Enforcing Termination of Interprocedural Analysis

Stefan Schulze Frielinghaus, Helmut Seidl, and Ralf Vogler

Fakultät für Informatik, TU München, Germany,
{schulzef,seidl,voglerr}@in.tum.de

Abstract. Interprocedural analysis by means of partial tabulation may not terminate when the same procedure is analyzed for infinitely many abstract calling contexts or when the abstract domain has infinite strictly ascending chains. As a remedy, we present a novel local solver for general abstract equation systems, be they monotonic or not, and prove that this solver fails to terminate only when infinitely many variables are encountered. We clarify in which sense the computed results are sound. Moreover, we show that interprocedural analysis performed by this novel local solver, is guaranteed to terminate for all non-recursive programs — irrespective of whether the complete lattice is infinite or has infinite strictly ascending or descending chains.

1 Introduction

Since [9], it has been known that static analysis of run-time properties of programs by means of abstract interpretation can be compiled into systems of equations over complete lattices. Thereby, various interesting properties require complete lattices which may have infinite strictly ascending or descending chains [15,7,6]. In order to determine a (post-) solution of a system of equations over such lattices, Cousot and Cousot propose to perform a first phase of iteration using a *widening* operator to obtain a post solution which later may be improved by a second phase of iteration using a *narrowing* operator. This strict arrangement into separate phases, though, has the disadvantage that precision unnecessarily may be given up which later is difficult to recover. In [4,5,2], it has been observed that widening and narrowing need not be organized into separate phases. Instead various algorithms are proposed which *intertwine* widening with narrowing in order to compute a (reasonably small) post fixpoint of the given system of equations. The idea there is to combine widening with narrowing into a single operator and then to iterate according to some fixed ordering over the variables of the system. Still, monotonicity of all right-hand sides is required for the resulting algorithms to be terminating [4,2].

Non-monotonic right-hand sides, however, are introduced by interprocedural analysis in the style of [11,3] when partial tabulation of all occurring abstract contexts is used. In order to see this, consider an abstract lattice $\mathbb{D}$ of possible program invariants. Then the abstract effect of a procedure call can be formalized

as a transformation $f^\sharp$ from $\mathbb{D} \to \mathbb{D}$. For rich lattices $\mathbb{D}$ such transformations may be difficult to represent and compute with. As a remedy, each single variable function may be decomposed into a set of variables — one for each possible argument — where each such variable now receives values from $\mathbb{D}$ only. As a result, the difficulty of dealing with elements of $\mathbb{D} \to \mathbb{D}$ is replaced with the difficulty of dealing with systems of equations which are infinite when $\mathbb{D}$ is infinite. Moreover, composition of abstract functions is translated into *indirect addressing* of variables (the outcome of the analysis for one function call determines for which argument another function is queried) — implying non-monotonicity [14]. Thus, termination of interprocedural analysis by means of the solvers from [4,2] cannot be guaranteed. Interestingly, the *local* solver SLR_3 from [2] terminates in many practical cases. Nontermination, though, may arise in two flavors:

- infinitely many variables may be encountered, i.e., some procedure may be analyzed for an ever growing number of calling contexts;
- the algorithm may for some variable switch infinitely often from a narrowing iteration back to a widening iteration.

From a conceptual view, the situation still is unsatisfactory: any solver used as a fixpoint engine within a static analysis tool should reliably terminate under reasonable assumptions. In this paper, we therefore re-examine interprocedural analysis by means of local solvers. First, we extend an ordinary local solver to a two-phase solver which performs widening and subsequently narrowing. The novel point is that both iterations are performed in a demand-driven way so that also during the narrowing phase fresh variables may be encountered for which no sound over-approximation has yet been computed.

In order to enhance precision of this demand-driven two-phase solver, we then design a new local solver which intertwines the two phases. In contrast to the solvers in [4,2], however, we can no longer rely on a fixed combination of a widening and a narrowing operator, but must enhance the solver with extra logic to decide when to apply which operator. For both solvers, we prove that they terminate — whenever only finitely many variables are encountered: irrespective whether the abstract system is monotonic or not. Both solvers are guaranteed to return (partial) post-solutions of the abstract system of equations only if all right-hand sides are monotonic. Therefore, we make clear in which sense the computed results are nonetheless sound — even in the non-monotonic case. For that, we provide a sufficient condition for an abstract variable assignment to be a sound description of a concrete system — given only a (possibly non-monotonic) abstract system of equations. This sufficient condition is formulated by means of the *lower monotonization* of the abstract system. Also, we elaborate for partial solutions in which sense the domain of the returned variable assignment provides sound information. Here, the formalization of purity of functions based on computation trees and variable dependencies plays a crucial role. Finally, we prove that interprocedural analysis in the style of [11,3] with partial tabulation using our local solvers terminates for all non-recursive programs and every complete lattice with or without infinite strictly ascending or descending chains.

The paper is organized as follows. In Section 2 we recall the basics of abstract interpretation and introduce the idea of a lower monotonization of an abstract system of equations. In Section 3 we recapitulate widening and narrowing. As a warm-up, a terminating variant of Round-Robin iteration is presented in Section 4. In Section 5 we formalize the idea of local solvers based on the notion of purity of functions of right-hand sides of abstract equation systems and provide a theorem indicating in which sense local solvers for non-monotonic abstract systems compute sound results for concrete systems. A first local solver is presented in Section 6 where widening and narrowing is done in conceptually separated phases. In Section 7, we present a local solver where widening and narrowing is intertwined. Section 8 considers the abstract equations systems encountered by interprocedural analysis. A concept of stratification is introduced which is satisfied if the programs to be analyzed are non-recursive. These notions enable us to prove our main result concerning termination of interprocedural analysis with partial tabulation by means of the solvers from sections 6 and 7.

2 Basics on Abstract Interpretation

In the following we recapitulate the basics of abstract interpretation as introduced by Cousot and Cousot [9,12]. Assume that the concrete semantics of a system is described by a system of equations

$$x = f_x, \quad x \in X \tag{1}$$

where X is a set of variables taking values in some power set lattice $(\mathbb{C}, \subseteq, \cup)$ where $\mathbb{C} = 2^Q$ for some set Q of concrete program states, and for each $x \in X$, $f_x : (X \to \mathbb{C}) \to \mathbb{C}$ is the defining right-hand side of x. For the concrete system of equations, we assume that all right-hand sides $f_x, x \in X$, are *monotonic*. Accordingly, this system of equations has a unique least solution σ which can be obtained as the least upper bound of all assignments σ_τ, τ an ordinal. The assignments $\sigma_\tau : X \to \mathbb{C}$ are defined as follows. If $\tau = 0$, then $\sigma_\tau\, x = \bot$ for all $x \in X$. If $\tau = \tau' + 1$ is a successor ordinal, then $\sigma_\tau\, x = f_x\, \sigma_{\tau'}$, and if τ is a limit ordinal, then $\sigma_\tau\, x = \bigcup\{f_x\, \sigma_{\tau'} \mid \tau' < \tau\}$. An *abstract* system of equations

$$y = f^\sharp_y, \quad y \in Y \tag{2}$$

specifies an analysis of the concrete system of equations. Here, Y is a set of *abstract* variables which may not necessarily be in one-to-one correspondence to the concrete variables in the set X. The variables in Y take values in some complete lattice $(\mathbb{D}, \sqsubseteq, \sqcup)$ of abstract values and for every abstract variable $y \in Y$, $f_y : (Y \to \mathbb{D}) \to \mathbb{D}$ is the abstract defining right-hand side of y. The elements $d \in \mathbb{D}$ are meant to represent invariants, i.e., properties of states. It is for simplicity that we assume the set $\mathbb{D}$ of all possible invariants to form a complete lattice, as any partial order can be embedded into a complete lattice so that all existing least upper and greatest lower bounds are preserved [21]. In order to relate concrete sets of states with abstract values, we assume that

there is a Galois connection between $\mathbb{C}$ and $\mathbb{D}$, i.e., there are monotonic functions $\alpha : \mathbb{C} \to \mathbb{D}$, $\gamma : \mathbb{D} \to \mathbb{C}$ such that for all $c \in \mathbb{C}$ and $d \in \mathbb{D}$, $\alpha(c) \sqsubseteq d$ iff $c \subseteq \gamma(d)$. Between the sets of concrete and abstract variables, we assume that there is a *description relation* $\mathcal{R} \subseteq X \times Y$. Via the Galois connection between $\mathbb{C}$ and $\mathbb{D}$, the description relation $\mathcal{R}$ between variables is lifted to a description relation $\mathcal{R}^*$ between assignments $\sigma : X \to \mathbb{C}$ and $\sigma^\sharp : Y \to \mathbb{D}$ by defining $\sigma \; \mathcal{R}^* \; \sigma^\sharp$ iff for all $x \in X, y \in Y$, $\sigma(x) \subseteq \gamma(\sigma^\sharp(y))$ whenever $x \; \mathcal{R} \; y$ holds. Following [12], we do not assume that the right-hand sides of the abstract equation system are necessarily monotonic. For a sound analysis, we only assume that all right-hand sides respect the description relation, i.e., that for all $x \in X$ and $y \in Y$ with $x \; \mathcal{R} \; y$,

$$f_x \, \sigma \subseteq \gamma(f_y^\sharp \, \sigma^\sharp) \tag{3}$$

whenever $\sigma \; \mathcal{R}^* \; \sigma^\sharp$ holds. Our key concept for proving soundness of abstract variable assignments w.r.t. the concrete system of equations is the notion of the *lower monotonization* of the abstract system. For every function $f^\sharp : (Y \to \mathbb{D}) \to \mathbb{D}$ we consider the function

$$\underline{f}^\sharp \, \sigma = \bigsqcap \{ \; f^\sharp \, \sigma' \; \mid \; \sigma \sqsubseteq \sigma' \; \} \tag{4}$$

which we call *lower monotonization* of f. By definition, we have:

Lemma 1. *For every function $f^\sharp : (Y \to \mathbb{D}) \to \mathbb{D}$ the following holds:*

1. *$\underline{f}^\sharp$ is monotonic;*
2. *$\underline{f}^\sharp \sqsubseteq f^\sharp$;*
3. *$\underline{f}^\sharp = f^\sharp$ whenever $f^\sharp$ is monotonic.* □

The lower monotonization of the abstract system (2) then is defined as the system

$$y = \underline{f}^\sharp_y, \qquad y \in Y \tag{5}$$

Since all right-hand sides of (5) are monotonic, this system has a least solution.

Example 1. Consider the single equation

$$y_1 \; = \; \mathsf{if} \; y_1 = 0 \; \mathsf{then} \; 1 \; \mathsf{else} \; 0$$

over the complete lattice of non-negative integers equipped with an infimum element, i.e., let the domain $\mathbb{D} = \mathbb{N} \cup \{\infty\}$. This system is not monotonic. Its lower monotonization is given by $y_1 = 0$. □

Proposition 1. *Assume that σ is the least solution of the concrete system* (1)*. Then $\sigma \; \mathcal{R} \; \sigma^\sharp$ for every post solution $\sigma^\sharp$ of the lower monotonization* (5)*.*

Proof. For every ordinal τ, let σ_τ denote the τth approximation of the least solution of the concrete system and assume that $\sigma^\sharp$ is a post solution of the lower monotonization of the abstract system, i.e.,

$$\sigma^\sharp \, y \sqsupseteq \underline{f}^\sharp_y \, \sigma^\sharp \qquad (y \in Y)$$

By ordinal induction, we prove that $\sigma_\tau \mathrel{\mathcal{R}} \sigma^\sharp$. The claim clearly holds for $\tau = 0$. First assume that $\tau = \tau' + 1$ is a successor ordinal, and that the claim holds for τ', i.e., $\sigma_{\tau'} \mathrel{\mathcal{R}} \sigma^\sharp$. Accordingly, $\sigma_{\tau'} \mathrel{\mathcal{R}} \sigma'$ holds for all $\sigma' \sqsupseteq \sigma^\sharp$. Consider any pair of variables x, y with $x \mathrel{\mathcal{R}} y$. Then $\sigma_\tau\, x = f_x \sigma_{\tau'} \subseteq \gamma(f_y^\sharp\, \sigma')$ for all $\sigma' \sqsupseteq \sigma^\sharp$. Accordingly, $\alpha(\sigma_\tau\, x) \sqsubseteq f_y^\sharp\, \sigma'$ for all $\sigma' \sqsupseteq \sigma^\sharp$, and therefore,

$$\alpha(\sigma_\tau\, x) \sqsubseteq \bigsqcap \{\, f_y^\sharp\, \sigma' \mid \sigma' \sqsupseteq \sigma^\sharp \,\} = \underline{f}_y^\sharp\, \sigma^\sharp \sqsubseteq \sigma^\sharp\, y$$

since $\sigma^\sharp$ is a post solution. From that, the claim follows for the ordinal τ. Now assume that τ is a limit ordinal, and that the claim holds for all ordinals $\tau' < \tau$. Again consider any pair of variables x, y with $x \mathrel{\mathcal{R}} y$. Then

$$\sigma_\tau\, x = \bigcup \{\, \sigma_{\tau'}\, x \mid \tau' < \tau \,\} \subseteq \bigcup \{\, \gamma(\sigma^\sharp\, y) \mid \tau' < \tau \,\} = \gamma(\sigma^\sharp\, y)$$

and the claim also follows for the limit ordinal τ. □

This means for the abstract system from Example 1, that $\sigma^\sharp = \{y_1 \mapsto 0\}$, as the least solution of the lower monotonization $y_1 = 0$, is a sound description of every corresponding concrete system.

Proposition 1 provides us with a sufficient condition guaranteeing that an abstract assignment $\sigma^\sharp$ is sound w.r.t. the concrete system (1) and the description relation $\mathcal{R}$, namely, that $\sigma^\sharp$ is a post solution of the system (5). This sufficient condition is remarkable as it is an *intrinsic* property of the abstract system since it does not refer to the concrete system. As a corollary we obtain:

Corollary 1. *Every post solution $\sigma^\sharp$ of the abstract system* (2) *is sound.*

Proof. We have for all $y \in Y$:

$$\sigma^\sharp\, y \sqsupseteq f_y^\sharp\, \sigma^\sharp \sqsupseteq \underline{f}_y^\sharp\, \sigma^\sharp$$

Accordingly, $\sigma^\sharp$ is a post solution of the lower monotonization of the abstract system and therefore sound. □

3 Widening and Narrowing

It is instructive to recall the basic algorithmic approach to determine non-trivial post solutions of abstract systems (2) when the set Y of variables is finite, all right-hand sides are monotonic and the complete lattice $\mathbb{D}$ has finite strictly increasing chains only. In this case, *chaotic iteration* may be applied. This kind of iteration starts with the initial assignment $\underline{\bot}$ which assigns $\bot$ to every variable $y \in Y$ and then repeatedly evaluates right-hand sides to update the values of variables until the values for all variables have stabilized. This method may also be applied if right-hand sides are non-monotonic: the only modification required is to update the value for each variable not just with the new value provided by the left-hand side, but with some upper bound of the old value for a variable

with the new value. As a result, a *post solution* of the system is computed which, according to Corollary 1, is sound.

The situation is more intricate, if the complete lattice in question has strictly ascending chains of infinite length. Here, we follow Cousot and Cousot [9,12,8] who suggest to accelerate iteration by means of *widening* and *narrowing*. A widening operator $\nabla : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ takes the old value $a \in \mathbb{D}$ and a new value $b \in \mathbb{D}$ and combines them to a value $a \sqcup b \sqsubseteq a \nabla b$ with the additional understanding that for any sequence $b_i, i \geq 0$, and any value a_0, the sequence $a_{i+1} = a_i \nabla b_i, i \geq 0$, is ultimately stable. In contrast, a narrowing operator $\Delta : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ takes the old value $a \in \mathbb{D}$ and a new value $b \in \mathbb{D}$ and combines them to a value $a \Delta b$ satisfying $a \sqcap b \sqsubseteq a \Delta b \sqsubseteq a$ — with the additional understanding that for any sequence $b_i, i \geq 0$, and any value a_0, the sequence $a_{i+1} = a_i \Delta b_i, i \geq 0$, is ultimately stable.

While the widening operator is meant to reach a post solution after a finite number of updates to each variable of the abstract system, the narrowing operator allows to improve upon a variable assignment once it is known to be sound. In particular, if all right-hand sides are monotonic, the result of a narrowing iteration, if started with a post solution of the abstract system, again results in a post solution. Accordingly, the returned variable assignment can easily be verified to be sound. In analyzers which iterate according to the syntactical structure of programs such as ASTREE [13], this strict separation into two phases, though, has been given up. There, when iterating over one loop, narrowing for the current loop is triggered as soon as locally a post-solution has been attained. This kind of intertwining widening and narrowing is systematically explored in [4,2]. There, a widening operator is combined with a narrowing operator into a single derived operator $\boxtriangledown$ defined by

$$a \boxtriangledown b = \begin{array}{l} \textbf{if } b \sqsubseteq a \textbf{ then } a \Delta b \\ \textbf{else } a \nabla b \end{array}$$

also called *warrowing*. Solvers which perform chaotic iteration and use warrowing to combine old values with new contributions, necessarily return post solutions — whenever they terminate. In [4,5], termination could only be guaranteed for systems of equations where all right-hand sides are monotonic. For *non-monotonic* systems as may occur at interprocedural analysis, only practical evidence could be provided for the proposed algorithms to terminate in interesting cases.

Here, our goal is to lift these limitations by providing solvers which terminate for all finite abstract systems of equations and all complete lattices — no matter whether right-hand sides are monotonic or not. For that purpose, we dissolve the operator $\boxtriangledown$ again into its components. Instead, we equip the solving routines with extra logic to decide when to apply which operator.

4 Terminating Structured Round-Robin Iteration

Let us consider a finite abstract system as given by:

$$y_i = f_i^\sharp, \qquad i = 1, \ldots, n \tag{6}$$

In [4], a variation of round robin iteration is presented which is guaranteed to terminate for monotonic systems, while it may not terminate for non-monotonic systems. In order to remedy this failure, we re-design this algorithm by additionally maintaining a flag which indicates whether the variable presently under consideration has or has not reached a sound value (Fig. 1). Solving starts with a

```
void solve(b, i) {
    if (i ≤ 0) return;
    solve(b, i − 1);
    tmp := f_i^♯ σ;
    b′ := b;
    if (b) tmp := σ[y_i] Δ tmp;
    else if (tmp ⊑ σ[y_i]) {
        tmp := σ[y_i] Δ tmp;
        b′ := true;
    } else tmp := σ[y_i] ∇ tmp;
    if (σ[y_i] = tmp) then return;
    σ[y_i] := tmp;
    solve(b′, i);
}
```

Fig. 1. Terminating structured round robin iteration.

call solve(**false**, n) where n is the highest priority of a variable. A variable y_i has a higher priority than a variable y_j whenever $i > j$ holds. A call solve(b, i) considers variables up to priority i only. The Boolean argument b indicates whether a sound abstraction (relative to the current values of the higher priority variables) has already been reached. The algorithm first iterates on the lower priority variables (if there are any). Once solving of these is completed, the right-hand side $f_i^\sharp$ of the current variable y_i is evaluated and stored in the variable *tmp*. Additionally, b' is initialized with the Boolean argument b. First assume that b has already the value **true**. Then the old value $\sigma\, y_i$ is combined with the new value in *tmp* by means of the narrowing operator giving the new value of *tmp*. If that is equal to the old value, we are done and solve returns. Otherwise, $\sigma\, y_i$ is updated to *tmp*, and solve(**true**, i) is called tail-recursively. Next assume that b has still the value **false**. Then the algorithm distinguishes two cases. If the old value $\sigma\, y_i$ exceeds the new value, the variable *tmp* receives the combination of both values by means of the narrowing operator. Additionally, b' is set to **true**. Otherwise, the new value for *tmp* is obtained by means of widening. Again, if the resulting value of *tmp* is equal to the current value $\sigma\, y_i$ of y_i, the algorithm returns, whereas if they differ, then $\sigma\, y_i$ is updated to *tmp* and the algorithm recursively calls itself for the actual parameters (n, b'). In light of Theorem 1, the resulting algorithm is called *terminating structured round robin iteration* or TSRR for short.

Theorem 1. *The algorithm in Figure 1 terminates for all finite abstract systems of the form* (6)*. Upon termination, it returns a variable assignment which is*

sound. If all right-hand sides are monotonic, the returned variable assignment is a post solution.

For a proof see Appendix A. In fact, for monotonic systems, the new variation of round robin iteration behaves identical to the algorithm SRR from [4].

5 Local Solvers

Local solving may gain efficiency by querying the value only of a hopefully small subset of variables whose values still are sufficient to answer the initial query. Such solvers are at the heart of program analysis frameworks such as the CIAO system [17,16] or GOBLINT. In order to reason about *partial* variable assignments as computed by local solvers, we can no longer consider right-hand sides in equations as black boxes, but require a notion of *variable dependence*.

For the concrete system we assume that right-hand sides are mathematical functions of type $(X \to \mathbb{C}) \to \mathbb{C}$ where for any such function f and variable assignment $\sigma : X \to \mathbb{C}$, we are given a superset $\mathsf{dep}(f, \sigma)$ of variables onto which f possibly depends, i.e.,

$$(\forall x \in \mathsf{dep}(f,\sigma).\, \sigma[x] = \sigma'[x]) \Longrightarrow f\,\sigma = f\,\sigma' \tag{7}$$

for all $\sigma' : X \to \mathbb{C}$. Let $\sigma : X \to \mathbb{C}$ denote a solution of the concrete system. Then we call a subset $X' \subseteq X$ of variables σ-*closed*, if for all $x \in X'$, $\mathsf{dep}(f_x, \sigma) \subseteq X'$. Then for every x contained in the σ-closed subset X', $\sigma[x]$ can be determined already if the values of σ are known for the variables in X' only.

In [22,23] it is observed that for suitable formulations of interprocedural analysis, the set of all run-time calling contexts of procedures can be extracted from σ-closed sets of variables.

Example 2. The following system may arise from the analysis of a program consisting of a procedure with a loop (signified by the program point u) within which the same procedure is called twice in a row. Likewise, the procedure p iterates on some program point v by repeatedly applying the function g:

$$\begin{aligned}\langle u, q\rangle &= \bigcup \{\, \langle v, q_1\rangle \mid q_1 \in \bigcup \{\, \langle v, q_2\rangle \mid q_2 \in \langle u, q\rangle \,\} \,\} \cup \{q\} \\ \langle v, q\rangle &= \bigcup \{\, g\, q_1 \mid q_1 \in \langle v, q\rangle \,\} \cup \{q\}\end{aligned}$$

for $q \in Q$. Here, Q is a superset of all possible system states, and the unary function $g : Q \to 2^Q$ describes the operational behavior of the body of the loop at v. The set of variables of this system is given by $X = \{\langle u, q\rangle, \langle v, q\rangle \mid q \in Q\}$ where $\langle u, q\rangle, \langle v, q\rangle$ represent the sets of program states possibly occurring at program points u and v, respectively, when the corresponding procedures have been called in context q. For any variable assignment σ, the dependence sets of the right-hand sides are naturally defined by:

$$\begin{aligned}\mathsf{dep}(f_{\langle u,q\rangle}, \sigma) &= \{\langle u, q\rangle\} \cup \{\langle v, q_2\rangle \mid q_2 \in \sigma\,\langle u, q\rangle\} \\ &\qquad \cup \{\langle v, q_1\rangle \mid q_2 \in \sigma\langle u, q\rangle, q_1 \in \sigma\,\langle v, q_2\rangle\} \\ \mathsf{dep}(f_{\langle v,q\rangle}, \sigma) &= \{\langle v, q\rangle\}\end{aligned}$$

where f_x again denotes the right-hand side function for a variable x. Assuming that $g(q_0) = \{q_1\}$ and $g(q_1) = \emptyset$, the least solution σ maps $\langle u, q_0\rangle, \langle v, q_0\rangle$ to the set $\{q_0, q_1\}$ and $\langle u, q_1\rangle, \langle v, q_1\rangle$ to $\{q_1\}$. Accordingly, the set $\{\langle u, q_i\rangle, \langle v, q_i\rangle \mid i = 0, 1\}$ is σ-closed. We conclude, given the program is called with initial context q_0, that the procedure p is called with contexts q_0 and q_1 only. □

In concrete systems of equations, right-hand sides may depend on *infinitely* many variables. Since abstract systems are meant to give rise to effective algorithms, we impose more restrictive assumptions onto their right-hand side functions. For these, we insist that only finitely many variables may be queried. Following the considerations in [19,18,20], we demand that every right-hand side $f^\sharp$ of the abstract system is *pure* in the sense of [19]. This means that, operationally, the evaluation of $f^\sharp$ for any abstract variable assignment $\sigma^\sharp$ consists of a finite sequence of variable look-ups before eventually, a value is returned. Technically, $f^\sharp$ can be represented by a *computation tree*, i.e., is an element of

$$\mathsf{tree} ::= \mathsf{Answer}\ \mathbb{D} \quad | \quad \mathsf{Query}\ Y \times (\mathbb{D} \to \mathsf{tree})$$

Thus, a computation tree either is a leaf immediately containing a value or a query, which consists of a variable together with a continuation which, for every possible value of the variable returns a tree representing the remaining computation. Each computation tree defines a function $[\![t]\!] : (Y \to \mathbb{D}) \to \mathbb{D}$ by:

$$\begin{aligned} [\![\mathsf{Answer}\, d]\!]\, \sigma &= d \\ [\![\mathsf{Query}\, (y, c)]\!]\, \sigma &= [\![c\, (\sigma[y])]\!]\, \sigma \end{aligned}$$

Following [19], the tree representation is uniquely determined by (the operational semantics of) $f^\sharp$.

Example 3. Computation trees can be considered as generalizations of binary decision diagrams to arbitrary sets $\mathbb{D}$. For example, let $\mathbb{D} = \mathbb{N} \cup \{\infty\}$, i.e., the natural numbers (equipped with the natural ordering and extended with ∞ as top element), the function $f^\sharp : (Y \to \mathbb{D}) \to \mathbb{D}$ with $\{y_1, y_2\} \subseteq Y$, defined by

$$f^\sharp\, \sigma = \mathbf{if}\ \sigma[y_1] > 5\ \mathbf{then}\ 1 + \sigma[y_2]\ \mathbf{else}\ \sigma[y_1]$$

is represented by the tree

$$\begin{aligned} \mathsf{Query}\, (y_1, \mathbf{fun}\, d_1 \to \mathbf{if}\ d_1 > 5\ \mathbf{then}\ \mathsf{Query}\, (y_2, \mathbf{fun}\, d_2 \to \mathsf{Answer}\, (1 + d_2)) \\ \mathbf{else}\ \mathsf{Query}\, (y_1, \mathbf{fun}\, d_1 \to \mathsf{Answer}\, d_1)) \end{aligned}$$ □

A set $\mathsf{dep}(f^\sharp, \sigma^\sharp) \subseteq Y$ with a property analogous to (7) can be explicitly obtained from the tree representation t of $f^\sharp$ by defining $\mathsf{dep}(f^\sharp, \sigma^\sharp) = \mathsf{treedep}(t, \sigma^\sharp)$ where:

$$\begin{aligned} \mathsf{treedep}(\mathsf{Answer}\, d, \sigma^\sharp) &= \emptyset \\ \mathsf{treedep}(\mathsf{Query}\, (y, c), \sigma^\sharp) &= \{y\} \cup \mathsf{treedep}(c\, (\sigma^\sharp[y]), \sigma^\sharp) \end{aligned}$$

Technically, this means that the value $f^\sharp\, \sigma^\sharp = [\![t]\!]\, \sigma^\sharp$ can be computed already for *partial* variable assignments $\sigma' : Y' \to \mathbb{D}$, whenever $\mathsf{dep}(f^\sharp, \underline{\top} \oplus \sigma') = \mathsf{treedep}(t, \underline{\top} \oplus \sigma') \subseteq Y'$. Here, $\underline{\top} : Y \to \mathbb{D}$ maps each variable of Y to $\top$ and $\underline{\top} \oplus \sigma'$ returns the value $\sigma'[y]$ for every $y \in Y'$ and $\top$ otherwise.

Example 4. Consider the function $f^\sharp$ from Example 3 together with the partial assignment $\sigma' = \{y_1 \mapsto 3\}$. Then $\mathsf{dep}(f^\sharp, \underline{\top} \oplus \sigma') = \{y_1\}$. □

We call a partial variable assignment $\sigma' : Y' \to \mathbb{D}$ *closed* (w.r.t. an abstract system (2)), if for all $y \in Y'$, $\mathsf{dep}(f_y^\sharp, \underline{\top} \oplus \sigma') \subseteq Y'$.

In the following, we strengthen the description relation $\mathcal{R}$ to a description relation $\mathcal{R}_{\mathsf{dep}}$ which additionally takes variable dependencies into account. We say that $f\ \mathcal{R}_{\mathsf{dep}}\ f^\sharp$ for a concrete function f and an abstract function $f^\sharp$, if $f\ \mathcal{R}\ f^\sharp$ and $\mathsf{dep}(f, \sigma)\ \mathcal{R}\ \mathsf{dep}(f^\sharp, \sigma^\sharp)$ whenever $\sigma\ \mathcal{R}\ \sigma^\sharp$ holds. Here, a pair of sets X', Y' of concrete and abstract variables is in relation $\mathcal{R}$ if for all $x \in X'$, $x\ \mathcal{R}\ y$ for some $y \in Y'$. Moreover, we say that the abstract system (2) *simulates* the concrete system (1) (relative to the description relation $\mathcal{R}$) iff for all pairs x, y of variables with $x\ \mathcal{R}\ y$, $f_x\ \mathcal{R}_{\mathsf{dep}}\ f_y^\sharp$ holds. Theorem 2 demonstrates the significance of closed abstract assignments which are sound.

Theorem 2. *Assume that the abstract system* (2) *simulates the concrete system* (1) *(relative to $\mathcal{R}$) where σ is the least solution of the concrete system. Assume that $\sigma^\sharp : Y' \to \mathbb{D}$ is a partial assignment with the following properties:*

1. *$\sigma^\sharp$ is closed;*
2. *$\underline{\top} \oplus \sigma^\sharp$ is a post solution of the lower monotonization of the abstract system.*

Then the set $X' = \{\, x \in X \mid \exists\, y \in Y'.\, x\ \mathcal{R}\ y \,\}$ is σ-closed.

Proof. By Proposition 1, $\sigma\ \mathcal{R}\ (\underline{\top} \oplus \sigma^\sharp)$ holds. Now assume that $x\ \mathcal{R}\ y$ for some $y \in Y'$. By definition therefore, $\mathsf{dep}(f_x, \sigma)\ \mathcal{R}\ \mathsf{dep}(f_y^\sharp, \underline{\top} \oplus \sigma^\sharp)$. Since the latter is a subset of Y', the former must be a subset of X', and the assertion follows. □

6 Terminating Structured Two-Phase Solving

We first present a local version of a two-phase algorithm to determine a sound variable assignment for an abstract system of equations. As the algorithm is local, no pre-processing of the equation system is possible. Accordingly, variables where widening or narrowing is to be applied must be determined dynamically. We solve this problem by assigning *priorities* to variables in decreasing order in which they are encountered, and consider a variable as a candidate for widening/narrowing whenever it is queried during the evaluation of a lower priority variable. The second issue is that during the narrowing iteration of the second phase, variables may be encountered which have not yet been seen and for which therefore no sound approximation is available. In order to deal with this situation, the algorithm does not maintain a single variable assignment, but two distinct ones. While assignment σ_0 is used for the widening phase, σ_1 is used for narrowing with the understanding that, once the widening phase is completed, the value of a variable y from σ_0 is copied as the initial value of y into σ_1. This clear distinction allows to continue the widening iteration for every newly encountered variable y' in order to determine an acceptable initial value before continuing with the narrowing iteration. The resulting algorithm can be found in Figures 2

```
void iterate_0(n) {
    if (Q ≠ ∅ ∧ min_prio(Q) ≤ n) {
        y := extract_min(Q);

        do_var_0(y);
        iterate_0(n);
    }
}
void solve_0(y) {
    if (y ∈ dom_0) return;
    dom_0 := dom_0 ∪ {y};
    prio[y] := next_prio();
    σ_0[y] := ⊥;
    infl[y] := ∅;
    do_var_0(y);
    iterate_0(prio[y]);
}
```

```
void iterate_1(n) {
    if (Q ≠ ∅ ∧ min_prio(Q) ≤ n) {
        y := extract_min(Q);
        solve_1(y, prio[y] − 1);
        do_var_1(y);
        iterate_1(n);
    }
}
void solve_1(y, n) {
    if (y ∈ dom_1) return;
    solve_0(y);
    dom_1 := dom_1 ∪ {y};
    σ_1[y] := σ_0[y];
    forall (z ∈ {y} ∪ infl[y]) insert z Q;
    infl[y] := ∅;
    iterate_1(n);
}
```

Fig. 2. The solver TSTP, part 1.

```
void do_var_0(y) {
    isp := y ∈ point;
    point := point\{y};
    𝔻 eval_0(z) {
        solve_0(z);
        if (prio[z] ≥ prio[y]) point := point ∪ {z};
        infl[z] := infl[z] ∪ {y};
        return σ_0[z]);
    }
    tmp := f_y^♯ eval_0;
    if (isp) tmp := σ_0[y] ∇ tmp;
    if (σ_0[y] = tmp) return;
    σ_0[y] := tmp;
    forall (z ∈ infl[y]) insert z Q;
    infl[y] := ∅;
    return;
}
```

```
void do_var_1(y) {
    isp := y ∈ point;
    point := point\{y};
    𝔻 eval_1(z) {
        solve_1(z, prio[y] − 1);
        if (prio[z] ≥ prio[y]) point := point ∪ {z};
        infl[z] := infl[z] ∪ {y};
        return σ_1[z];
    }
    tmp := f_y^♯ eval_1;
    if (isp) tmp := σ[y] Δ tmp;
    if (σ_1[y] = tmp) return;
    σ_1[y] := tmp;
    forall (z ∈ infl[y]) insert z Q;
    infl[y] := ∅;
    return;
}
```

Fig. 3. The solver TSTP, part 2.

and 3. Initially, the priority queue Q and the sets dom_0 and dom_1 are empty. Accordingly, the mappings $\sigma_i : \mathsf{dom}_i \to \mathbb{D}$ and $\mathsf{infl} : \mathsf{dom} \to 2^Y$ are also empty. Likewise, the set point is initially empty. Solving for the variable y_0 starts with the call $\mathsf{solve}_1(y_0, 0)$.

Let us first consider the functions $\mathsf{solve}_0, \mathsf{iterate}_0, \mathsf{do\_var}_0$. These are meant to realize a local widening iteration. A call $\mathsf{solve}_0(y)$ first checks whether $y \in \mathsf{dom}_0$. If this is the case, solving immediately terminates. Otherwise, $\sigma_0[y]$ is initialized with $\bot$, y is added to dom_0, the empty set is added to $\mathsf{infl}[y]$, and y receives the next available priority by means of the call $\mathsf{next\_prio}$. Subsequently, $\mathsf{do\_var}_0(y)$ is called, followed by a call to $\mathsf{iterate}_0(\mathsf{prio}[y])$ to complete the widening phase for y. Upon termination, a call $\mathsf{iterate}_0(n)$ for an integer n has removed all variables of priority at most n from the queue Q. It proceeds as follows. If Q is empty or contains only variables of priority exceeding n, it immediately returns. Otherwise, the variable y with least priority is extracted from Q. Having processed $\mathsf{do\_var}_0(y)$, the iteration continues with the tail-recursive call $\mathsf{iterate}_0(n)$.

It remains to describe the function $\mathsf{do\_var}_0$. When called for a variable y, the algorithm first determines whether or not y is a widening/narrowing point, i.e., contained in the set point. If so, y is removed from point, and the flag *isp* is set to **true**. Otherwise, *isp* is set to **false**. Then the right-hand side $f_y^\sharp$ is evaluated and the result stored in the variable *tmp*. For its evaluation, the function $f_y^\sharp$, however, does not receive the current variable assignment σ_0 but an auxiliary function eval_0 which serves as a wrapper to the assignment σ_0. The wrapper function eval_0, when queried for a variable z, first calls $\mathsf{solve}_0\,(z)$ to compute a first non-trivial value for z. If the priority of z is greater or equal to the priority of y, a potential widening point is detected. Therefore, z is added to the set point. Subsequently, the fact that z was queried during the evaluation of the right-hand side of y, is recorded by adding y to the set $\mathsf{infl}[z]$. Finally, $\sigma_0[z]$ is returned.

Having evaluated $f_y^\sharp\,\mathsf{eval}_0$ and stored the result in *tmp*, the function $\mathsf{do\_var}_0$ then applies widening only if *isp* equals **true**. In this case, *tmp* receives the value of $\sigma[y] \nabla \mathit{tmp}$. In the next step, *tmp* is compared with the current value $\sigma_0[y]$. If both values are equal, the procedure returns. Otherwise, $\sigma_0[y]$ is updated to *tmp*. The variables in $\mathsf{infl}[y]$ are inserted into the queue Q, and the set $\mathsf{infl}[y]$ is reset to the empty set. Only then the procedure returns.

The functions solve_1, $\mathsf{iterate}_1$ and $\mathsf{do\_var}_1$, on the other hand, are meant to realize the narrowing phase. They essentially work analogously to the corresponding functions solve_0, $\mathsf{iterate}_0$ and $\mathsf{do\_var}_0$. In particular, they re-use the mapping infl which records the currently encountered variable dependencies as well as the variable priorities and the priority queue Q. Instead of σ_0, dom_0, however, they now refer to σ_1, dom_1, respectively. Moreover, there are the following differences.

First, the function solve_1 now receives not only a variable, but a pair of an integer n and a variable y. When called, the function first checks whether $y \in \mathsf{dom}_1$. If this is the case, solving immediately terminates. Otherwise, $\mathsf{solve}_0(y)$ is called first. After that call, the widening phase for y is assumed to have terminated where the resulting value is $\sigma_0[y]$. Accordingly, $\sigma_1[y]$ is initialized with $\sigma_0[y]$, and y is added to dom_1. As the value of σ_1 for y has been updated, y

together with all variables in $\mathsf{infl}[y]$ are added to the queue, whereupon $\mathsf{infl}[y]$ is set to the empty set, and $\mathsf{iterate}_1(n)$ is called to complete the narrowing phase up to the priority n. Upon termination, a call $\mathsf{iterate}_1(n)$ for an integer n has removed all variables of priority at most n from the queue Q. In distinction to $\mathsf{iterate}_0$, however, it may extract variables y from Q which have not yet been encountered in the present phase of iteration, i.e., are not yet included in dom_1 and thus have not yet received a value in σ_1. To ensure initialization, $\mathsf{solve}_1(y, n)$ is called for $n = \mathsf{prio}[y] - 1$. This choice of the extra parameter n ensures that all lower priority variables have been removed from Q before $\mathsf{do\_var}_1(y)$ is called.

It remains to explain the function $\mathsf{do\_var}_1(y)$. Again, it essentially behaves like $\mathsf{do\_var}_0(y)$ — with the distinction that the narrowing operator is applied instead of the widening operator. Furthermore, the auxiliary local function eval_0 is replaced with eval_1 which now uses a call to solve_1 for the initialization of its argument variable z (instead of solve_0) where the extra integer argument is given by $\mathsf{prio}[y] - 1$, i.e., an iteration is performed to remove all variables from Q with priorities lower than the priority of y (not of z).

In light of Theorem 3, we call the algorithm from Figures 2 and 3 *terminating structured two-phase solver*.

Theorem 3. *The local solver* TSTP *from Figure 2 and 3 when started with a call* $\mathsf{solve}_1(y_0, 0)$ *for a variable* y_0*, terminates for every system of equations whenever only finitely many variables are encountered.*

Upon termination, assignments $\sigma_i^\sharp : Y_i \to \mathbb{D}$*,* $i = 0, 1$ *are obtained for finite sets* $Y_0 \supseteq Y_1$ *of variables so that the following holds:*

1. $y_0 \in Y_1$*;*
2. $\sigma_0^\sharp$ *is a closed partial post solution of the abstract system* (2)*;*
3. $\sigma_1^\sharp$ *is a closed partial assignment such that* $\underline{\bot} \oplus \sigma_1^\sharp$ *is a post solution of the lower monotonization of the abstract system* (2)*.*

For a proof see Appendix B.

7 Terminating Structured Mixed-Phase Solving

The draw-back of the two-phase solver TSTP from the last section is that it may lose precision already in very simple situations.

Example 5. Consider the system:

$$y_1 = \mathsf{max}(y_1, y_2) \qquad y_2 = \mathsf{min}(y_3, 2) \qquad y_3 = y_2 + 1$$

over the complete lattice $\mathbb{N}^\infty$ and the following widening and narrowing operators:

$$a \nabla b = \textbf{if } a < b \textbf{ then } \infty \textbf{ else } a$$
$$a \Delta b = \textbf{if } a = \infty \textbf{ then } b \textbf{ else } a$$

Then $\mathsf{solve}_0(y_1)$ detects y_2 as the only widening point resulting in

$$\sigma_0 = \{y_1 \mapsto \infty, y_2 \mapsto \infty, y_3 \mapsto \infty\}$$

A call to $\mathsf{solve}_1(y_1, 0)$ therefore initializes y_1 with ∞ implying that $\sigma_1[y_1] = \infty$ irrespective of the fact that $\sigma_1[y_2] = 2$. □

We may therefore aim at intertwining the two phases into one — without sacrificing the termination guarantee. The idea is to operate on a single variable assignment only and iterate on each variable first in widening and then in narrowing mode. In order to keep soundness, after every update of a variable y in the widening phase, all possibly influenced lower priority variables are iterated upon until all stabilize with widening and narrowing. Only then the widening iteration on y continues. If on the other hand an update for y occurs during narrowing, the iteration on possibly influenced lower priority variables is with narrowing only. The distinction between the two modes of the iteration is maintained by a flag where **false** and **true** correspond to the widening and narrowing phases, respectively. The algorithm is provided in Figure 4 and 5.

```
void iterate(b, n) {
    if (Q ≠ ∅ ∧ min_prio(Q) ≤ n) {
        y := extract_min(Q);
        b' := do_var(b, y);
        n' := prio[y];
        if (b ≠ b' ∧ n > n') {
            iterate(b', n');
            iterate(b, n);
        } else iterate(b', n);
    }
}
```

```
void solve(y) {
    if (y ∈ dom) return;
    dom := dom ∪ {y};
    prio[y] := next_prio();
    σ[y] := ⊥;
    infl[y] := ∅;
    b' := do_var(false, y);
    iterate(b', prio[y]);
}
```

Fig. 4. The solver TSMP, part 1.

Initially, the priority queue Q and the set dom are empty. Accordingly, the mappings $\sigma : \mathsf{dom} \to \mathbb{D}$ and $\mathsf{infl} : \mathsf{dom} \to Y$ are also empty. Likewise, the set point is initially empty. Solving for the variable y_0 starts with the call $\mathsf{solve}(y_0)$. Solving for some variable y first checks whether $y \in \mathsf{dom}$. If this is the case, solving immediately terminates. Otherwise, y is added to dom and receives the next available priority by means of a call to next_prio. That call should provide a values which is less than any priority of a variable in dom. Subsequently, the entries $\sigma[y]$ and $\mathsf{infl}[y]$ are initialized to $\bot$ and the empty set, respectively, and do_var is called for the pair $(\mathbf{false}, y)$. The return value of this call is stored in the Boolean variable b. During its execution, this call may have inserted further variables into the queue Q. These are dealt with by the call $\mathsf{iterate}(b, \mathsf{prio}(y))$.

Upon termination, a call $\mathsf{iterate}(b, n)$ has removed all variables of priority at most n from the queue Q. It proceeds as follows. If Q is empty or contains only variables of priority exceeding n, it immediately returns. Otherwise, the variable

```
bool do_var(b, y) {
    isp := y ∈ point;
    point := point\{y};
    𝔻 eval(z) {
        solve(z);
        if (prio[z] ≥ prio[y]) point := point ∪ {z};
        infl[z] := infl[z] ∪ {y};
        return σ[z];
    }
    tmp := f♯_y eval;
    b′ := b;
    ...
    ...
    if (isp)
        if (b) tmp := σ[y] Δ tmp;
        else if (tmp ⊑ σ[y]) {
            tmp := σ[y] Δ tmp;
            b′ := true;
        } else tmp := σ[y] ∇ tmp;
    if (σ[y] = tmp) return true;
    σ[y] := tmp;
    forall (z ∈ infl[y]) insert z Q;
    infl[y] := ∅;
    return b′;
}
```

Fig. 5. The solver TSMP, part 2.

y with least priority n' is extracted from Q. For (b, y), do_var is called and the return value of this call is stored in b'.

Now we distinguish several cases. If $b = \textbf{true}$, then the value b' returned by do_var will necessarily be **true** as well. In that case, iteration proceeds by tail-recursively calling again iterate(**true**, n). If on the other hand $b = \mathsf{false}$, then the value b' returned by do_var can be either **true** or **false**. If $b' = \textbf{false}$ or $b' = \textbf{true}$ and $n' = n$, then iterate(b', n) is tail-recursively called. If, however, $b' = \textbf{true}$ and $n > n'$, then first a sub-iteration is triggered for (**true**, n') before the main loop proceeds with the call iterate(**false**, n).

It remains to describe the function do_var. When called for a pair (b, y) consisting of a Boolean value b and variable y, the algorithm first determines whether or not y is a widening/narrowing point, i.e., contained in the set point. If so, y is removed from point, and the flag *isp* is set to **true**. Otherwise, *isp* is just set to **false**. Then the right-hand side $f^\sharp_y$ is evaluated and the result stored in the variable *tmp*. For its evaluation, the function $f^\sharp_y$, however, does not receive the current variable assignment σ, but an auxiliary function eval which serves as a wrapper to σ. The wrapper function eval, when queried for a variable z, first calls solve z to compute a first non-trivial value for z. If the priority of z exceeds or is equal to the priority of y, a potential widening/narrowing point is detected. Therefore, z is added to the set point. Subsequently, the fact that the value of z was queried during the evaluation of the right-hand side of y, is recorded by adding y to the set infl$[z]$. Finally, the value $\sigma[z]$ is returned.

Having evaluated $f^\sharp_y$ eval and stored the result in *tmp*, the function do_var then decides whether to apply widening or narrowing or none of them according to the following scheme. If *isp* has not been set to **true**, no widening or narrowing is applied. In this case, the flag b' receives the value b. Therefore now consider the case $isp = \textbf{true}$. Again, the algorithm distinguishes three cases. If $b = \textbf{true}$, then necessarily narrowing is applied, i.e., *tmp* is updated to the value of $\sigma[y] \,\Delta\, tmp$,

and b' still equals b, i.e., **true**. If $b = \mathbf{false}$ then narrowing is applied whenever $tmp \sqsubseteq \sigma[y]$ holds. In that case, tmp is set to $\sigma[y] \,\Delta\, tmp$, and b' to **true**. Otherwise, i.e., if $b = \mathbf{false}$ and $tmp \not\sqsubseteq \sigma\, y$, then widening is applied by setting tmp to $\sigma[y] \,\nabla\, tmp$, and b' obtains the value **false**.

In the next step, tmp is compared with the current value $\sigma[y]$. If both values are equal, the value of b' is returned. Otherwise, $\sigma[y]$ is updated to tmp. The variables in $\mathsf{infl}[y]$ are inserted into the queue Q, and the set $\mathsf{infl}[y]$ is reset to the empty set. Only then the value of b' is returned.

Example 6. Consider the system of equations from Example 5. Calling solve for variable y_1 will assign the priorities $0, -1, -2$ to the variables y_1, y_2 and y_3, respectively. Evaluation of the right-hand side of y_1 proceeds only after $\mathsf{solve}(y_2)$ has terminated. During the first update of y_2, y_2 is inserted into the set point, implying that at the subsequent evaluation the widening operator is applied resulting in the value ∞ for y_2 and y_3. The subsequent narrowing iteration on y_2 and y_3 improves these values to 2 and 3, respectively. Only then the value for y_1 is determined which is 2. During that evaluation, y_1 has also been added to the set point. The repeated evaluation of its right-hand side, will however, again produce the value 2 implying that the iteration terminates with the assignment

$$\sigma = \{y_1 \mapsto 2, y_2 \mapsto 2, y_3 \mapsto 3\} \qquad \square$$

In light of Theorem 4, we call the algorithm from Figures 4 and 5, *terminating structured mixed-phase solver* or TSMP for short.

Theorem 4. *The local solver* TSMP *from Figure 4 and 5 when started for a variable y_0, terminates for every system of equations whenever only finitely many variables are encountered.*

Upon termination, an assignment $\sigma^\sharp : Y_0 \to \mathbb{D}$ is returned where Y_0 is the set of variables encountered during $\mathsf{solve}(\mathbf{false}, y_0)$ *such that the following holds:*

- $y_0 \in Y_0$,
- $\sigma^\sharp$ *is a closed partial assignment such that* $\bot \oplus \sigma^\sharp$ *is a post solution of the lower monotonization of the abstract system* (2).

For a proof see Appendix C.

8 Interprocedural Analysis

In principle, interprocedural analysis can be cast as an abstract system of equations for a set of variables $Y = \{\langle u, d\rangle \mid u \in U, d \in \mathbb{D}\}$ where U is a finite set of program points which meant to be analyzed for various abstract calling contexts $d \in \mathbb{D}$. Right-hand sides for these variables are given by expressions e according to the following grammar:

$$e \quad ::= \quad d \mid \alpha \mid g^\sharp\, e_1 \cdots e_k \mid \langle u, e\rangle$$

where $d \in \mathbb{D}$ denotes arbitrary constants, α is a dedicated variable representing the current calling context, $g^\sharp : \mathbb{D} \to \cdots \to \mathbb{D}$ is a k-ary function, and $\langle u, e\rangle$ with $u \in U$ refers to a variable of the equation system. Each expression e describes a function $[\![e]\!]^\sharp : \mathbb{D} \to (Y \to \mathbb{D}) \to \mathbb{D}$ which is defined by:

$$\begin{array}{ll} [\![d]\!]^\sharp\, a\, \sigma & = d \\ [\![\alpha]\!]^\sharp\, a\, \sigma & = a \\ [\![g^\sharp\, e_1 \cdots e_k]\!]^\sharp\, a\, \sigma & = g^\sharp\, ([\![e_1]\!]^\sharp\, a\, \sigma) \cdots ([\![e_k]\!]^\sharp\, a\, \sigma) \\ [\![\langle u, e\rangle]\!]^\sharp\, a\, \sigma & = \sigma\, \langle u, [\![e]\!]^\sharp\, a\, \sigma\rangle \end{array}$$

A finite representation of the abstract system of equations then is given by the finite set of schematic equations

$$\langle u, \alpha\rangle = e_u, \qquad u \in U$$

for expressions e_u. Each schematic equation $\langle u, \alpha\rangle = e_u$ denotes the (possibly infinite) family of equations for the variables $\langle u, a\rangle, a \in \mathbb{D}$. For each $a \in \mathbb{D}$, the right-hand side function of $\langle u, a\rangle$ is given by the function $[\![e_u]\!]^\sharp\, a$. This function is indeed pure for every expression e_u and every $a \in \mathbb{D}$. Such systems of equations have been used, e.g., in [10,3] to specify interprocedural analyses.

Example 7. Consider the schematic system:

$$\langle u, \alpha\rangle = \langle v, \langle v, \langle u, \alpha\rangle\rangle\rangle \sqcup \alpha \qquad \langle v, \alpha\rangle = g^\sharp\, \langle v, \alpha\rangle \sqcup \alpha$$

for some unary function $g^\sharp : \mathbb{D} \to \mathbb{D}$. A correspondence between the resulting abstract system of equations and the concrete system from Example 2 can be established as follows. First, we require a Galois connection between $\mathbb{C} = 2^Q$ and $\mathbb{D}$. Moreover, $g(q) \subseteq \gamma(g^\sharp(a))$ should hold whenever $q \in \gamma(a)$. The system from Example 2 then is simulated by the abstract system w.r.t. the description relation $\mathcal{R}$ between concrete and abstract variables given by $\langle u, q\rangle\ \mathcal{R}\ \langle u, a\rangle$ and $\langle v, q\rangle\ \mathcal{R}\ \langle v, a\rangle$ whenever $q \in \gamma(a)$ holds. □

As we have seen in the example, function calls result in indirect addressing via nesting of variables. In case that the program does not have recursive procedures, there is a mapping $\lambda : U \to \mathbb{N}$ so that for every u with current calling context α, right-hand side e_u and every subexpression $\langle u', e'\rangle$ of e_u the following holds:

- If $\lambda(u') = \lambda(u)$, then $e' = \alpha$;
- If $\lambda(u') \neq \lambda(u)$, then $\lambda(u') < \lambda(u)$.

If this property is satisfied, we call the equation scheme *stratified* where $\lambda(u)$ is the *level* of u. Intuitively, stratification means that a new context is created only for some point u' of a strictly lower level. For the interprocedural analysis as formalized, e.g., in [3], all program points of a given procedure may receive the same level while the level decreases whenever another procedure is called. The system from Example 7 is stratified: we may, e.g., define $\lambda(u) = 2$ and $\lambda(v) = 1$.

Theorem 5. *The solver* TSTP *as well as the solver* TSMP *terminate for stratified equation schemes.*

Proof. We only consider the statement of the theorem for solver TSMP. Assume we run the solver TSMP on an abstract system specified by a stratified equation scheme. In light of Theorem 4, it suffices to prove that for every $u \in U$, only finitely many contexts $a \in \mathbb{D}$ are encountered during fixpoint computation. First, we note that variables $\langle v, a\rangle$ may not be influenced by variables $\langle u, a'\rangle$ with $\lambda(u) > \lambda(v)$. Second, let $D_{v,a}$ denote the set of variables $\langle u, a'\rangle$ with $\lambda(u) = \lambda(v)$ onto which $\langle v, a'\rangle$ may depend. Then all these variables share the same context. We conclude that new contexts for a point v at some level k are created only by the evaluation of right-hand sides of variables of smaller levels. For each level k, let $U_k \subseteq U$ denote the set of all u with $\lambda(u) \leq k$. We proceed by induction on k. Assume that we have proven termination for all calls $\mathsf{solve}\,\langle u, a'\rangle$, $\lambda(u) < k$ for any subset of variables $\langle u', a''\rangle$ which have already been solved. Then evaluating a call $\mathsf{solve}\,\langle v, a\rangle$ with $\lambda(v) = k$ will either query the values of other variables $\langle v', a'\rangle$ where $\lambda(v') = k$. In this case, $a' = a$. Therefore, only finitely many of these are encountered. Or variables $\langle v', a'\rangle$ are queried with $\lambda(v') < k$. For those which have not yet been encountered $\mathsf{solve}\,\langle v', a'\rangle$ is called. By induction hypothesis, all these calls terminate and therefore query only finitely many variables. As the evaluation of $\mathsf{call}\,\langle v, a\rangle$ encounters only finitely many variables, it terminates. □

A similar argument explains why interprocedural analyzers based on the functional approach of Sharir/Pnueli [24,1] terminate not only for finite domains but also for full constant propagation — if only the programs are non-recursive.

9 Conclusion

We have presented local solvers which are guaranteed to terminate for all abstract systems of equations given that only finitely many variables are encountered — irrespective of whether right-hand sides of the equations are monotonic or not or whether the complete lattice has infinite strictly ascending/descending chains or not. Furthermore, we showed that interprocedural analysis with partial tabulation of procedure summaries based on these solvers is guaranteed to terminate with the only assumption that the program has no recursive procedures. Clearly, theoretical termination proofs may only give an indication that the proposed algorithms are well-suited as fixpoint engines within a practical analysis tool. Termination within reasonable time and space bounds is another issue. The numbers provided by our preliminary practical experiments within the analysis framework Goblint seem encouraging (see Appendix D). Interestingly, a direct comparison of the two-phase versus mixed-phase solver for full context-sensitive interprocedural analysis, indicated that TSMP was virtually always faster, while the picture w.r.t. precision is not so clear. Also, the new solvers always returned post-solutions of the abstract systems — although they are not bound to do so.

There are several ways how this work can be extended. Our techniques crucially require a Galois connection to relate the concrete with the abstract domain. It is not clear how this restriction can be lifted. Also one may think of extending two phased approaches to a many-phase iteration as suggested in [8].

References

1. M. Alt and F. Martin. Generation of efficient interprocedural analyzers with PAG. In *Second International Symposium on Static Analysis (SAS'95)*, pages 33–50. Springer-Verlag, LNCS 983, 1995.
2. G. Amato, F. Scozzari, H. Seidl, K. Apinis, and V. Vojdani. Efficiently intertwining widening and narrowing. *Science of Computer Programming*, 2016.
3. K. Apinis, H. Seidl, and V. Vojdani. Side-effecting constraint systems: A swiss army knife for program analysis. In *Programming Languages and Systems - 10th Asian Symposium (APLAS)*, pages 157–172. Springer, LNCS 7705, 2012.
4. K. Apinis, H. Seidl, and V. Vojdani. How to combine widening and narrowing for non-monotonic systems of equations. In *34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 377–386. ACM, 2013.
5. K. Apinis, H. Seidl, and V. Vojdani. Enhancing top-down solving with widening and narrowing. In *Semantics, Logics, and Calculi - Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays*, pages 272–288. Springer, LNCS 9560, 2016.
6. R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. *Science of Computer Programming*, 58(1-2):28–56, 2005.
7. L. Chen, A. Miné, J. Wang, and P. Cousot. An abstract domain to discover interval linear equalities. In *Verification, Model Checking, and Abstract Interpretation, 11th International Conference (VMCAI)*, pages 112–128. Springer, LNCS 5944, 2010.
8. P. Cousot. Abstracting induction by extrapolation and interpolation. In *Verification, Model Checking, and Abstract Interpretation - 16th International Conference (VMCAI)*, pages 19–42. Springer, LNCS 8931, 2015.
9. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM, 1977.
10. P. Cousot and R. Cousot. Static determination of dynamic properties of generalized type unions. In *ACM Conference on Language Design for Reliable Software (LDRS)*, pages 77–94. ACM, 1977.
11. P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In *IFIP Conf. on Formal Description of Programming Concepts*, pages 237–277. North-Holland, 1977.
12. P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.
13. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does astrée scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.
14. C. Fecht and H. Seidl. An even faster solver for general systems of equations. In *Static Analysis, Third International Symposium (SAS)*, pages 189–204. Springer, LNCS 1145, 1996.
15. L. Gonnord and N. Halbwachs. Combining widening and acceleration in linear relation analysis. In *Static Analysis, 13th International Symposium (SAS)*, pages 144–160. Springer, LNCS 4134, 2006.
16. M. V. Hermenegildo, F. Bueno, M. Carro, P. López-García, E. Mera, J. F. Morales, and G. Puebla. An overview of ciao and its design philosophy. *Theory Pract. Log. Program.*, 12(1-2):219–252, 2012.

17. M. V. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated program debugging, verification, and optimization using abstract interpretation (and the ciao system preprocessor). *Science of Computer Programming*, 58(1-2):115–140, 2005.
18. M. Hofmann, A. Karbyshev, and H. Seidl. Verifying a local generic solver in coq. In *Static Analysis, 17th International Symposium (SAS)*, pages 340–355. Springer, LNCS 6337, 2010.
19. M. Hofmann, A. Karbyshev, and H. Seidl. What is a pure functional? In *37th International Colloquium Conference on Automata, Languages and Programming (ICALP)*, pages 199–210. Springer, LNCS 6199, 2010.
20. A. Karbyshev. *Monadic Parametricity of Second-Order Functionals*. PhD thesis, Institut für Informatik, Technische Universität München, September 2013.
21. H. M. MacNeille. Partially ordered sets. *Trans. Amer. Math. Soc.*, 42(3):416–460, 1937.
22. K. Muthukumar and M. V. Hermenegildo. Deriving a fixpoint computation algorithm for top-down abstract interpretation of logic programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, April 1990.
23. H. Seidl and C. Fecht. Interprocedural analyses: A comparison. *Journal of Logic Programming*, 43(2):123–156, 2000.
24. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Application*, pages 189–233. Prentice-Hall, 1981.

A Proof of Theorem 1

By induction on i, we prove that $\mathsf{solve}(b, i)$ terminates. For $i = 0$, the statement is obviously true. Now assume that $i > 0$ and that by induction hypothesis, $\mathsf{solve}(b', i-1)$ terminates for $b' \in \{\mathbf{true}, \mathbf{false}\}$. First consider the case where $b = \mathbf{true}$. In this case, the flag b' for the tail-recursive will be equal to **true** as well, and only narrowing will be applied to y_i. Therefore, the sequence of tail-recursive calls $\mathsf{solve}(b, i)$ eventually will terminate. Now consider the case where $b = \mathbf{false}$. By induction hypothesis, all recursive calls $\mathsf{solve}(b', i-1)$ terminate. Consider the sequence of tail recursive calls where the flag b' is not set to **true**. Within this sequence, the new values for y_i form an ascending chain $d_0 \sqsubset d_1 \sqsubset d_2 \ldots$ where $d_{j+1} = d_j \nabla a_j$ for suitable values a_j. Due to the properties of a widening operator, this sequence is finite, i.e., there is some j such that $d_{j+1} \sqsubseteq d_j$. In this case the call either terminates directly or recursively calls $\mathsf{solve}(b', i)$ for $b' = \mathbf{true}$. Therefore, solving terminates also in this case.

It remains to prove that upon termination, a sound variable assignment σ is found. For $j = 1, \ldots, n$, and a variable assignment $\rho : \{y_1, \ldots, y_n\} \to \mathbb{D}$, we consider the system $\mathcal{E}_{\rho,j}$ defined by:

$$y_i = f^\sharp_{\rho,i} \qquad (i = 1, \ldots, j)$$

with $f^\sharp_{\rho,i}\, \sigma^\sharp = f_i(\rho \oplus \sigma^\sharp)$ for $\sigma^\sharp : \{y_1, \ldots, y_j\} \to \mathbb{D}$. Here, the operator $\oplus$ is meant to overwrite the values of ρ by the corresponding values of $\sigma^\sharp$ whenever $\sigma^\sharp$ is defined. Let $\underline{\mathcal{E}}_{\rho,j}$ denote the lower monotonization of $\mathcal{E}_{\rho,j}$. We claim:

1. Assume σ_1 is a post solution of the system $\underline{\mathcal{E}}_{\rho,j}$. Then $\mathsf{solve}(\mathbf{true}, j)$ when started with $\rho_1 = \rho \oplus \sigma_1$, returns with a variable assignment $\rho_2 = \rho \oplus \sigma_2$ where σ_2 is still a post solution of $\underline{\mathcal{E}}_{\rho,j}$.
2. $\mathsf{solve}(\mathbf{false}, j)$ returns a post solution of $\underline{\mathcal{E}}_{\rho,j}$.

We proceed by induction on j. For $j = 0$, nothing must be proven. Therefore assume $j > 0$. Consider the first claim. As post solutions of $\underline{\mathcal{E}}_{\rho,j}$ are preserved by each update which combines an old value $\sigma(y_i)$ with the value of the corresponding right-hand side $f^\sharp_{\rho,i}$ for σ by means of $\sqcap$ and thus also by Δ, the claim follows.

For a proof of the second claim, let us consider the sub-sequence of tail-recursive calls $\mathsf{solve}(b, j)$ where b' remains false. Eventually this sequence ends with a last call within which b' is set to $\mathbf{true}$. Let ρ' denote the variable assignment before this update occurs. Then $\rho'(y_j) \sqsupseteq f_j\,\rho' \sqsupseteq \underline{f}_j\,\rho'$. Likewise, by induction hypothesis, $\rho'|_{\{y_1,\ldots,y_{j-1}\}}$ is a post solution of $\underline{\mathcal{E}}_{\rho',j-1}$. Altogether therefore, $\rho' = \rho \oplus \sigma'$ for some variable assignment $\sigma' : \{y_1, \ldots, y_j\} \to \mathbb{D}$ which is a post solution of $\underline{\mathcal{E}}_{\rho',j}$. Accordingly, $\mathsf{solve}(\mathbf{false}, j)$ either directly terminates with ρ', and the second claim follows, or $\mathsf{solve}(\mathbf{true}, j)$ is called, and the second claim follows from the first one. This completes the proof of the two claims. Since the second claim, instantiated with $j = n$, implies that the variable assignment returned by the algorithm is a post solution of the lower monotonization of the system, it is sound. And by Lemma 1.3, it then is also a post solution of the original abstract system whenever all right-hand sides are monotonic. □

B Proof of Theorem 3

Proof. Assume that only finitely many variables are encountered during the run of the algorithm, i.e., from some point neither dom_0 nor dom_1 receive new elements. Since $\mathsf{solve}_0(y)$ is called before the variable y is added to dom_1, and $\mathsf{solve}_0(y)$ enforces that y is included in dom_0, we have that $\mathsf{dom}_1 \subseteq \mathsf{dom}_0$ throughout the algorithm. Due to the initial call $\mathsf{solve}_1(y_0, 0)$, y_0 is contained in Y_1 implying the first item in the list.

Variables y are added into sets $\mathsf{infl}[z]$ only during the evaluation of a call to eval_i and after an appropriate call to solve_i — implying that y is contained in dom_i whenever eval_i was called inside a call $\mathsf{do\_var}_i(y)$. Accordingly, all variables added to the priority queue necessarily are contained in dom_0. Thus, all variables for which $\mathsf{do\_var}_0$ is called at a call of $\mathsf{iterate}_0(n)$ are all contained in dom_0, while all variables for which $\mathsf{do\_var}_1$ is called at a call of $\mathsf{iterate}_1(n)$ are already contained in dom_1. Therefore, we define $Y_i = \mathsf{dom}_i$ when the iteration has terminated for $i = 0, 1$. We claim that for every priority n, the following holds:

1. Every call $\mathsf{iterate}_0(n)$ during the evaluation of $\mathsf{solve}_0(0, y_0)$ terminates.
2. Every call $\mathsf{iterate}_1(n)$ during the evaluation of $\mathsf{solve}_1(0, y_0)$ terminates as well.

In order to prove the first claim, assume for a contradiction that there is some n such that the call $\mathsf{iterate}_0(n)$ does not terminate. Since Y_0 is finite, there must by a variable y of maximal priority $\mathsf{prio}(y) \leq n$ so that $\mathsf{do\_var}_0(y)$ is evaluated

infinitely often. This means that from some point on, y is the variable of maximal priority for which $\mathsf{do\_var}_0$ is called. Let $d_i, i \geq 0$ denote the sequence of the new values for y. We claim that for every $i \geq 0$, $d_{i+1} = d_i \nabla a_i$ holds for some suitable value a_i. This holds if $y \in \mathsf{point}$ from the first evaluation onward. Clearly, if this were the case, we arrive at a contradiction, as any such widening sequence is ultimately stable. Accordingly, it remains to prove that from the first evaluation onward, y is contained in point — whenever $\mathsf{do\_var}_0(y)$ is called. Assume for a contradiction that there is a first such call where y is not contained in point. Assume that this call provided the ith value d_i for y. This means that, since the last evaluation of $f_y^\sharp$, no query to the value of y during the evaluation of lower priority variables has occurred. Accordingly, the set $\mathsf{infl}[y]$ does not contain any lower priority variables, which means that no further variable is evaluated before the next call $\mathsf{do\_var}_0(y)$. But then this next evaluation of $f_y^\sharp$ will return the value a. Subsequently, the queue Q does no longer contain variables of priority less then or equal to n, and therefore the iteration would terminate — in contradiction to our assumption.

Now consider the second claim. For a contradiction now assume that there is some n so that the call $\mathsf{iterate}_1(n)$ does not terminate. Since every call $\mathsf{iterate}_0(m)$ encountered during its evaluation is already known to terminate, we conclude that there must be a variable y of priority less then or equal to n so that $\mathsf{do\_var}_1(y)$ is evaluated infinitely often. As before this means that from some point on, y is the variable of maximal priority for which $\mathsf{do\_var}_1(y)$ is called. Let $d_i, i \geq 0$ denote the sequence of the new values for y. We claim that for every $i \geq 0$, $d_{i+1} = d_i \Delta a_i$ holds for some suitable value a_i. This holds if $d_i \sqsubseteq d_{i+1}$ and $y \in \mathsf{point}$ from $i = 1$ onward. Again, if this were the case, we arrive at a contradiction, as any such widening sequence is ultimately stable. Accordingly, it remains to prove that from the first evaluation onward, y is contained in point — whenever $\mathsf{do\_var}_1(y)$ is called. This, however, follows by the same argument as for $\mathsf{iterate}_0(y)$. This completes the proof of the claim.

By the claim which we have just proven, each occurring call $\mathsf{iterate}_i(n)$ will terminate. From that, the termination of the call $\mathsf{solve}_1(y_0, 0)$ follows as stated by the theorem.

It remains to prove the remaining two assertions of the enumeration. Again, we assume that only finitely many variables are encountered in a run of the local two-phase solver when started for a variable y_0, and assume that after some call to $\mathsf{do\_var}_i$, no further variable is added to dom_0, and likewise no further variable is added to dom_1. In order to prove the second assertion, we prove that the following invariants hold before every call $\mathsf{do\_var}_i(y_1)$:

1. For every variable y in the current domain dom_0, $\mathsf{infl}[y]$ contains (at least) all variables $z \notin Q \cup \{y_1\}$ whose last evaluation of $f_z^\sharp$ has called $\mathsf{eval}_i(y)$;
2. If $y \in \mathsf{dom}_0 \backslash (Q \cup \{y_1\})$, then $\sigma_0[y] \sqsupseteq f_y^\sharp(\underline{\top} \oplus \sigma_0)$;
3. If $y \in \mathsf{dom}_1 \backslash (Q \cup \{y_1\})$, then $\sigma_1[y] \sqsupseteq \underline{f}_y^\sharp(\underline{\top} \oplus \sigma_1)$;

Here, $\underline{\top}$ is the variable assignment which maps each variable in Y_0 to $\top$. Here, we only prove the second invariant. For that, consider a call to $\mathsf{do\_var}_1(y_1)$. If this

is the very first call of $\mathsf{do\_var}_1$ for y_1, then this occurs inside a call $\mathsf{solve}_1(y_1)$. Accordingly, the value $\sigma_1[y]$ has been initialized to $\sigma_0[y]$. Then we have, by the first invariant:

$$\sigma_1[y_1] = \sigma_0[y_1] \sqsupseteq f^\sharp_{y_1}\,(\top\!\!\top \oplus \sigma_0) \sqsupseteq \underline{f}^\sharp_{y_1}\,(\top\!\!\top \oplus \sigma_0)$$

At that moment, the priority queue does not contain any variable y with priority less or equal the priority of y_1, implying that for all these y, $\sigma_1[y] \sqsupseteq \underline{f}^\sharp_y(\top\!\!\top \oplus \sigma_1)$ holds. This property is preserved by updating $\sigma_1[y_1]$ with a value exceeding $\underline{f}^\sharp_{y_1}\,(\top\!\!\top \oplus \sigma_1)$. Accordingly, all variables z from $\mathsf{infl}[y_1]$ with priority less or equal to y_1 will subsequently be iterated upon with $\mathsf{iterate}_1$. But since these variables z satisfy $\sigma_1[z] \sqsupseteq \underline{f}^\sharp_z(\bot\!\!\bot \oplus \sigma_1)$, the invariant holds for the calls of $\mathsf{do\_var}_1$ therein.

By construction, 0 is the maximal priority of any variable. y_0 receives the least priority. Therefore, $\mathsf{solve}_1(0, y_0)$ returns with an empty queue Q. By the second invariant the second assertion of the theorem follows. □

C Proof of Theorem 4

By Theorem 4 the only condition for TSMP to terminate is that only finitely many variables are encountered. No further assumptions, e.g., w.r.t. monotonicity of right-hand sides must be made as in [4,2]. Upon termination, the algorithm is guaranteed to return sound results. The returned variable assignment is a (partial) post solution of the lower monotonization of the system, which means it may not necessarily be a post solution of the original system — given that some right-hand sides are not monotonic.

Assume that only variables from the finite set Y_0 are encountered during the run of the algorithm. We claim that for every priority i, the following holds:

1. Every call $\mathsf{iterate}(\mathbf{true}, i)$ during the evaluation of $\mathsf{solve}\, y_0$ terminates.
2. Every call $\mathsf{iterate}(\mathbf{false}, i)$ during the evaluation of $\mathsf{solve}\, y_0$ terminates as well.

In order to prove the first claim, assume for a contradiction that there is some i such that the call $\mathsf{iterate}(\mathbf{true}, i)$ does not terminate. Note that then any subsequent call to $\mathsf{do\_var}$ as well as $\mathsf{iterate}$ will always be evaluated for the Boolean value **true**. Since Y_0 is finite, there is a variable y of maximal priority $\mathsf{prio}(y) \leq i$ so that $\mathsf{do\_var}(\mathbf{true}, y)$ is evaluated infinitely often. This means that from some point on, y is the variable of maximal priority for which $\mathsf{do\_var}$ is called. Let $d_i, i \geq 0$ denote the sequence of the new values for y. We claim that for every $i \geq 0$, $d_{i+1} = d_i \,\Delta\, a_i$ holds for some suitable value a_i. This holds if $y \in \mathsf{point}$ from the first evaluation onward. Clearly, if this were the case, we arrive at a contradiction, as any such narrowing sequence is ultimately stable. Accordingly, it remains to prove that from the first evaluation onward, y is contained in point — whenever $\mathsf{do\_var}(\mathbf{true}, y)$ is called. Assume for a contradiction that there is a first such call where y is not contained in point. Assume that this call provided the ith value d_i for y. This means in particular that, since the last evaluation of $f^\sharp_y$, no query to the value of y during the evaluation of lower priority variables

has occurred. Accordingly, the set $\mathsf{infl}[y]$ does not contain any lower priority variables, which means that no further variable is evaluated before the next call $\mathsf{do\_var}(\mathbf{true}, y)$. But then this next evaluation of $f_y^\sharp$ will return the value a. Subsequently, the queue Q does no longer contain variables of priority $\leq i$, and therefore the iteration would terminate — in contradiction to our assumption.

Let us therefore now consider the second claim. For a contradiction now assume that there is some i so that the call $\mathsf{iterate}(\mathbf{false}, i)$ does not terminate. Since every call $\mathsf{iterate}(\mathbf{true}, j)$ encountered during its evaluation is already known to terminate, we conclude that there must be a variable y of maximal priority $\leq i$ so that $\mathsf{do\_var}(\mathbf{false}, y)$ is evaluated infinitely often. As before this means that from some point on, y is the variable of maximal priority for which $\mathsf{do\_var}(\mathbf{false}, y)$ is called. Let $d_i, i \geq 0$ denote the sequence of the new values for y. We claim that for every $i \geq 0$, $d_{i+1} = d_i \nabla a_i$ holds for some suitable value a_i where $y \in \mathsf{point}$ from $i = 1$ onward. Again, if this were the case, we arrive at a contradiction, as any such widening sequence is ultimately stable. Accordingly, it remains to prove that from the first evaluation onward, y is contained in point — whenever $\mathsf{do\_var}(\mathbf{false}, y)$ is called. This, however, follows by the same argument as for $\mathsf{iterate}(\mathbf{true}, y)$. This completes the proof of the claim.

Now assume that only finitely many variables are encountered in a run of TSMP when started for a variable x, and assume that after some call to $\mathsf{do\_var}$, no further variable is encountered. Let Y_0 denote this set of variables. By the claim which we have just proven, each subsequent call to the function $\mathsf{iterate}$ will terminate. From that, the termination of the call $\mathsf{solve}\, y_0$ follows as stated by the theorem.

It remains to prove the second assertion. We remark that, whenever a new variable is encountered, it is added into the set dom and never removed. Let Y_0 again denote the finite set of variables encountered during $\mathsf{solve}\, y_0$, i.e., the final value of dom. In particular, y_0 is contained in Y_0. In order to prove the second assertion, we note that the following invariants hold before every call $\mathsf{do\_var}(b, y_1)$:

1. For every variable y in the current domain dom, $\mathsf{infl}[y]$ contains (at least) all variables $z \notin Q \cup \{y_1\}$ whose last evaluation of $f_z^\sharp$ has called $\mathsf{eval}\, y$;
2. If $y \in \mathsf{dom} \backslash (Q \cup \{y_1\})$, then $\sigma[y] \sqsupseteq \underline{f}_y^\sharp(\underline{\top} \oplus \sigma)$;
3. If $b = \mathbf{true}$, then $\sigma[y_1] \sqsupseteq \underline{f}_{y_1}^\sharp(\underline{\top} \oplus \sigma)$.

Here, $\underline{\top}$ is the variable assignment which maps each variable in Y_0 to $\top$. In order to see the second statement, we observe that iteration on a variable always starts with the flag $\mathbf{false}$. Now consider a call to $\mathsf{do\_var}(\mathbf{false}, y_1)$ where b' is set to $\mathbf{true}$. This is the case when $\sigma[y_1] \sqsupseteq tmp$ where tmp is the value of the last evaluation of the right-hand side of y_1. Accordingly,

$$\sigma[y_1] \sqsupseteq f_{y_1}^\sharp\,(\underline{\top} \oplus \sigma) \sqsupseteq \underline{f}_{y_1}^\sharp\,(\underline{\top} \oplus \sigma)$$

At that moment, the priority queue does not contain any variable y with priority less or equal the priority of y_1, implying that for y, $\sigma[y] \sqsupseteq \underline{f}_y^\sharp(\underline{\top} \oplus \sigma)$ holds.

This property is preserved by updating $\sigma[y_1]$ with a value exceeding $\underline{f}_y^\sharp(\bot \oplus \sigma)$. Accordingly, all variables z from $\mathsf{infl}[y_1]$ with priority less or equal to y_1 will subsequently be iterated upon with $b = \mathbf{true}$. But since these satisfy $\sigma[z] \sqsupseteq \underline{f}_z^\sharp(\bot \oplus \sigma)$, the invariant holds for the calls of do_var therein.

By construction, y_0 receives the least priority. Therefore, solve y_0 returns with an empty queue Q. By the second invariant the second assertion of the theorem follows. □

D Experimental Evaluation

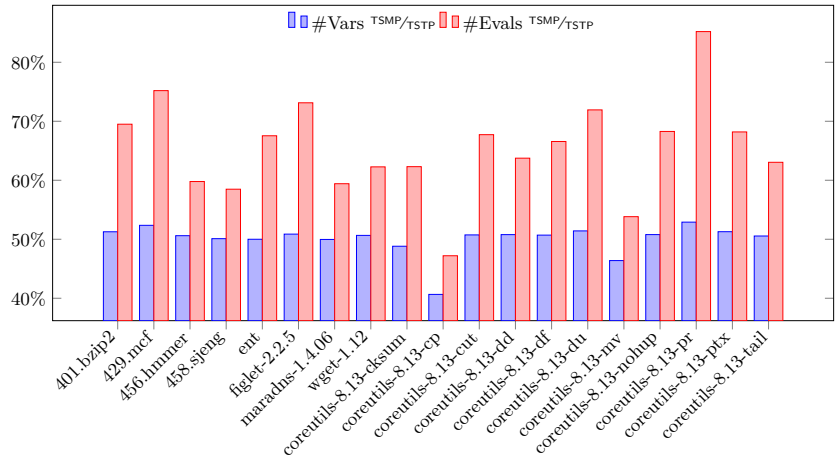
We implemented the solvers TSTP and TSMP presented in Sections 6 and 7 within the analysis framework GOBLINT[1]. For that, these solvers have been extended to deal with *side-effects* (see [3] for a detailed discussion of this mechanism) to jointly deal with flow- and context-sensitive and flow-insensitive analyses. In order to perform a fair comparison of the new solvers with *warrowing*-based local solving as proposed in [4,2], we provided a simplified version of TSMP. This simplified solver performs priority based iteration in the same way as TSMP but uses the warrowing operator instead of selecting operators according to extra flags. These three solvers were evaluated on the SPECint benchmark suite[2] consisting of not too small real-world C programs (1,600 to 34,000 LOC). Furthermore, the following C programs where analyzed: ent[3], figlet[4], maradns[5], wget[6], and some programs from the coreutils[7] package. The analyzed program wget is the largest one with around 77,000 LOC.

The analyses which we performed are put on top of a basic analysis of pointers, strings and enums. For enum variables, *sets* of possibles values are maintained. The benchmark programs were analyzed with full context-sensitivity of local data while globals were treated flow-insensitively.

The experimental setting is a fully context-sensitive interval analysis of int variables. Therefore, program 482.sphinx had to be excluded from the benchmark suite since it uses procedures which recurse on int arguments. Interestingly, the *warrowing* solver behaves exactly the same as TSMP on all of our benchmark programs. We interpret this that for the given analysis the right-hand sides are effectively monotonic. Accordingly, table 7 only reports the relative precision of TSTP compared to TSMP.

The table in Figure 6 compares the solvers TSMP and TSTP in terms of space and time. For a reasonable metric for space we choose the total number of variables (i.e., occurring pairs of program points and contexts) and for time the total number of evaluations of right-hand sides of a corresponding variable. The

[1] http://goblint.in.tum.de/
[2] https://www.spec.org/cpu2006/CINT2006/
[3] http://www.fourmilab.ch/random/ (version 28.01.2008)
[4] http://www.figlet.org/
[5] http://www.maradns.org/
[6] https://www.gnu.org/s/wget/
[7] https://www.gnu.org/s/coreutils/

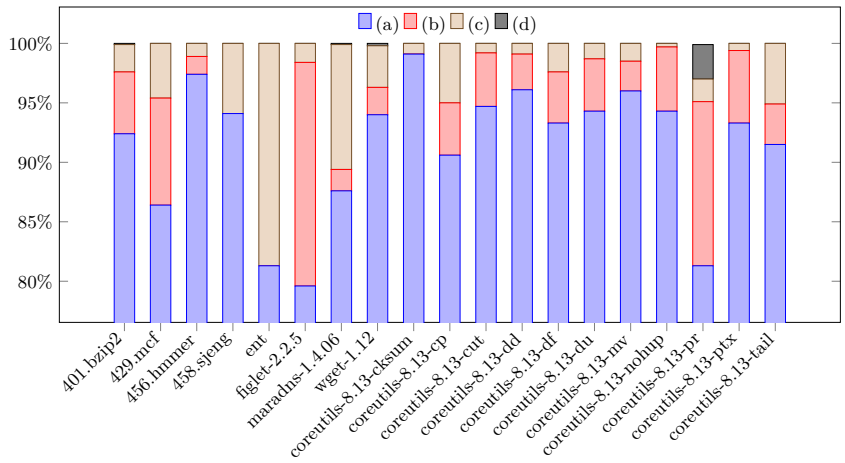


The blue bars (resp. red bars) depict the percentage of required variables (resp. evaluations of right-hand sides) of the solver TSMP compared to the solver TSTP.

Fig. 6. Efficiency of TSMP vs. TSTP.

table indicates that the solver TSMP requires only around half of the variables of the solver TSTP. Interestingly, the percentage of evaluations of right-hand sides in a run of TSMP is still around 60 to 70 percent of the solver TSTP.

In the second experiment, as depicted by Figure 7, we compare the precision of the two solvers. For a reasonable metric we only compare the values of variables which occur in both solver runs. As a result, between 80 and 95 percent of the variables receive the same value. It is remarkable that the mixed phase solver is not necessarily more precise. In many cases, an increase in precision is observed, yes — there are, however, also input programs where the two-phase solver excels.



Percentage of variables occurring during a a run of both solvers

(a) for which TSMP and TSTP compute the same value;
(b) for which TSMP computes more precise results then TSTP;
(c) for which TSTP computes more precise results then TSMP;
(d) for which the results computed by TSMP and TSTP are incomparable.

Fig. 7. Precision of TSMP vs. TSTP.