \lambda v \in \mathcal{T}.
\end{cases}
\end{align*}
\]

Let \( \text{call}^\uparrow(x) \), \( \text{exit}^\uparrow(x) \), \( \text{inout}^\uparrow(x) \) be the renaming function induced on \( \text{call}^\uparrow(x) \) and finally let \( \text{call}^\uparrow(x) \), \( \text{exit}^\uparrow(x) \), \( \text{inout}^\uparrow(x) \) be the renaming functions induced on \( D \),

\[
\begin{align*}
d(d) = \begin{cases}
\text{null} & \text{if} \quad \lambda v \in \mathcal{T}. d = \text{null}
\end{cases}
\end{align*}
\]

where \( i \in \{\text{call}, \text{inout}, \text{exit}\} \).

**Claim 14.**

\[
\begin{align*}
\text{call}^\uparrow(R_G) &= R_G, & \text{call}^\uparrow(R_{G'}) &= R_{G'}, & \text{call}^\uparrow(R_{G''}) &= R_G \quad &\text{The other cases are symmetric.}
\end{align*}
\]

**Proof** We prove the claim on \( \text{call} \) and \( R_G \). The other cases are symmetric.

\[
\begin{align*}
\text{call}^\uparrow(R_G) &= \text{null} \quad &\text{call}^\uparrow(R_{G''}) &= \text{null}
\end{align*}
\]

\[
\begin{align*}
\text{out}(\text{call}^\uparrow) &= \begin{cases}
\text{null} & \text{if} \quad \lambda v \in \mathcal{T}. d(\text{call}^\uparrow(\text{null})) \quad &\text{otherwise (12)}
\end{cases}
\end{align*}
\]

where \( i \in \{\text{call}, \text{inout}, \text{exit}\} \).

**Claim 15.** For all \( d \in D \),

\[
\begin{align*}
\text{out}(\text{call}^\uparrow(d)) &= \text{null} = \text{call}^\uparrow(d)
\end{align*}
\]

**Proof** Let \( d \in D \) and let \( v \in \mathcal{T} \). If \( d = \text{null} \) then

\[
\begin{align*}
\text{call}^\uparrow(d) &= \text{null} = \text{call}^\uparrow(d)
\end{align*}
\]

Otherwise, by Eq. 12,

\[
\begin{align*}
\text{call}^\uparrow(d)(v) &= \text{call}^\uparrow(d)(\text{call}^\uparrow(\text{null})(v))
\end{align*}
\]

and

\[
\begin{align*}
\text{out}(\text{call}^\uparrow(d))(v) &= \text{null} = \text{call}^\uparrow(d) \quad &\text{null} = \text{call}^\uparrow(d)
\end{align*}
\]

Therefore,

\[
\begin{align*}
\text{call}^\uparrow(d)(v) &= \text{null}
\end{align*}
\]

and

\[
\begin{align*}
\text{out}(\text{call}^\uparrow(d))(v) &= \text{null}
\end{align*}
\]

Hence,

\[
\begin{align*}
\text{call}^\uparrow(d)(v) &= \text{null}
\end{align*}
\]

and

\[
\begin{align*}
\text{out}(\text{call}^\uparrow(d))(v) &= \text{null}
\end{align*}
\]

Otherwise, if \( v \not\in G \), by the definition of the renaming functions

\[
\begin{align*}
\text{call}^\uparrow(d)(v) &= (\text{call}^\uparrow(d) \circ \text{call}^\uparrow(d))(v)
\end{align*}
\]

and by the definition of \( D \),

\[
\begin{align*}
\text{out}(\text{call}^\uparrow(d))(v) &= \text{null}
\end{align*}
\]

and therefore again by the definition of the renaming functions

\[
\begin{align*}
\text{call}^\uparrow(d)(v) &= \text{call}^\uparrow(d) \circ \text{call}^\uparrow(d)(v)
\end{align*}
\]

**Claim 16.** For all \( d \in D \),

\[
\begin{align*}
\text{call}^\uparrow(d) &= d
\end{align*}
\]

**Proof** By the definition of \( D \) and of \( \text{call} \).

\[
\begin{align*}
\text{call}^\uparrow(d) &= d
\end{align*}
\]
We define
\[ t_{\text{entry}} = \{ v' \mapsto \{ v', \bar{v} \} \mid v \in \mathcal{G} \cup \{ \bar{v} \mapsto \{ v \} \mid v \in \mathcal{G} \cup \bar{v} \mapsto \{ \bar{v} \} \mid v \in \mathcal{G} \} \]  

Claim 17. For every \( d \in \mathcal{D}_{\text{out}} \), \( d \sqcup t_{\text{entry}} = \text{finout}(d) \sqcup t_{\text{entry}} \).

Proof
\[
\begin{align*}
d \sqcup t_{\text{entry}} &= [d \cup t_{\text{entry}}] \\
&= \lambda \nu \in \Upsilon. \begin{cases}
d(g') \cup \{ g', \bar{g} \} & v = g' \in \mathcal{G}' \\
\{ \bar{g} \} \cup \{ g', \bar{g} \} & v = \bar{g} \in \mathcal{G'} \\
\{ \bar{g} \} & v = \bar{g} \in \mathcal{G'} \\
\end{cases} \\
&= \lambda \nu \in \Upsilon. \begin{cases}
\{ h, h' \mid h', h' \in d(g') \} & v = g' \in \mathcal{G'} \lor v = \bar{g} \in \mathcal{G'} \\
\{ \bar{g} \} & v = \bar{g} \in \mathcal{G} \\
\end{cases} \\
&= \lambda \nu \in \Upsilon. \begin{cases}
\{ g' \} \cup \{ g', \bar{g} \} & v = g' \in \mathcal{G'} \\
\{ \bar{g} \} \cup \{ g', \bar{g} \} & v = \bar{g} \in \mathcal{G} \\
\{ g' \} \cup \{ g', \bar{g} \} & v = g' \in \mathcal{G'} \\
\{ \bar{g} \} \cup \{ g', \bar{g} \} & v = \bar{g} \in \mathcal{G} \\
\end{cases} \\
&= \lambda \nu \in \Upsilon. \begin{cases}
\{ g' \} \cup \{ g', \bar{g} \} & v = g' \in \mathcal{G'} \\
\text{finout}(d)(\bar{g}) \cup \{ g', \bar{g} \} & v = \bar{g} \in \mathcal{G'} \\
\{ \bar{g} \} \cup \{ g' \} & v = \bar{g} \in \mathcal{G} \\
\end{cases} \\
&= \text{finout}(d) \sqcup t_{\text{entry}} \\
\end{align*}
\]