Upper Adjoints for Fast Inter-procedural Variable Equalities

Markus Müller-Olm[1] and Helmut Seidl[2]

[1] Westf. Wilhelms-Universität Münster, Mathematik und Informatik, 48149 Münster, Germany
mmo@math.uni-muenster.de
[2] TU München, Informatik, I2, 85748 Garching, Germany
seidl@in.tum.de

Abstract. We present a polynomial-time algorithm which at the extra cost of a factor $\mathcal{O}(k)$ (k the number of variables) generalizes inter-procedural copy constant propagation. Our algorithm infers variable-variable equalities in addition to equalities between variables and constants. Like copy constant propagation, it tracks constant and copying assignments but abstracts more complex assignments and guards. The algorithm is based on the observation that, for the abstract lattice of consistent equivalence relations, the upper adjoints of summary functions can be represented much more succinctly than summary functions themselves.

1 Introduction

The key task when realizing inter-procedural analyses along the lines of the functional approach of Sharir/Pnueli [13, 8], is to determine the *summary functions* for procedures which describe their effects on the abstract program state before the call. Given a complete lattice $\mathbb{D}$ for the abstract program states, the summary functions are taken from the set of monotonic or (if we are lucky) distributive functions $\mathbb{D} \to \mathbb{D}$. This set is often large (if not infinite), rendering it a nontrivial task to identify a representation for summary functions which is efficiently supports basic operations such as function composition or function application to values of $\mathbb{D}$. Examples for such efficient representations are pairs of sets in case of gen/kill bit-vector problems [6] or vector spaces of matrices in case of affine equality analyses [11].

In this paper we present one further analysis where efficient representations of summary functions exist, namely, the analysis of variable-variable together with variable-constant equalities. This analysis is a generalization of copy constant propagation [6]. Based on the new analysis, register allocation can be enhanced to additionally remove certain register-register assignments. The idea is to allow the allocator to assign variables $\mathbf{x}$ and $\mathbf{y}$ to the same register, given that $\mathbf{x} = \mathbf{y}$ holds at each program point where both variables are live. This technique is known as *register coalescing* [3].

Example 1. Consider the program from Fig. 1. In this program, the variables $\mathbf{x}_2$ and $\mathbf{x}_3$ are both live at program point 3. Since $\mathbf{x}_2 = \mathbf{x}_3$ definitively holds at this program point, we can coalesce $\mathbf{x}_2, \mathbf{x}_3$ into a variable $\mathbf{y}$. By doing so, the assignment $\mathbf{x}_3 \leftarrow \mathbf{x}_2$ becomes $\mathbf{y} \leftarrow \mathbf{y}$ and thus can be removed. □

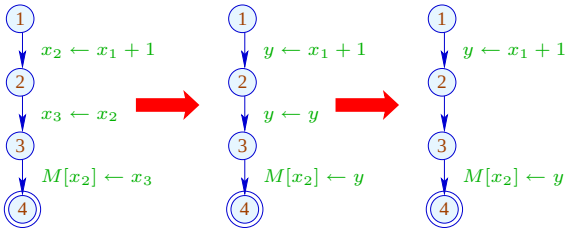

Fig. 1. A program with variable-variable assignments.

The summary functions for inter-procedural equality analysis are completely distributive, i.e., commute with arbitrary least upper bounds. Also, the complete lattice of abstract values at program points is $\sqcup$-*atomic* (for a precise definition see below). Therefore, summary functions can (at least in principle) be represented through tabulation of their values for $\sqcup$-atoms. The number of these atoms, though, is exponential in the number of program variables — rendering this immediate idea as not practical.

In this paper, we report that summary functions for equality analysis can nonetheless be succinctly represented. The key idea is not to represent summary functions themselves, but their *upper adjoints* — a well-known construction from the theory of Galois connections which, for a completely $\sqcup$-distributive function, returns a completely $\sqcap$-distributive function. This construction has also been used for *demand-driven* program analyses [5, 7]. It provides the solution in our application since the lattice in question has quadratically many $\sqcap$-atomic elements only, thus allowing for an efficient tabulation of upper adjoints. As a result, we obtain a fast inter-procedural equality analysis whose worst-case complexity is only one factor k (k the number of variables) slower than the fastest known algorithm for inferring copy constants [6].

Related work. Equality of variables can be considered as a particular case of a generalized analysis of availability of expressions, also called *value numbering* [1]. Originally, this analysis tracks for basic blocks the symbolic expressions representing the values of the variables assigned to. The key point is that operator symbols are left uninterpreted. The inferred equalities between variables and terms therefore are *Herbrand* equalities. Later, the idea of inferring Herbrand equalities was generalized to arbitrary control-flow graphs [14]. Only recently, this problem has attracted fresh attention. In [4], Gulwani and Necula show that the original algorithm of Steffen, Knoop and Rüthing can be turned into a polynomial time algorithm if one is interested in polynomially sized equalities between variables and terms only. Progress in a different direction was made in [10] and [12] where the authors generalize Herbrand equalities to deal with negative guards or side-effect free functions, respectively. Still, it is open whether full inter-procedural analysis of Herbrand equalities is possible.

On the other hand, when only assignments of variables and constants are tracked, the abstract domain can be chosen *finite* – thus implying computability of the analysis. The naive approach, however, results in an exponential-time algorithm. A less naive approach may interpret (or code) the constants as *numbers*. The problem then consists

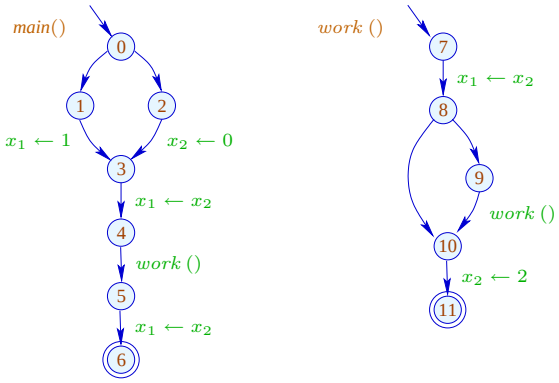

Fig. 2. An example program with procedures.

in inferring specific *affine* equalities between variables. The latter problem is known to be inter-procedurally decidable in polynomial time (given that each required arithmetic operation counts for $\mathcal{O}(1)$). The algorithm for this problem as presented in [11], however, has a factor k^8 in the worst-case complexity bound (k the number of variables). In the present paper we improve on this by reducing the exponent to 4 in the worst case — where sparse representations could be even more efficient.

This paper is organized as follows. After defining programs and their collecting semantics in Section 2, we introduce in Section 3 the complete lattice of consistent equivalence relations that is central for our approach. We discuss basic operations on this lattice and their complexity. Section 4 is concerned with representing summary functions as needed in our inter-procedural analysis. It turns out that in our scenario it is advantageous to represent summary functions not directly but by their upper adjoints. We then present in Section 5 our inter-procedural analysis. We extend our approach to local variables in Section 6. Section 7 summarizes and concludes.

2 Programs and Their Collecting Semantics

For this paper, we view programs as a finite collection of procedures f where each f is given by a finite control-flow graph as in Fig. 2. Each edge in the control-flow graphs is either labeled with a call $f()$ to a procedure f or with an assignment s. In pictures, we omit the label if it represents a skip operation $\mathbf{x}_i \leftarrow \mathbf{x}_i$. Let V denote the set of values the program uses in its computations. For technical reasons, we assume $|V| \geq 2$. Let $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ be the set of global program variables. Later, we extend our approach to deal with local variables as well. In order to concentrate on the essentials of the analysis, we consider simplified programs only. So, we assume that conditional branching has already been abstracted to non-deterministic branching. Since our analysis only tracks variable-to-variable assignments and constant-to-variable assignments, we consider assignments $\mathbf{x}_i \leftarrow \mathbf{x}_j$, $\mathbf{x}_i \leftarrow c$ or $\mathbf{x}_i \leftarrow ?$ for variables $\mathbf{x}_i, \mathbf{x}_j \in \mathbf{X}$ and constants $c \in V$. Here the *non-deterministic* assignment $\mathbf{x}_i \leftarrow ?$ may assign *any* value

to $\mathbf{x}_i$. This is used to abstract read operations which change the value of $\mathbf{x}_i$ in an unpredictable way or assignments of expressions that are not just a variable or constant. Note that the same class of programs is considered for *copy constant propagation* [6].

The *collecting semantics* of a program assigns to each program point v the set $\mathcal{C}[u]$ of program states which occur at u in an execution of the program. In our application, the variables $\mathbf{x}_1, \ldots, \mathbf{x}_k$ take values from the set V. Accordingly, an individual program state can be represented by a vector $(x_1, \ldots, x_k) \in V^k$ where x_i denotes the value of variable $\mathbf{x}_i$, and $\mathcal{C}[u]$ is a subset of V^k. The definition of the collecting semantics of a program is based on a specification of the effects $[\![s]\!]$ of assignments s onto the set of states in which s is to be executed. The effects of assignments are given by:

$$\begin{array}{ll} [\![\mathbf{x}_i \leftarrow ?]\!]\, Y & = \{x' \mid \exists\, x \in Y :\ \forall\, k \neq i :\ x'_k = x_k\} \\ [\![\mathbf{x}_i \leftarrow \mathbf{x}_j]\!]\, Y & = \{x' \mid \exists\, x \in Y :\ x'_i = x_j \wedge \forall\, k \neq i :\ x'_k = x_k\} \\ [\![\mathbf{x}_i \leftarrow c]\!]\, Y & = \{x' \mid \exists\, x \in Y :\ x'_i = c \wedge \forall\, k \neq i :\ x'_k = x_k\} \end{array}$$

A procedure f induces a transformation of the set of possible program states before the call to the set of program states that occur after the call if the procedure is called in any of these states. Here, we choose to collect this transformation *from the rear* and consider for each program point u of a procedure f, the transformation $\mathcal{S}[u] : 2^{V^k} \to 2^{V^k}$ induced by same-level program executions starting from u and reaching the procedure exit of f at the same level. Then, the transformation for procedure f is given by $\mathcal{S}[\mathsf{st}_f]$, where st_f is the entry point of f. The transformations $\mathcal{S}[u]$ are characterized as the least solution of the following system of in-equations:

$$\begin{array}{lll} \mathcal{S}[\mathsf{rt}_f] \supseteq \mathsf{Id} & & \mathsf{rt}_f \text{ exit point of procedure } f \\ \mathcal{S}[u] \supseteq \mathcal{S}[v] \circ \mathcal{S}[\mathsf{st}_f] & & (u, f(), v) \text{ a call edge, } \mathsf{st}_f \text{ entry point of } f \\ \mathcal{S}[u] \supseteq \mathcal{S}[v] \circ [\![s]\!] & & (u, s, v) \text{ an assignment edge} \end{array}$$

where $\mathsf{Id}\ X = X$ for every set of program states X and "$\supseteq$" is the pointwise extension of the superset relation to set-valued functions. Since the expressions on right-hand sides of all in-equations denote monotonic functions, the system of in-equations has a unique least solution by the Knaster-Tarski fixpoint theorem.

Assume that we are given the effects $\mathcal{S}[\mathsf{st}_f]$ of calls to the procedures f. Then these can be used to determine, for every program point u, the set of program states $\mathcal{C}[u] \subseteq V^k$ which possibly are attained when reaching u. These can be determined as the least solution of the following system of in-equations:

$$\begin{array}{lll} \mathcal{C}[\mathsf{st}_{main}] \supseteq V^k & & \\ \mathcal{C}[\mathsf{st}_f] \supseteq \mathcal{C}[u] & & (u, f(), \_) \text{ a call edge} \\ \mathcal{C}[v] \supseteq \mathcal{S}[\mathsf{st}_f]\,(\mathcal{C}[u]) & & (u, f(), v) \text{ a call edge} \\ \mathcal{C}[v] \supseteq [\![s]\!]\,(\mathcal{C}[u]) & & k = (u, s, v) \text{ an assignment edge}. \end{array}$$

3 The Abstract Domain

We are interested in equalities between variables and variables and between variables and constants. In order to express such properties, we introduce the complete lattice

$\mathbb{E}(\mathbf{X}, V)$ (or $\mathbb{E}$ for short). Its least element $\bot$ describes the empty set of program states and represents that all conceivable equalities are valid. Thus, $\alpha(\emptyset) = \bot$ where $\alpha : 2^{V^k} \to \mathbb{E}$ is the function which maps sets of program states to their respective best description in $\mathbb{E}$. Every element $E \neq \bot$ in the lattice is an equivalence relation on $\mathbf{X} \cup V$ where constants are considered as pairwise distinct, i.e., non-equivalent. Let us call such an equivalence relation *consistent*. The consistent equivalence relation describing a set $\emptyset \neq X \subseteq 2^{V^k}$, is given by $\alpha(X)$ where $(\mathbf{x}_i, \mathbf{x}_j) \in \alpha(X)$ iff $x_i = x_j$ for all $(x_1, \ldots, x_k) \in X$ and $(\mathbf{x}_i, c) \in \alpha(X)$ for $c \in V$ iff $x_i = c$ for all $(x_1, \ldots, x_k) \in X$. Technically, we can represent a consistent equivalence relation E as an array (for simplicity also denoted by E) where $E[i] = c$ iff the equivalence class of $\mathbf{x}_i$ contains the constant $c \in V$ and $E[i] = \mathbf{x}_j$ for one representative variable $\mathbf{x}_j$ from the equivalence class of $\mathbf{x}_i$ if this class does not contain a constant. Then, two variables $\mathbf{x}_i$ and $\mathbf{x}_j$ belong to the same equivalence class iff $E[i] = E[j]$ and a variable $\mathbf{x}_i$ and a constant c belong to the same class iff $E[i] = c$. Logically, we can represent E by the conjunction of equalities $\mathbf{x}_i = E[i]$ for those $\mathbf{x}_i$ with $E[i] \neq \mathbf{x}_i$, i.e., by a conjunction of at most k equalities of the form $\mathbf{x}_i = \mathbf{x}_j$ or $\mathbf{x}_i = c$ for distinct variables $\mathbf{x}_i, \mathbf{x}_j$ and constants c.

On the set $\mathbb{E}$, we define an ordering $\sqsubseteq$ as implication, i.e., $E_1 \sqsubseteq E_2$ iff either $E_1 = \bot$ or E_1 is a consistent equivalence relation where every equality of E_2 is implied by the conjunction of equalities of E_1. Thus, the least upper bound of two consistent equivalence relations E_1, E_2 is the equivalence relation which is represented by the conjunction of all equalities implied by E_1 as well as by E_2. The greatest lower bound of two equivalence relations logically is given by their conjunction.

Not every two consistent equivalence relations have an other consistent equivalence relation as greatest lower bound. The conjunction of $\mathbf{x}_1 = 1$ and $\mathbf{x}_1 = 2$, e.g., equates the distinct constants 1 and 2. The greatest lower bound therefore is given by $\bot$ which thus logically denotes false. Note that the length h of a strictly increasing sequence:

$$\bot \sqsubset E_1 \sqsubset \ldots \sqsubset E_h$$

in $\mathbb{E}$ is bounded by $h \leq k + 1$ where k is the number of program variables.

Lemma 1. *1. The least upper bound $E_1 \sqcup E_2$ can be computed in time $\mathcal{O}(k)$.*
2. The greatest lower bound $E_1 \sqcap \ldots \sqcap E_n$ of n equivalence relations can be computed in time $\mathcal{O}((n + k) \cdot k)$.

Proof. W.l.o.g. assume that all E_i are different from $\bot$. The first assertion follows since we can determine, in linear time, for each variable $\mathbf{x}_i$, the pair $(E_1[i], E_2[i])$ giving us the pair of equivalence classes w.r.t. E_1 and E_2, respectively, to which $\mathbf{x}_i$ belongs. Then using bucket sort, the equivalence classes of $E = E_1 \sqcup E_2$ can be computed in time $\mathcal{O}(k)$. Let X denote a maximal set of variables all mapped to the same pair (t_1, t_2). If $t_1 = t_2 = c$ for a constant c, then $E[j] = c$ for all $\mathbf{x}_j \in X$. Otherwise, all variables in X are equal, but have unknown value. Therefore, we set, for each $\mathbf{x}_j \in X$, $E[j] = \mathbf{x}_i$ for some (e.g., the first) variable $\mathbf{x}_i \in X$.

An algorithm establishing the complexity bound of the second assertion works as follows. We start with one of the given equivalence relations $E = E_1$ and then successively add the at most $(n - 1) \cdot k$ equalities to represent the remaining equivalence

$$
\begin{array}{lll}
(\mathbf{x}_i = c) \wedge E & = & \textbf{if}\ (E[i] = c' \in V)\ \ \textbf{if}\ (c = c')\ \ \textbf{return}\ \ E;\ \ \textbf{else}\ \ \textbf{return}\ \ \bot; \\
& & \quad \textbf{else}\ \ \{ \quad // \quad E[i] \text{ is a variable} \\
& & \qquad X \leftarrow \{\mathbf{x}_j \mid E[j] = E[i]\}; \\
& & \qquad \textbf{forall}\ (\mathbf{x}_j \in X)\ \ E[j] \leftarrow c; \\
& & \qquad \textbf{return}\ \ E; \\
& & \quad \} \\
(\mathbf{x}_i = \mathbf{x}_j) \wedge E & = & \textbf{if}\ (E[i] = E[j])\ \ \textbf{return}\ \ E; \\
& & \quad \textbf{else if}\ (E[i] = c \in V)\ \ \textbf{if}\ (E[j] = c' \in V)\ \ \textbf{return}\ \ \bot; \\
& & \qquad\qquad\qquad \textbf{else}\ \ \{ \quad // \quad E[j] \text{ is a variable} \\
& & \qquad\qquad\qquad\qquad X \leftarrow \{\mathbf{x}_{j'} \mid E[j'] = E[j]\}; \\
& & \qquad\qquad\qquad\qquad \textbf{forall}\ (\mathbf{x}_{j'} \in X)\ \ E[j'] \leftarrow c; \\
& & \qquad\qquad\qquad\qquad \textbf{return}\ \ E; \\
& & \qquad\qquad\qquad \} \\
& & \quad\ \ \textbf{else}\ \ \{ \quad // \quad E[i] \text{ is a variable} \\
& & \qquad\qquad X \leftarrow \{\mathbf{x}_{i'} \mid E[i'] = E[i]\}; \\
& & \qquad\qquad \textbf{forall}\ (\mathbf{x}_{i'} \in X)\ \ E[i'] \leftarrow E[j]; \\
& & \qquad\qquad \textbf{return}\ \ E; \\
& & \qquad \}
\end{array}
$$

Fig. 3. The Implementation of conjunctions $\mathbf{x}_i = t \wedge E$ for $E \neq \bot$.

relations. An algorithm for computing $(\mathbf{x}_i = t) \wedge E$ for (an array representation of) a consistent equivalence relation $E \neq \bot$ is presented in Fig. 3.

Every test in this algorithm takes time $\mathcal{O}(1)$. If some update of E occurs, then either an equivalence class receives a constant value or two equivalence classes are merged. Both events can only occur $\mathcal{O}(k)$ times. Since each update can be executed in time $\mathcal{O}(k)$, the complexity estimation for the greatest lower bound computation follows. $\square$

The mapping α is completely distributive, i.e., it commutes with arbitrary least upper bounds. Thus, it is the lower adjoint (*abstraction*) of a *Galois*-connection [9]. The counterpart to α, the *concretization* $\gamma : \mathbb{E} \to 2^{V^k}$ is given by $\gamma(E) = \{x \in V^k \mid x \models E\}$ for $E \neq \bot$ and $\gamma(\bot) = \emptyset$. Here, we write $x \models E$ for a vector x satisfying the equivalence relation E. For every assignment s, we define the abstract effect $[\![s]\!]^\sharp$ by:

$$
\begin{array}{l}
[\![\mathbf{x}_i \leftarrow ?]\!]^\sharp\, E = \exists^\sharp \mathbf{x}_i.\, E \\
[\![\mathbf{x}_i \leftarrow t]\!]^\sharp\, E = \begin{cases} (\mathbf{x}_i = t) \wedge \exists^\sharp \mathbf{x}_i.\, E, & \text{if } t \neq \mathbf{x}_i \\ E, & \text{if } t = \mathbf{x}_i \end{cases}
\end{array}
$$

for every $t \in \mathbf{X} \cup V$. Here, the abstract existential quantification $\exists^\sharp \mathbf{x}_i.\, E'$ is defined as $\bot$ if $E' = \bot$ and otherwise as the conjunction of all equalities implied by E' which do not contain $\mathbf{x}_i$. We note that $E' \mapsto \exists^\sharp \mathbf{x}_i.\, E'$ is completely distributive, i.e., commutes with arbitrary least upper bounds. Therefore, all abstract transformers $[\![s]\!]^\sharp$ are completely distributive. An implementation of the transformer $[\![\mathbf{x}_i \leftarrow ?]\!]^\sharp$ for consistent equivalence relations $E \neq \bot$ is provided in Fig. 4. According to this implementation, the result of $[\![\mathbf{x}_i \leftarrow ?]\!]^\sharp\, E$, i.e., $\exists^\sharp \mathbf{x}_i.\, E$ can be computed in time $\mathcal{O}(k)$. Therefore, all abstract transformers $[\![s]\!]^\sharp$ can be evaluated in linear time. The result of the analysis of the program from Fig. 1 is shown in Fig. 5. Note that we have listed only the non-trivial

$$
\begin{array}{l}
[\![\mathbf{x}_i \leftarrow ?]\!]^\sharp\, E = \mathbf{if}\,(E[i] = c \in V)\ \ \{E[i] \leftarrow \mathbf{x}_i;\ \mathbf{return}\ \ E;\} \\
\qquad\qquad\qquad \mathbf{else}\ \ \{\ X \leftarrow \{\mathbf{x}_j \mid j \neq i, E[j] = E[i]\}; \\
\qquad\qquad\qquad\qquad E[i] \leftarrow \mathbf{x}_i; \\
\qquad\qquad\qquad\qquad \mathbf{if}\ (X \neq \emptyset)\ \ \{\mathbf{choose}\ \mathbf{x}_j \in X;\ \ \mathbf{forall}\ (\mathbf{x}_{j'} \in X)\ \ E[j'] \leftarrow \mathbf{x}_j;\} \\
\qquad\qquad\qquad\qquad \mathbf{return}\ \ E; \\
\qquad\qquad\qquad \}
\end{array}
$$

Fig. 4. The Implementation of the transformer $[\![\mathbf{x}_i \leftarrow ?]\!]^\sharp$ for $E \neq \bot$.

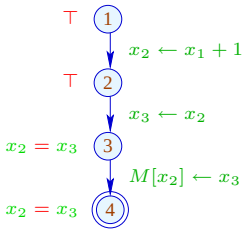

Fig. 5. The equalities between the variables in the program of Fig. 1.

equalities. Also, the information is propagated in forward direction through the control-flow graph where at program start only trivial equalities between variables are assumed, i.e., $E_0[i] = \mathbf{x}_i$ for all i.

4 Representing Summary Functions

Let $g : \mathbb{E} \to \mathbb{E}$ denote a completely distributive function. As the lattice $\mathbb{E}$ is quite large, a direct representation of g, e.g., through a value table is not practical. Surprisingly, this is different for the *upper adjoint* g^- of g. For an arbitrary complete lattice $\mathbb{D}$ and $g : \mathbb{D} \to \mathbb{D}$, the function $g^- : \mathbb{D} \to \mathbb{D}$ is defined by:

$$g^-(d) = \bigsqcup\{d' \in \mathbb{D} \mid g(d') \sqsubseteq d\}$$

It is well-known from lattice theory that for completely distributive g, the pair of functions (g, g^-) forms a *Galois connection*. Thus, $g^- \circ g \sqsupseteq \mathsf{Id}$ and $g \circ g^- \sqsubseteq \mathsf{Id}$. In particular, the upper adjoint g^- is completely distributive as well – however, for the reverse ordering (see, e.g., [9]), i.e., for every $X \subseteq \mathbb{D}$, $g^-(\bigsqcap X) = \bigsqcap\{g^-(d) \mid d \in X\}$. For a distinction, we call g^- completely $\sqcap$-distributive. For completely distributive g, the function g^- is just another representation of the function g itself. In order to see this, we define for a completely $\sqcap$-distributive function $g : \mathbb{D} \to \mathbb{D}$ the *lower adjoint*:

$$g^+(d') = \bigsqcap\{d \in \mathbb{D} \mid d' \sqsubseteq g(d)\}$$

It is well-known that lower and upper adjoints determine each other uniquely. Thus, we have, for every completely distributive g, $(g^-)^+ = g$. Summarizing, we conclude that instead of computing with g, we as well might compute with its upper adjoint g^-. From

an efficiency point of view, however, the functions g and g^- need not behave identical. Exactly this is the case for equality analysis.

An element $d \neq \bot$ of a complete lattice is *atomic* if $d_1 \sqsubseteq d$ implies that either $d_1 = \bot$ or $d_1 = d$. A lattice is *atomic* if every element x is the least upper bound of all atomic elements smaller or equal to x. Indeed, the set $\mathbb{E}$ ordered with "$\sqsubseteq$" is an atomic complete lattice — the number of atomic elements, though, is huge. A consistent equivalence relation E is atomic iff each equivalence class contains a distinct constant. Thus, the number of atomic elements in $\mathbb{E}$ equals m^k where k and m are the numbers of variables and constants, respectively.

Interestingly, the set $\mathbb{E}$ ordered with the reverse ordering "$\sqsupseteq$" is also an atomic complete lattice. For a distinction, we call the atomic elements of $(\mathbb{E}, \sqsubseteq)$ $\sqcup$-atomic and the atomic elements of the dual lattics $(\mathbb{E}, \sqsupseteq)$ $\sqcap$-atomic. For our lattice, the $\sqcap$-atomic elements are given by the single equalities $\mathbf{x}_i = \mathbf{x}_j$ and $\mathbf{x}_i = c$ for variables $\mathbf{x}_i, \mathbf{x}_j$ and constants c. Thus, the number of $\sqcap$-atomic elements is only $\mathcal{O}(k \cdot (k+m))$.

Over an atomic lattice, a completely distributive function g is given by its image on the atoms. In our case, this means that this representation for g^- is much more succinct than the corresponding representation for g. More specifically, it is of size $\mathcal{O}(k^2 \cdot (k+m))$ as the image of each of the $\mathcal{O}(k \cdot (k+m))$ $\sqcap$-atoms can be represented by a conjunction of at most k equalities. For computing the upper adjoints of the effects of procedures, we need the upper adjoints of the basic computation steps of the program. Thus, we define $[\![s]\!]^- = ([\![s]\!]^\sharp)^-$ for statements s and find:

$$[\![\mathbf{x}_i \leftarrow ?]\!]^- E = \forall \mathbf{x}_i . E = \begin{cases} E & \text{if } \mathbf{x}_i \text{ does not occur in } E \\ \bot & \text{otherwise} \end{cases}$$
$$[\![\mathbf{x}_i \leftarrow t]\!]^- E = E[t/\mathbf{x}_i]$$

In case of the complete lattice $\mathbb{E}$, we realize that the upper adjoints of the abstract transformers of assignments in fact equal the *weakest pre-condition transformers* for these statements. An implementation of these abstract transformers for arguments $E \neq \bot$ (represented as an array) is given in Fig. 6. In particular, we find that each of these transformers can be evaluated in time $\mathcal{O}(k)$.

5 Inter-procedural Analysis

In the following, we present our inter-procedural analysis. For simplicity, we first consider global variables only. Assume that the set of global variables is given by $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$. The effect of a single edge is represented by a completely distributive function from $\mathbb{F} = \mathbb{E} \to \mathbb{E}$ where $\mathbb{E} = \mathbb{E}(\mathbf{X}, V)$. Again, we collect the abstract effects of procedures *from the rear*:

$[\![\mathsf{rt}_f]\!]^-$	$\sqsubseteq \mathsf{Id}$	rt_f exit point of procedure f
$[\![u]\!]^-$	$\sqsubseteq [\![\mathsf{st}_f]\!]^- \circ [\![v]\!]^-$	$(u, f(), v)$ a call edge, st_f entry point of f
$[\![u]\!]^-$	$\sqsubseteq [\![s]\!]^- \circ [\![v]\!]^-$	(u, s, v) an assignment edge

where $\mathsf{Id}\, E = E$ for every $E \in \mathbb{E}$. For a program point u of a procedure f, $[\![u]\!]^-$ describes the upper adjoint of the transformation induced by same-level program executions starting from u and reaching the procedure exit of f at the same level.

$$
\begin{array}{lll}
[\![\mathbf{x}_i \leftarrow ?]\!]^- E & = & \mathbf{if}\ (E[i] \neq \mathbf{x}_i)\ \ \mathbf{return}\ \ \bot; \\
& & \mathbf{else\ if}\ (\exists\, j \neq i.\ E[j] = E[i])\ \ \mathbf{return}\ \ \bot; \\
& & \mathbf{else}\ \ \mathbf{return}\ \ E; \\
[\![\mathbf{x}_i \leftarrow c]\!]^- E & = & \mathbf{if}\ (E[i] = c' \in V)\ \mathbf{if}\ (c \neq c')\ \ \mathbf{return}\ \ \bot; \\
& & \qquad\qquad \mathbf{else}\ \ \{E[i] \leftarrow \mathbf{x}_i; \mathbf{return}\ \ E;\} \\
& & \mathbf{else}\ \{\quad // \quad E[i] \text{ is a variable} \\
& & \quad X \leftarrow \{\mathbf{x}_j \mid j \neq i, E[j] = E[i]\}; \\
& & \quad E[i] \leftarrow \mathbf{x}_i;\ \ \mathbf{forall}\,(\mathbf{x}_j \in X)\ \ E[j] \leftarrow c; \\
& & \quad \mathbf{return}\ \ E; \\
& & \} \\
[\![\mathbf{x}_i \leftarrow \mathbf{x}_j]\!]^- E & = & \mathbf{if}\ (E[i] = c \in V)\ \mathbf{if}\ (E[j] = c' \in V)\ \mathbf{if}\ (c \neq c')\ \ \mathbf{return}\ \ \bot; \\
& & \qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \ \{E[i] \leftarrow \mathbf{x}_i; \mathbf{return}\ \ E;\} \\
& & \qquad\qquad\qquad \mathbf{else}\ \{\quad // \quad E[j] \text{ is a variable} \\
& & \qquad\qquad\quad X \leftarrow \{\mathbf{x}_{j'} \mid E[j'] = E[j]\}; \\
& & \qquad\qquad\quad E[i] \leftarrow \mathbf{x}_i;\ \ \mathbf{forall}\,(\mathbf{x}_{j'} \in X)\ \ E[j'] \leftarrow c; \\
& & \qquad\qquad\quad \mathbf{return}\ \ E; \\
& & \qquad\qquad\} \\
& & \mathbf{else}\ \{\quad // \quad E[i] \text{ is a variable} \\
& & \quad X \leftarrow \{\mathbf{x}_{i'} \mid i' \neq i, E[i'] = E[i]\}; \\
& & \quad E[i] \leftarrow \mathbf{x}_i;\ \ \mathbf{forall}\,(\mathbf{x}_{i'} \in X)\ \ E[i'] \leftarrow E[j]; \\
& & \quad \mathbf{return}\ \ E; \\
& & \}
\end{array}
$$

Fig. 6. The Implementation of the transformers $[\![s]\!]^-$ for $E \neq \bot$.

The crucial computation step here is the composition $h^- \circ g^-$ where $g^-, h^- \in \mathbb{F}$. In order to determine $h^-(g^-(e))$ for an equality e, we recall that $g^-(e)$ is represented by at most k equalities e'. We can determine $h^-(g^-(e))$ by computing the greatest lower bound of the values $h^-(e')$, i.e. of at most k equivalence relations. By Lemma 1 (2), the latter takes time $\mathcal{O}(k^2)$. For determining a representation of $h^- \circ g^-$, the values $h^-(g^-(e))$ need to be computed for $\mathcal{O}(k \cdot (k+m))$ equalities if m is the number of constants. We conclude that composition can be computed in time $\mathcal{O}(k^3 \cdot (k+m))$.

Since the expressions on right-hand sides of in-equations are monotonic, the system of in-equations has a unique *greatest* solution. Since the operations used in right-hand sides of the equation system are completely $\sqcap$-distributive, we obtain:

Theorem 1. *For every procedure f and every program point u of f, $[\![u]\!]^- = (\bar{\alpha}(\mathcal{S}[u]))^-$.*

Here, the abstraction function $\bar{\alpha} : (2^{V^k} \to 2^{V^k}) \to \mathbb{E} \to \mathbb{E}$ for summary functions is defined as the best abstract transformer, i.e., by $\bar{\alpha}(g) = \alpha \circ g \circ \gamma$. We observe that, during evaluation of a procedure, the values of constants will not change. Therefore, instead of analyzing the weakest pre-condition for every equation $\mathbf{x}_i = c$, $c \in V$, separately, we can as well determine the weakest pre-condition for the single equation $\mathbf{x}_i = \bullet$ for a distinguished fresh variable $\bullet$. The weakest pre-condition E_c for the specific equation $\mathbf{x}_i = c$ then can be determined from the weakest pre-condition E for $\mathbf{x}_i = \bullet$ by substituting c for $\bullet$, i.e., as $E_c = E[c/\bullet]$. The advantage is that now the size of the representation of a function is just $\mathcal{O}(k^3)$ and thus independent of the number of

constants occurring in the program. Also, composition of function then can be executed in time $\mathcal{O}(k^4)$. Note that variables not assigned to during procedure evaluation can also be treated as constants and therefore be captured by $\bullet$ — thus allowing to shrink the representation of summary functions even further.

Example 2. Consider the program from Fig. 2. The set of variables is $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2\}$. The assignments $\mathbf{x}_1 \leftarrow \mathbf{x}_2$ and $\mathbf{x}_2 \leftarrow 2$ correspond to the functions h_1^-, h_2^- with

	h_1^-	h_2^-
$\mathbf{x}_1 = \mathbf{x}_2$	$\top$	$\mathbf{x}_1 = 2$
$\mathbf{x}_1 = \bullet$	$\mathbf{x}_2 = \bullet$	$\mathbf{x}_1 = \bullet$
$\mathbf{x}_2 = \bullet$	$\mathbf{x}_2 = \bullet$	$2 = \bullet$

In a first round of Round-Robin iteration, we obtain for program points $11, 10, 9, 8, 7$ of the procedure *work*:

	11	10	9	8	7
$\mathbf{x}_1 = \bullet$	$\mathbf{x}_1 = \bullet$	$\mathbf{x}_1 = \bullet$	$\top$	$\mathbf{x}_1 = \bullet$	$\mathbf{x}_2 = \bullet$
$\mathbf{x}_2 = \bullet$	$\mathbf{x}_2 = \bullet$	$2 = \bullet$	$\top$	$2 = \bullet$	$2 = \bullet$
$\mathbf{x}_1 = \mathbf{x}_2$	$\mathbf{x}_1 = \mathbf{x}_2$	$\mathbf{x}_1 = 2$	$\top$	$\mathbf{x}_1 = 2$	$\mathbf{x}_2 = 2$

In this example, the fixpoint is reached already after the first iteration. □

From the upper adjoint $[\![\mathsf{st}_f]\!]^-$, we obtain the abstract effect of procedure f by:

$$[\![f]\!]^\sharp(E) = ([\![\mathsf{st}_f]\!]^-)^+(E) = \bigsqcap\{E' \mid [\![\mathsf{st}_f]\!]^-(E') \sqsupseteq E\}$$

where the E' in the greatest lower bound are supposed to be $\sqcap$-atomic. The number of these elements is $\mathcal{O}(k \cdot (k+m))$. Using the trick with the extra variable $\bullet$, we can compute the application of $[\![f]\!]^\sharp$ to a given element E in time $\mathcal{O}(k^3)$ – independent of the number of constants occurring in the program.

The functions $[\![f]\!]^\sharp$ can be used to determine, for every program point u the conjunction of all equalities $\mathcal{E}[u] \in \mathbb{E}$ which definitely hold when the program point u is reached. For that, we put up the following system of in-equations whose unknowns $\mathcal{E}[v]$ (v program point) take values in $\mathbb{E}$:

$$\begin{array}{lll}
\mathcal{E}[\mathsf{st}_{main}] & \sqsupseteq \top & \\
\mathcal{E}[\mathsf{st}_f] & \sqsupseteq \mathcal{E}[u] & (u, f(), _) \text{ a call edge} \\
\mathcal{E}[v] & \sqsupseteq [\![f]\!]^\sharp(\mathcal{E}[u]) & (u, f(), v) \text{ a call edge} \\
\mathcal{E}[v] & \sqsupseteq [\![s]\!]^\sharp(\mathcal{E}[u]) & (u, s, v) \text{ an assignment edge}
\end{array}$$

It should be noted that, during fixpoint iteration, we never must construct $[\![f]\!]^\sharp$ as a whole. Instead, we only need to evaluate these functions on argument values E. Since all right-hand sides are monotonic, this system of in-equations has a least solution.

Example 3. Consider again the program from Fig. 2. Then we obtain the following equalities for program points 0 through 11:

$1, 2, 3$	4	5	6	$7, 8, 9$	10	11
$\top$	$\mathbf{x}_1 = \mathbf{x}_2$	$\mathbf{x}_2 = 2$	$\mathbf{x}_1 = \mathbf{x}_2 = 2$	$\mathbf{x}_1 = \mathbf{x}_2$	$\top$	$\mathbf{x}_2 = 2$

□

Also for inter-procedural reachability, a precision theorem can be proven. We have:

Theorem 2. *The least solution $\mathcal{E}[v]$, v a program point, can be computed in time $\mathcal{O}(n \cdot k^4)$ where n is the size of the program and k is the number of variables in the program. Moreover, for every program point v, $\mathcal{E}[v] = \alpha(\mathcal{C}[v])$.* □

In order to compute the least solution within the stated running time, we first compute the values $[\![\mathsf{st}_f]\!]^-$ by applying *semi-naive* fixpoint iteration as in [2] to the system of in-equations characterizing the (upper adjoints of) summary functions. The key idea of semi-naive iteration is to propagate just the individual increments to attained values instead of abstract values as a whole. In our case, such an increment consists of a single equality $(\mathbf{x}_i = t)$ that is added as an additional conjunct to the pre-condition of some $\sqcap$-atomic element in the representation of some computed summary function. Thus, distributed over the fixpoint computation, the accumulated effort spent on a single in-equation is not bigger than the effort for a single complete evaluation of the right hand side on functions with a representation of maximum size. As mentioned, the most complex operation occuring in a right hand side, composition of functions, can be computed in time $\mathcal{O}(k^4)$ using the special variable $\bullet$. Given the values $[\![\mathsf{st}_f]\!]^-$, the fixpoint of the system of in-equations for $\mathcal{E}$ can be computed by standard fixpoint iteration: as the height of the lattice $\mathbb{E}$ is $\mathcal{O}(k)$ each right hand side is re-evaluated at most $\mathcal{O}(k)$ times and the most complex operation, application of $[\![f]\!]^\sharp$ takes time $\mathcal{O}(k^3)$. The total running time estimation given in Theorem 2 follows by summing up over all in-equations as their number is bounded by the size of the program.

The resulting bound is by a factor k larger than the best known upper bound for copy constant propagation [6] where no equalities between variables are tracked. On the other hand, instead of relying on equivalence relations, we could code variable equalities as specific linear dependences. The techniques from [11] then result in an algorithm with worst-case complexity $\mathcal{O}(n \cdot k^8)$ — which is a factor k^4 worse than the new bound.

6 Local Variables

In the following, we extend our inter-procedural analysis to local variables.

Example 4. Consider the program from Fig. 7. The local variable a_1 of procedure $work$ can be coalesced with the global x_1 as both are equal throughout the body of $work$. □

In order to simplify notation, we assume that all procedures have the same set of local variables $\mathbf{A} = \{\mathbf{a}_1, \ldots, \mathbf{a}_l\}$. The set of global variables is still $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$. First of all, we extend the collecting semantics to local variables. A state is now described by a vector $(x_1, \ldots, x_k, a_1, \ldots, a_l) \in V^{k+l}$ which is identified with the pair (x, a) of vectors $x = (x_1, \ldots, x_k) \in V^k$ and $a = (a_1, \ldots, a_l) \in V^l$ of values for the global and local variables, respectively. The transformations $\mathcal{S}[u]$ now are taken from the set $\mathbb{T} = V^{k+l} \to V^{k+l}$. In order to avoid confusion between the values of the local variables of caller and callee the rules for call edges must be modified. For this purpose we introduce two transformations: The first, $\mathsf{enter} \in \mathbb{T}$, captures how a set of states propagates from the call to the start edge of the called procedure:

$$\mathsf{enter}(X) = \{(x, a) \mid \exists a' : (x, a') \in X\}$$

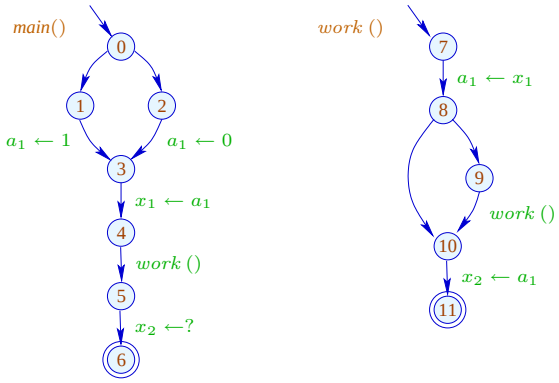

Fig. 7. An example program with local variable $\mathbf{a}_1$.

Here, we assume that local variables have an arbitrary value at the beginning of their scope but other conventions can be described similarly. The second transformation $H : \mathbb{T} \to \mathbb{T}$ adjusts the transformation computed for a called procedure to the caller:

$$H(g)(X) = \{(x', a) \mid \exists x, a' : (x', a') \in g(\mathsf{enter}\,\{(x, a)\})\}$$

It ensures that local variables of the caller are left untouched by the call. The modified rules for call edges in the systems of in-equations for $\mathcal{S}$ and $\mathcal{C}$ look as follows:

$\mathcal{S}[u]$	$\supseteq \mathcal{S}[v] \circ H(\mathcal{S}[\mathsf{st}_f])$	$(u, f(), v)$ a call edge, st_f entry point of f
$\mathcal{C}[\mathsf{st}_f]$	$\supseteq \mathsf{enter}(\mathcal{C}[u])$	$(u, f(), \_)$ a call edge
$\mathcal{C}[v]$	$\supseteq H(\mathcal{S}[\mathsf{st}_f])(\mathcal{C}[u])$	$(u, f(), v)$ a call edge

In addition, V^k is replaced by $\mathsf{enter}(V^{k+l})$ in the in-equation for $\mathcal{C}[\mathsf{st}_{main}]$.

As for global variables alone, we first define the domain for the forward analysis whose summary functions then are represented through their upper adjoints. The extra complication is that now equalities may involve local variables of the procedures on the call stack which are not visible inside the called procedure. The solution is to allow *auxiliary variables* from a set $\mathbf{Y}$ (of cardinality k) for distinct local information of the caller which must be tracked by the callee, but not modified. Thus, the abstract forward semantics of procedures operates on conjunctions of equations over global variables $\mathbf{X}$, local variables $\mathbf{A}$, and auxiliary variables $\mathbf{Y}$, i.e., takes elements from $\mathbb{E}' = \mathbb{E}(\mathbf{X} \cup \mathbf{A} \cup \mathbf{Y}, V)$. Since at procedure exit, local variables of the procedure are no longer of interest, post-conditions are conjunctions just over global and auxiliary variables, i.e., summary functions should return elements from $\mathbb{E}'' = \mathbb{E}(\mathbf{X} \cup \mathbf{Y}, V)$. Thus, forward summary functions are completely distributive functions from $\mathbb{F}' = \mathbb{E}' \to \mathbb{E}''$ whereas their upper adjoints are completely $\sqcap$-distributive functions from $\mathbb{F}'' = \mathbb{E}'' \to \mathbb{E}'$. In this setting, the abstraction function $\tilde{\alpha} : (2^{V^{k+l}} \to 2^{V^{k+l}}) \to \mathbb{F}'$ takes the form:

$$\begin{aligned}\tilde{\alpha}(F)(E) \quad = \quad & \bigsqcap\{E' \mid \forall y, x', a' : \\ & (x', a') \in F(\{(x, a) \mid (x, a, y) \models E\}) \Rightarrow (x', y) \models E'\}\,.\end{aligned}$$

As in Section 5, we aim at avoiding to treat each constant in post-conditions separately. Recall that auxiliary variables from $\mathbf{Y}$ are not modified during the execution of the call. We conclude that, for the sake of determining weakest pre-conditions, at most one auxiliary variable, say $\bullet$, suffices in single equality post-conditions. Since we have at most this single $\bullet$ in the post-condition, we also have at most one $\bullet$-variable in pre-conditions. Accordingly, we represent upper adjoints by completely $\sqcap$-distributive functions from:

$$\mathbb{F}_0 = \mathbb{E}(\mathbf{X} \cup \{\bullet\}, \emptyset) \to \mathbb{E}(\mathbf{X} \cup \mathbf{A} \cup \{\bullet\}, V)$$

Any such function $g \in \mathbb{F}_0$ is meant to represent the function $\mathsf{Ext}(g) \in \mathbb{F}''$ defined by:

$$\begin{array}{ll} \mathsf{Ext}(g)(\mathbf{x}_i = \mathbf{x}_j) & = g(\mathbf{x}_i = \mathbf{x}_j) \\ \mathsf{Ext}(g)(e) & = \begin{cases} e, & \text{if } g^-(\mathbf{x}_1 = \bullet) \neq \top \\ \top, & \text{if } g^-(\mathbf{x}_1 = \bullet) = \top \end{cases} \\ \mathsf{Ext}(g)(\mathbf{x}_i = t) & = g(\mathbf{x}_i = \bullet)[t/\bullet] \end{array}$$

where the equality e and the term t contain only constants, local variables or $\bullet$. The first clause exploits that g is special in that it does not introduce $\bullet$ for post-conditions not containing $\bullet$. The second clause deals with equalities between local variables and constants in presence of non-termination of the called procedure (identified through $g^-(\mathbf{x}_1 = \bullet) = \top$). In order to determine the representations from $\mathbb{F}_0$ for procedures, effects of control-flow edges are described by completely $\sqcap$-distributive functions from

$$\mathbb{F} = \mathbb{E}(\mathbf{X} \cup \{\bullet\} \cup \mathbf{A}, V) \to \mathbb{E}(\mathbf{X} \cup \{\bullet\} \cup \mathbf{A}, V)$$

If g^- is the (upper adjoint of the) effect of a procedure body, the (upper adjoint of the) effect of a call to this procedure is given by $H^-(g^-) \in \mathbb{F}$ where

$$\begin{array}{ll} H^-(g^-)(\mathbf{x}_i = \mathbf{x}_j) & = \forall \mathbf{a}_1 \ldots \mathbf{a}_l.\, g^-(\mathbf{x}_i = \mathbf{x}_j) \\ H^-(g^-)(e) & = \begin{cases} e, & \text{if } g^-(\mathbf{x}_1 = \bullet) \neq \top \\ \top, & \text{if } g^-(\mathbf{x}_1 = \bullet) = \top \end{cases} \\ H^-(g^-)(\mathbf{x}_i = t) & = (\forall \mathbf{a}_1 \ldots \mathbf{a}_l.\, g^-(\mathbf{x}_i = \bullet))[t/\bullet] \end{array}$$

Here, the equality e and the term t contain only constants, local variables or $\bullet$. Then summary functions can be characterized by the least solution of the constraint system:

$$\begin{array}{lll} [\![\mathsf{rt}_f]\!]^- \sqsubseteq \mathsf{Id} & & \mathsf{rt}_f \text{ exit point of procedure } f \\ [\![u]\!]^- \sqsubseteq H^-([\![\mathsf{st}_f]\!]^-) \circ [\![v]\!]^- & & (u, f(), v) \text{ a call edge, } \mathsf{st}_f \text{ entry point of } f \\ [\![u]\!]^- \sqsubseteq [\![s]\!]^- \circ [\![v]\!]^- & & (u, s, v) \text{ an assignment edge} \end{array}$$

where $\mathsf{Id}\, E = E$ for every $E \in \mathbb{E}(\mathbf{X} \cup \{\bullet\}, \emptyset)$. For a program point u of a procedure f, $[\![u]\!]^- \in \mathbb{F}_0$ describes the upper adjoint of the transformation induced by program executions that start at u and reach the procedure exit of f at the same level.

The crucial computation step here is the composition $h^- \circ g^-$ for $g^- \in \mathbb{F}_0$ and $h^- \in \mathbb{F}$. In order to determine the value $h^-(g^-(e))$ for an equality e, we recall that every equivalence relation $g^-(e)$ is represented by at most $k + l + 1$ equalities e' for k global and l local variables. Thus, $h^-(g^-(e))$ can be computed as the greatest lower bound of the $\mathcal{O}(k + l)$ equivalence relations $h^-(e')$. By Lemma 1 (2), the latter can be done in time $\mathcal{O}((k + l)^2)$. For determining $h^- \circ g^-$, the values $h^-(g^-(e))$ must be computed for $\mathcal{O}(k^2)$ equalities. Thus, composition can be computed in time $\mathcal{O}(k^2(k + l)^2)$.

Example 5. Consider the program from Fig. 7. The assignments $\mathbf{a}_1 \leftarrow \mathbf{x}_1$ and $\mathbf{x}_2 \leftarrow \mathbf{a}_1$ correspond to the following functions:

	$\mathbf{a}_1 = \mathbf{x}_1$	$\mathbf{a}_1 = \mathbf{x}_2$	$\mathbf{x}_1 = \mathbf{x}_2$	$\mathbf{a}_1 = \bullet$	$\mathbf{x}_1 = \bullet$	$\mathbf{x}_2 = \bullet$
$[\![\mathbf{a}_1 \leftarrow \mathbf{x}_1]\!]^-$	$\top$	$\mathbf{x}_1 = \mathbf{x}_2$	$\mathbf{x}_1 = \mathbf{x}_2$	$\mathbf{x}_1 = \bullet$	$\mathbf{x}_1 = \bullet$	$\mathbf{x}_2 = \bullet$
$[\![\mathbf{x}_2 \leftarrow \mathbf{a}_1]\!]^-$	$\mathbf{a}_1 = \mathbf{x}_1$	$\top$	$\mathbf{a}_1 = \mathbf{x}_1$	$\mathbf{a}_1 = \bullet$	$\mathbf{x}_1 = \bullet$	$\mathbf{a}_1 = \bullet$

In a first round of Round-Robin iteration, we obtain for program points $11, 10, 9, 8, 7$:

	11	10	9	8	7
$\mathbf{x}_1 = \bullet$	$\mathbf{x}_1 = \bullet$	$\mathbf{x}_1 = \bullet$	$\top$	$\mathbf{x}_1 = \bullet$	$\mathbf{x}_1 = \bullet$
$\mathbf{x}_2 = \bullet$	$\mathbf{x}_2 = \bullet$	$\mathbf{a}_1 = \bullet$	$\mathbf{a}_1 = \bullet$	$\mathbf{a}_1 = \bullet$	$\mathbf{x}_1 = \bullet$
$\mathbf{x}_1 = \mathbf{x}_2$	$\mathbf{x}_1 = \mathbf{x}_2$	$\mathbf{x}_1 = \mathbf{a}_1$	$\top$	$\mathbf{x}_1 = \mathbf{a}_1$	$\top$

The second iteration changes the value for the postcondition $\mathbf{x}_1 = \mathbf{x}_2$ at program point 9 from $\top$ to $\mathbf{x}_1 = \mathbf{a}_1$. Here, the fixpoint is reached after the second iteration. □

Since the expressions on right-hand sides of in-equations are completely distributive, the system of in-equations has a unique greatest solution, and we find:

Theorem 3. *For every program point u, $\mathsf{Ext}([\![u]\!]^-) = (\tilde{\alpha}(\mathcal{S}[u]))^-$.*

The proof of this theorem is a generalization of the corresponding proof for Theorem 1. From $[\![\mathsf{st}_f]\!]^-$, we again obtain the abstract effect of a call to f, this time by

$$[\![f]\!]^\sharp(E) = (H^-([\![\mathsf{st}_f]\!]^-))^+(E)$$

where $g^+(E) = \bigwedge\{e \mid E \sqsubseteq g(e)\}$. According to the special structure of g, time $\mathcal{O}((k+l)^2 \cdot k)$ is sufficient to compute all equalities e with $E \sqsubseteq H^-([\![\mathsf{st}_f]\!]^-)(e)$.

The abstract effects $[\![f]\!]^\sharp$ allow to determine for every program point u, the conjunction of all equalities which hold when reaching u. These are characterized by:

$$\begin{array}{lll}
\mathcal{E}[\mathsf{st}_{main}] & \sqsupseteq \mathsf{enter}^\sharp(\top) & \\
\mathcal{E}[\mathsf{st}_f] & \sqsupseteq \mathsf{enter}^\sharp(\mathcal{E}[u]) & (u, f(), _) \text{ a call edge} \\
\mathcal{E}[v] & \sqsupseteq [\![f]\!]^\sharp(\mathcal{E}[u]) & (u, f(), v) \text{ a call edge} \\
\mathcal{E}[v] & \sqsupseteq [\![s]\!]^\sharp(\mathcal{E}[u]) & (u, s, v) \text{ an assignment edge}
\end{array}$$

where $\mathsf{enter}^\sharp(E)$ is the conjunction of all equalities e involving only globals and constants implied by E. The resulting consistent equivalence relation can be constructed in time $\mathcal{O}(k+l)$. This is also the case for $[\![s]\!]^\sharp(E)$, s an assignment (see Section 3).

Example 6. Consider the program from Fig. 7. We obtain the following equalities:

$0, 1, 2, 3$	4	5	6	7	$8, 9, 10$	11
$\top$	$\mathbf{x}_1 = \mathbf{a}_1$	$\mathbf{x}_1 = \mathbf{x}_2 = \mathbf{a}_1$	$\mathbf{x}_1 = \mathbf{a}_1$	$\top$	$\mathbf{x}_1 = \mathbf{a}_1$	$\mathbf{x}_1 = \mathbf{x}_2 = \mathbf{a}_1$

We conclude that inside the procedure *work*, we can coalesce $\mathbf{x}_1$ and $\mathbf{a}_1$ and thus avoid to intermediately move the value of the global $\mathbf{x}_1$ into the local $\mathbf{a}_1$. □

Theorem 4. *The system of in-equations for reachability in presence of local variables has a least solution $\mathcal{E}[v]$, v program point, where for every v, $\mathcal{E}[v] = \alpha(\mathcal{C}[v])$.*

Thus, the sets of valid equalities at all program points can be computed in time $\mathcal{O}(n \cdot k^2 \cdot (k+l)^2)$ for programs of size n with k global and l local variables. □

7 Conclusion

We have provided an algorithm for inter-procedurally inferring all valid variable-variable and variable-constant equalities — after abstracting from guards and complex assignments. Based on the succinct representation of summary functions through their upper adjoints, we constructed a polynomial time algorithm with worst-case complexity $\mathcal{O}(n \cdot k^4)$ (where k is the number of program variables and n is the size of the program). We then extended our approach to programs with local variables. The key observation is that upper adjoints allow very succinct representations of summary functions: on the one hand, the number of $\sqcap$-atomic elements is smaller than the number of $\sqcup$-atomic elements, on the other hand, we can avoid tracking each constant individually. Similar ideas may also help to speed up further inter-procedural program analyses. In future work, we also want to apply our analysis to inter-procedural register coalescing.

References

1. B. Alpern, M. Wegman, and F. K. Zadeck. Detecting Equality of Variables in Programs. In *15th ACM Symp. on Principles of Programming Languages (POPL)*, pages 1–11, 1988.
2. C. Fecht and H. Seidl. Propagating Differences: An Efficient New Fixpoint Algorithm for Distributive Constraint Systems. *Nordic Journal of Computing (NJC)*, 5(4):304–329, 1998.
3. L. George and A. W. Appel. Iterated Register Coalescing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(3):300–324, 1996.
4. S. Gulwani and G. C. Necula. A Polynomial-Time Algorithm for Global Value Numbering. In *11th Int. Static Analysis Symposium (SAS)*, pages 212–227. Springer, LNCS 3148, 2004.
5. S. Horwitz, T. W. Reps, and M. Sagiv. Demand Interprocedural Dataflow Analysis. In *3rd ACM Symp. on the Foundations of Software Engineering (FSE)*, pages 104–115, 1995.
6. S. Horwitz, T. W. Reps, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *22nd ACM Symp. on Principles of Programming Languages (POPL)*, pages 49–61, 1995.
7. J. Knoop. Parallel Data-Flow Analysis of Explicitly Parallel Programs. In *5th Int. Euro-Par Conference*, pages 391–400. Springer-Verlag, LNCS 1565, 1999.
8. J. Knoop and B. Steffen. The Interprocedural Coincidence Theorem. In *Compiler Construction (CC)*, pages 125–140. LNCS 541, Springer-Verlag, 1992.
9. A. Melton, D. A. Schmidt, and G. E. Strecker. Galois Connections and Computer Science Applications. In D. Pitt, S. Abramsky, A. Poigné, and D. Rydeheard, editors, *Category Theory and Computer Programming*, pages 299–312. Springer-Verlag, LNCS 240, 1985.
10. M. Müller-Olm, O. Rüthing, and H. Seidl. Checking Herbrand Equalities and Beyond. In *Verification, Model-Checking, and Abstract Interpretation (VMCAI)*, pages 79–96. Springer Verlag, LNCS 3385, 2005.
11. M. Müller-Olm and H. Seidl. Precise Interprocedural Analysis through Linear Algebra. In *31st ACM Symp. on Principles of Programming Languages (POPL)*, pages 330–341, 2004.
12. M. Müller-Olm, H. Seidl, and B. Steffen. Interprocedural Herbrand Equalities. In *14th European Symp. on Programming (ESOP)*, pages 31–45. Springer Verlag, LNCS 3444, 2005.
13. M. Sharir and A. Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In: S.S. Muchnick and N.D. Jones (editors). *Program Flow Analysis: Theory and Applications*. Prentice Hall, Engelwood Cliffs, New Jersey, 1981, chapter 7, pages 189–233.
14. B. Steffen, J. Knoop, and O. Rüthing. The Value Flow Graph: A Program Representation for Optimal Program Transformations. In *3rd European Symp. on Programming (ESOP)*, pages 389–405. Springer-Verlag, LNCS 432, 1990.