Efficiently intertwining widening and narrowing[☆,☆☆]

Gianluca Amato[a], Francesca Scozzari[a], Helmut Seidl[b], Kalmer Apinis[c], Vesal Vojdani[c]

[a] *Università di Chieti-Pescara*
[b] *Technische Universität München*
[c] *University of Tartu*

Abstract

Accelerated fixpoint iteration by means of widening and narrowing is the method of choice for solving systems of equations over domains with infinite ascending chains. The strict separation into an ascending widening and a descending narrowing phase, however, may unnecessarily give up precision that cannot be recovered later. It is also unsuitable for equation systems with infinitely many unknowns—where local solving must be used.

As a remedy, we present a novel operator $\boxslash$ that combines a given widening operator ∇ with a given narrowing operator Δ. We present adapted versions of round-robin and worklist iteration as well as local and side-effecting solving algorithms for the combined operator $\boxslash$. We prove that the resulting solvers always return sound results and are guaranteed to terminate for monotonic systems whenever only finitely many unknowns are encountered.

Keywords: Static Program Analysis, Fixpoint Iteration, Constraint Solving, Widening and Narrowing, Termination

1. Introduction

From an algorithmic point of view, static analysis typically boils down to solving systems of equations over a suitable domain of values. The unknowns of the system correspond to the invariants to be computed, e.g., for each program point or for each program point in a given calling context or instance of a class. For abstract interpretation, complete lattices were proposed as domains of abstract values (Cousot and Cousot, 1977a). In practice, partial orders can

☆This article extends and generalizes results presented by Apinis et al. (2013) by integrating key ideas from Amato and Scozzari (2013).

☆☆This work was partially supported by the ARTEMIS Joint Undertaking under grant agreement n° 269335 and from the German Science Foundation (DFG). This work was funded by institutional research grant IUT2-1 from the Estonian Research Council

Email addresses: `gamato@unich.it` (Gianluca Amato), `fscozzari@unich.it` (Francesca Scozzari), `seidl@in.tum.de` (Helmut Seidl), `kalmer.apinis@ut.ee` (Kalmer Apinis), `vesal.vojdani@ut.ee` (Vesal Vojdani)

be applied which are not necessarily complete lattices as long as they support an effective binary upper bound operation. This is the case, e.g., for polyhedra (Cousot and Halbwachs, 1978), zonotopes (Ghorbal et al., 2009) or parallelotopes (Amato and Scozzari, 2012). Still, variants of Kleene iteration can be used to compute solutions. Right from the beginnings of abstract interpretation, it became clear that many interesting invariants can only be expressed by domains with *infinite* strictly ascending chains. In the presence of possibly infinite strictly ascending chains, naive Kleene iteration is no longer guaranteed to terminate. For that reason, Cousot and Cousot proposed a *widening* iteration to obtain a valid invariant or, technically speaking, a *post* solution which subsequently may be improved by means of a *narrowing* iteration (Cousot and Cousot, 1976, 1992b). The widening phase can be considered as a Kleene iteration that is accelerated by means of a widening operator to enforce that only finitely many increases of values occur for every unknown. While enforcing termination, it may result in a crude over-approximation of the invariants of the program. In order to compensate for that, the subsequent narrowing iteration tries to improve a given post solution by means of a downward fixpoint iteration, which again may be accelerated, in this case by means of a narrowing operator.

Trying to recover precision once it has been thrown away, though, may not always possible (see, e.g., Halbwachs and Henry (2012) for a recent discussion). Some attempts try to improve precision by reducing the number of points where widening is applied (Cousot, 1981; Bourdoncle, 1993), while others rely on refined widening or narrowing operators (see, e.g., Simon and King (2006); Cortesi and Zanioli (2011)). Recently, several authors have focused on methods to guide or stratify the exploration of the state space (Gopan and Reps, 2007, 2006; Gulavani et al., 2008; Monniaux and Guen, 2012; Henry et al., 2012b), including techniques for automatic transformation of irregular loops (Gulwani et al., 2009; Sharma et al., 2011) or by repeating the widening/narrowing phases starting from a different initial state (Halbwachs and Henry, 2012). An interesting novel idea is to add a third phase where fixpoint iteration is started from scratch, but the best known previously computed upper bound for each unknown is exploited to improve intermediate values (Cousot, 2015). In that third phase, yet another operator, namely a *dual* narrowing is applied to enforce termination.

Our approach here at least partly encompasses those of Cousot (1981) and Bourdoncle (1993), while it is complementary to the other techniques and can, potentially, be combined with these. Our idea is to avoid postponing narrowing to a second phase after a post solution has been computed, in which all losses of information have already occurred and been propagated. Instead, we attempt to systematically improve the current information immediately by downward iterations. This means that increasing and decreasing iterations are applied in an *interleaved* manner. A similar idea is already used by syntax-directed fixpoint iteration engines as, e.g., in the static analyzers Astrée (Blanchet et al., 2003; Cousot et al., 2007) and Jandom (Amato and Scozzari, 2013). The Astrée analyzer follows the syntax of the program and performs a fixpoint iteration at every detected loop, consisting of a widening iteration followed by a narrowing iteration. Nesting of loops, therefore, also results in nested iterations. In order to enforce termination, *ad hoc* techniques such as restrictions to the number of updates are applied. Here, we explore iteration strategies in a generic setting, where no *a priori* knowledge of the application is available, and provide sufficient conditions for when particular fixpoint algorithms are guaranteed to terminate.

As we concentrate on the algorithmic side and application-independent generic solvers, we use the original notions of narrowing (Cousot and Cousot, 1976, 1992b), rather than more elaborate definitions (Cousot and Cousot, 1992a; Cousot, 2015) which refer to the *concrete* semantics of the system to be analyzed. The classic formulation of narrowing requires right-hand sides of equations to be *monotonic* so that the second iteration phase is guaranteed to be *descending* and thus improving. Accordingly, the narrowing operator is guaranteed to return meaningful results only when applied in *decreasing* sequences of values. The assumption of monotonicity of right-hand sides, even disregarding the occurrences of widening and narrowing operators, may not always be met. Monotonicity can no longer be guaranteed, e.g., when compiling context-sensitive inter-procedural analyses into systems of equations (Fecht and Seidl, 1999; Apinis et al., 2012).

Example 1. *For some domain $\mathbb{D}$, consider the system of equations*

$$h(x) = g(f(x)) \quad (x \in \mathbb{D})$$

over the unknowns $f(d), g(d), h(d), d \in \mathbb{D}$. Similar systems of equations are used by a context-sensitive inter-procedural analysis when separate unknowns are introduced for every possible (abstract) calling context of a procedure. In the given example system, the value of $f(x)$ determines the unknown $g(x')$ whose value contributes to the value of $h(x)$. Now consider a domain $\mathbb{D}$ with two distinct elements $a \sqsubset b$ and consider the assignments ρ_1, ρ_2 with

$$\begin{array}{lll} \rho_1[f(a)] = a & \rho_1[g(a)] = b & \rho_1[g(b)] = a \\ \rho_2[f(a)] = b & \rho_2[g(a)] = b & \rho_2[g(b)] = a \end{array}$$

which otherwise agree for every unknown. Then $\rho_1 \sqsubseteq \rho_2$, but the right-hand side of $h(a)$, when evaluated over ρ_1 results in b, while an evaluation over ρ_2 results in a. Accordingly, the given right-hand side cannot be monotonic. ■

For inter-procedural analysis with infinite domains, the resulting equation systems may also be *infinite*. These can be handled by *local* solvers. Local solvers query the value of an *interesting* unknown and explore the space of unknowns only as much as required for answering the query. For this type of algorithm, neither the set of unknowns to be evaluated nor their respective dependences are known beforehand. Accordingly, the values of fresh unknowns that have not yet been encountered may be queried in the narrowing phase. As a consequence, the rigid two-phase iteration strategy of one widening phase followed by one narrowing phase can no longer be maintained.

In order to cope with these obstacles, we introduce an operator $\boxslash$ which is a combination of a given widening ∇ with a given narrowing operator Δ and show that this new operator can be plugged into any solver of equation systems, be they monotonic or non-monotonic. The $\boxslash$ operator behaves like narrowing as long as the iteration is descending and like widening otherwise. As a result, solvers are obtained that return reasonably precise post solutions in one go, given that they terminate.

Termination, then, is indeed an issue. We present two example systems of monotonic equations where standard fixpoint algorithms, such as round robin or worklist iteration, fail to terminate when enhanced with the new operator. As a remedy, we develop a variant of round robin as well as a variant of worklist iteration which in absence of widening and narrowing are not much worse than their standard counterparts—but which additionally are guaranteed to terminate when the $\boxslash$ operator is applied to monotonic systems.

The idea of plugging the new operator $\boxslash$ into a *local* solver works as well. A local solver, such as that of Hofmann et al. (2010a), however, is not necessarily a solver in the sense of the present paper—meaning that a naive enhancement with the operator $\boxslash$ is no longer guaranteed to return sound results. As our main contribution, we therefore present a variation of this algorithm which always returns a (partial) post solution and, moreover, is guaranteed to terminate—at least for monotonic equation systems and if only finitely many unknowns are encountered. This algorithm is *generic* in that it considers right-hand sides as black-box functions, implemented perhaps in some programming language. It relies on *self-observation* not only for identifying dependencies between unknowns on the fly, but also to determine a suitable prioritization of the unknowns. This vanilla version of a local iterator then is extended to cope with the losses in precision detected by Amato and Scozzari (2013). We present novel techniques for *localizing* the use of the operator $\boxslash$ to loop heads only. These loop heads are dynamically detected and recomputed depending on the status of the fixpoint computation. Interestingly, dynamically recomputing loop heads during fixpoint computation increases precision significantly. As another improvement, we also considered dynamically restarting the iteration for subsets of unknowns during a narrowing sub-iteration. These algorithms are further extended to solvers for *side-effecting* constraint systems. Side-effecting systems can conveniently specify analyses that combine context-sensitive analysis of local information with flow-insensitive analysis of globals (Apinis et al., 2012) as provided, e.g., by the program analyzer Goblint (Vojdani and Vene, 2009). Since the different contributions to a global unknown are generated during the evaluation of a subset of right-hand sides, which is not known beforehand and may vary during fixpoint iteration, further non-trivial changes are required to handle this situation.

The obstacle remains that termination guarantees in presence of unrestricted non-monotonicity cannot be given. By practical experiments, we nevertheless provide evidence that our iterator as is, not only terminates but is reasonably efficient — at least for the equation systems of an inter-procedural interval analysis of several non-trivial real-world programs. Secondly, termination can be effectively guaranteed by *bounding* for each unknown the number of switches from narrowing back to widening, or, more smoothly, to apply more and more aggressive narrowing operators. Note that this family of restrictions is more liberal than restricting the number of updates of each unknown directly.

The rest of the paper is organized as follows. In Section 2, we present the concept of $\Box$-solvers for any binary operator $\Box$. In Section 3, we show that any such solver, when instantiated with $\boxslash$, returns a post solution of an arbitrary equation system (be it monotonic or not) whenever the solver terminates. In order to enforce termination at least for finite systems of monotonic equations, we provide in Section 4 new variants of round-robin iteration as well as of worklist-based fixpoint computation. Section 5 introduces the new generic local $\boxslash$-solver **SLR**, which is subsequently enhanced with localization of $\boxslash$ (Section 6) and restarting (Section 7). All three local solvers then are generalized to equation systems with side effects in Section 8. In Section 9, we compare the local solvers w.r.t. precision and efficiency within the analyzer framework Goblint. Finally, we discuss related work in Section 10 and conclude in Section 11.

Sections 2 to 5 and the first part of Section 8 are based on prior work (Apinis et al., 2013). The extension of ordinary and generic local solving provided in Sections 6 and 7, as well as the second half of Section 8 are new. Also the experimental evaluation in Section 9 has been redone completely.

2. Chaotic fixpoint iteration

Consider a system S of equations $x = f_x$, for a set of unknowns $x \in X$, and over a set $\mathbb{D}$ of values where the right-hand sides f_x are mappings $(X \to \mathbb{D}) \to \mathbb{D}$. Furthermore, let $\Box : \mathbb{D} \to \mathbb{D} \to \mathbb{D}$ be a binary operator to combine old values with the new contributions of the right-hand sides.

A $\Box$-*solution* of S is an assignment $\rho : X \to \mathbb{D}$ such that for all $x \in X$,

$$\rho[x] = \rho[x] \Box f_x\, \rho.$$

In the case that $\Box$ is defined as $a \Box b = b$, a $\Box$-solution is an ordinary solution of the system, i.e., a mapping ρ with $\rho[x] = f_x\, \rho$ for all unknowns x.

Example 2. *Consider the system:*

$$x_1 = x_2 - 1 \qquad x_2 = x_1 + 1$$

with $\mathbb{D} = \mathbb{R}$. Then a solution is given by, e.g., $\rho = \{x_1 \mapsto 4, x_2 \mapsto 5\}$. Now consider an operator $\Box$, defined by $a \Box b = \frac{a+b}{2}$. Then any solution is also a $\Box$-solution and vice versa. For arbitrary operators $\Box$, though, this need not necessarily be the case. ■

Most of the time $\mathbb{D}$ is an *upward-directed set*, i.e., a poset such that for each pair of elements $a, b \in \mathbb{D}$, there exists an upper bound z such that $z \sqsupseteq a$ and $z \sqsupseteq b$ (see, e.g., Carl and Heikkilä (2010) for general background). We denote by $a \sqcup b$ a generic upper bound of a and b. In case $\mathbb{D}$ is an upward-directed set, and the $\Box$-operator is an upper bound, a $\Box$-solution is a *post* solution of the system, i.e., a mapping ρ with $\rho[x] \sqsupseteq f_x\, \rho$ for all unknowns x. Likewise in case $\mathbb{D}$ is a *downward*-directed set and $\Box$ is a lower bound, a $\Box$-solution is a *pre* solution of the system, i.e., a mapping ρ with $\rho[x] \sqsubseteq f_x\, \rho$ for all unknowns x.

```
do
  dirty ← false;
  foreach $x \in X$ do
    new ← $\rho[x] \,\square\, f_x\ \rho$;
    if $\rho[x] \neq$ new then
      $\rho[x]$ ← new;
      dirty ← true;
while (dirty)
```

Figure 1: The solver **RR**.

```
W ← $X$;
while W $\neq \emptyset$ do
  $x$ ← extract(W);
  new ← $\rho[x] \,\square\, f_x\ \rho$;
  if $\rho[x] \neq$ new then
    $\rho[x]$ ← new;
    W ← W $\cup$ infl$_x$
done
```

Figure 2: The Solver **W**.

Example 3. *The set $\mathbb{N}$ of all non-negative numbers, equipped with the natural ordering is upward-directed. This set has a* least *element, namely 0. Now consider the system:*

$$x_1 = x_2 + 1 \qquad x_2 = x_1$$

Then $\rho_0 = \{x_1 \mapsto 2, x_2 \mapsto 1\}$ is a pre solution, while there is no post solution. If, on the other hand, we add ∞ as a greatest element to the domain of values, then there is one (post) solution, namely $\rho = \{x_1 \mapsto \infty, x_2 \mapsto \infty\}$. ■

The operator $\square$ can also be instantiated with widening and narrowing operators. According to Cousot and Cousot (1976, 1977a, 1992b), a *widening* operator ∇ for a poset $\mathbb{D}$ must satisfy $a \sqsubseteq a \nabla b$, $b \sqsubseteq a \nabla b$ for all $a, b \in \mathbb{D}$, and any sequence $a_i, i \geq 0$, with $a_{i+1} = a_i \nabla b_i$ for $i \geq 0$, cannot be strictly ascending. This implies that a ∇-solution then again provides a post solution of the original system S. A *narrowing* operator Δ, on the other hand, is an *interpolant* (Cousot, 2015). This means that $a \sqsupseteq b$ must imply $a \sqsupseteq (a \Delta b) \sqsupseteq b$. Furthermore, any sequence, $a_i, i \geq 0$, with $a_{i+1} = a_i \Delta b_i$ for $i \geq 0$ and $a_i \sqsupseteq b_i$ cannot be strictly descending. This means that narrowing can only be applied if the right-hand sides of equations are guaranteed to return values that are less than or equal to the values of the current left-hand sides. Thus a mapping ρ can only be a Δ-solution if it is a post solution of the system.

A (chaotic) $\square$-*solver* for systems of equations is an algorithm that maintains a mapping $\rho : X \to \mathbb{D}$ and performs a sequence of *update steps*, starting from an initial mapping ρ_0. In practice, ρ_0 is chosen to map each unknown to the *least element* of the upward-directed set—if available. Each update step selects an unknown x, evaluates the right-hand side f_x of x w.r.t. the current mapping ρ_i and updates the value for x, i.e.,

$$\rho_{i+1}[y] = \begin{cases} \rho_i[x] \,\square\, f_x\, \rho_i, & \text{if } x = y \\ \rho_i[y], & \text{otherwise.} \end{cases}$$

The algorithm is a $\square$-*solver* if upon termination the final mapping (after completing n steps) ρ_n is a $\square$-solution of S. In this sense, the *round-robin* iteration **RR** of Fig. 1 is a $\square$-solver for every binary operator $\square$. Note that, in most cases, we omit update step indices and, additionally, use imperative assignment syntax of the form $\rho[x] \leftarrow w$ to change the value of the unknown x to w in the mapping ρ.

In order to prove that a given algorithm is a $\Box$-solver, i.e., upon termination returns a $\Box$-solution, one typically verifies the invariant that for every terminating run of the algorithm producing the sequence $\rho_0, \rho_1, \ldots, \rho_n$ of mappings, and every unknown x, $\rho_i[x] \neq \rho_i[x] \,\Box\, f_x\, \rho_i$ implies that for some $j \geq i$, an update $\rho_{j+1}[x] = \rho_j[x] \,\Box\, f_x\, \rho_j$ occurs.

The *round-robin* algorithm considers right-hand sides of equations as black boxes. Such solvers are also called *generic*. Not every solver algorithm, though, is generic in this sense. The *worklist* algorithm **W** from Fig. 2 can only be used as $\Box$-solver, when the *dependences* between unknowns are provided beforehand. This means that for each right-hand side f_x, a (super-)set dep_x of unknowns must be given such that for all mappings ρ, ρ', $f_x\, \rho = f_x\, \rho'$ whenever ρ and ρ' agree on all unknowns in dep_x. From these sets, we define the sets infl_y of unknowns possibly *influenced* by (a change of the value of) the unknown y, i.e.,

$$\mathsf{infl}_y = \{x \in X \mid y \in \mathsf{dep}_x\} \cup \{y\}\,.$$

In the case that the value of some unknown y changes, all right-hand sides of unknowns in the set infl_y must be recomputed. Note that whenever an update to an unknown y provides a new value, we reschedule y for evaluation as well. This is a precaution for the case that the operator $\Box$ is *not* (right) idempotent. Here, an operator $\Box$ is called *idempotent* if the following equality:

$$(a \;\Box\; b) \;\Box\; b = a \;\Box\; b$$

holds for all a, b. In this sense, the operators $\sqcup$ and $\sqcap$ are idempotent and often also ∇ and Δ. An operator such as $\frac{a+b}{2}$, however, for $a, b \in \mathbb{R}$ is not idempotent.

3. Enhancing Narrowing

First, we observe:

Lemma 4. *Assume that all right-hand sides of the system S of equations over a poset $\mathbb{D}$ are monotonic and that ρ_0 is a post solution of S, and $\Box$ is a narrowing operator Δ. Then the sequence $\rho_0, \rho_1, \ldots$ of mappings produced by a generic $\Box$-solver, is defined and decreasing and consists of post solutions only.*

Proof. Assume that for $i = 1, 2, \ldots,$ x_i is the unknown corresponding to the ith evaluation of a right-hand side in the sequence of evaluation produced by the $\Box$-solver. The proof is by induction on i. The case $i = 0$ is trivial. Now consider $i > 0$, and assume that ρ_{i-1} is a post solution. This means that $f_{x_i}\, \rho_{i-1} \sqsubseteq \rho_{i-1}(x_i)$. Therefore, the value $\rho_i(x_i) = \rho_{i-1}(x_i) \,\Delta\, f_{x_i}\, \rho_{i-1}$ is defined and, due to the property of a narrowing, less than or equal to $\rho_{i-1}(x_i)$. Accordingly, we have $\rho_i \sqsubseteq \rho_{i-1}$. Due to the property of a narrowing and the monotonicity of f_{x_i}, we furthermore have that $\rho_i(x_i) \sqsupseteq f_{x_i}\, \rho_{i-1} \sqsupseteq f_{x_i}\, \rho_i$. Therefore, ρ_i is again a post solution. ■

Thus, any generic solver can be applied to improve a post solution by means of a narrowing iteration when all right-hand sides of equations are monotonic.

Equation systems for context-sensitive inter-procedural analysis, though, are not necessarily monotonic. In the following we show how to lift the technical restrictions to the applicability of narrowing. For a widening operator ∇ and a narrowing operator Δ, we define a new binary operator $\boxslash$ by:

$$a \boxslash b = \begin{cases} a \,\Delta\, b, & \text{if } b \sqsubseteq a \\ a \,\nabla\, b, & \text{otherwise.} \end{cases}$$

Let us call the resulting operator $\boxslash$ a *warrowing*. Note that the operator $\boxslash$ is not necessarily idempotent, but whenever narrowing is idempotent the following holds:

$$(a \;\boxslash\; b) \;\boxslash\; b = (a \;\boxslash\; b) \;\;\Delta\;\; b$$

and therefore also

$$((a \;\boxslash\; b) \;\boxslash\; b) \;\boxslash\; b = (a \;\boxslash\; b) \;\boxslash\; b\,.$$

A fixpoint algorithm equipped with warrowing applies widening as long as values grow or are incomparable. Only once the evaluation of the right-hand side of an unknown results in a smaller or equal value, narrowing is applied and values may shrink. For the warrowing operator $\boxslash$, we observe:

Lemma 5. *Consider a finite system S of equations over an upward-directed set $\mathbb{D}$. Then every $\boxslash$-solution ρ of S is a post solution.*

Proof. Consider a mapping ρ that is a $\boxslash$-solution of S and an arbitrary unknown x. For a contradiction assume that $\rho[x] \not\sqsupseteq f_x\,\rho$. But then we have

$$\rho[x] \;=\; \rho[x] \boxslash f_x\,\rho \;=\; \rho[x] \,\nabla\, f_x\,\rho \;\sqsupseteq\; f_x\,\rho$$

in contradiction to our assumption! Accordingly, ρ must be a post solution of the system of equations S. ■

Thus, every generic solver for upward-directed sets $\mathbb{D}$ can be turned into a solver computing post solutions by using the combined widening and narrowing operator. The intertwined application of widening and narrowing, which naturally occurs when solving the system of equations by means of $\boxslash$, has the additional advantage that values may also *shrink* in-between. Improvement of too great (imprecise) values, thus, may take place immediately, resulting in overall lesser (more precise) post solutions. Moreover, no restriction is imposed any longer concerning monotonicity of right-hand sides.

4. Enforcing termination

For the new warrowing operator, termination cannot generally be guaranteed for all solvers. In this section, we therefore present a modifications of *round-robin* as well as of *worklist* iteration which are guaranteed to terminate when all right-hand sides of equations are monotonic.

Example 6. *Consider the system:*

$$x_1 = x_2 \qquad x_2 = x_3 + 1 \qquad x_3 = x_1$$

with $\mathbb{D} = \mathbb{N} \cup \{\infty\}$*, the lattice of non-negative integers, equipped with the natural ordering* $\sqsubseteq$ *given by* $\leq$ *and extended with* ∞*. Consider a widening* ∇ *where* $a \nabla b = a$ *if* $a = b$ *and* $a \nabla b = \infty$ *otherwise, together with a narrowing* Δ *where, for* $a \geq b$, $a \Delta b = b$ *if* $a = \infty$*, and* $a \Delta b = a$ *otherwise.* Round-robin *iteration with the warrowing operator for this system starting from the mapping* $\rho_0 = \{x_1 \mapsto 0, x_2 \mapsto 0, x_3 \mapsto 0\}$*, will produce the following sequence of mappings:*

	0	1	2	3	4	5	
x_1	0	0	∞	1	∞	2	...
x_2	0	∞	1	∞	2	∞	...
x_3	0	0	∞	1	∞	2	...

Iteration does not terminate—although right-hand sides are monotonic. ■

A similar example shows that ordinary *worklist* iteration, enhanced with $\boxslash$, also may not terminate, even if all equations are monotonic.

Example 7. *Consider the two equations:*

$$x_1 = (x_1 + 1) \sqcap (x_2 + 1) \qquad x_2 = (x_2 + 1) \sqcap (x_1 + 1)$$

using the same lattice as in Example 6 where $\sqcap$ *denotes minimum, i.e., the greatest lower bound. Assume that the work set is maintained with a lifo discipline. For* $W = [x_1, x_2]$*, worklist iteration, starting with the initial mapping* $\rho_0 = \{x_1 \mapsto 0, x_2 \mapsto 0\}$*, results in the following iteration sequence:*

W	$[x_1, x_2]$	$[x_1, x_2]$	$[x_1, x_2]$	$[x_2]$	$[x_2, x_1]$	$[x_2, x_1]$	$[x_1]$	$[x_1, x_2]$	
x_1	0	∞	1	1	1	1	1	∞	...
x_2	0	0	0	0	∞	2	2	2	...

which does not terminate. ■

We present modified versions of the *round-robin* solver **RR** as well as the *worklist* solver **W** for which termination can be guaranteed. For both algorithms, we assume that we are given a fixed *linear ordering* on the set of unknowns so that $X = \{x_1, \ldots, x_n\}$.

The ordering will affect the iteration strategy, and therefore, as shown by Bourdoncle (1990), may have a significant impact on performance and, in presence of widening and narrowing, also upon precision. Hence, the ordering should be chosen in a way that innermost loops would be evaluated before iteration on outer loops.

```
let rec solve $i$ =
  if $i = 0$ then return;
  do
    solve($i$−1);
    old ← $\rho[x_i]$;
    $\rho[x_i]$ ← $\rho[x_i] \,\square\, f_i\ \rho$;
  while old $\neq \rho[x_i]$;
}
```

Figure 3: The new solver **SRR**.

For the system of equations given by $x_i = f_i$, for $i = 1, \ldots, n$, the new algorithm **SRR** (*structured* round-robin) is shown in Fig. 3. For a given initial mapping ρ_0, *structured round-robin* is started by calling solve n. The idea of the algorithm is, when called for a number i, to iterate on the unknown x_i until stabilization. Before every update of the unknown x_i, however, all unknowns $x_j, j < i$ are recursively solved. The resulting algorithm is a $\square$-solver for every binary operator $\square$.

Recall that a poset $\mathbb{D}$ has *height* h if h is the maximal length of a strictly increasing chain $d_0 \sqsubset d_1 \sqsubset \ldots \sqsubset d_h$. We find:

Theorem 8. *Assume the algorithm **SRR** is applied to a system of n equations over an upward-directed set $\mathbb{D}$.*

1. *If $\mathbb{D}$ has bounded height h and $\square = \sqcup$, then **SRR** terminates with a post solution after at most $n + \frac{h}{2}n(n+1)$ evaluations of right-hand sides.*
2. *In presence of possibly unbounded ascending chains, when instantiated with $\square = \boxslash$, **SRR** terminates with a post solution—whenever each right-hand side is monotonic.*

The first statement indicates that **SRR** may favorably compete with ordinary *round-robin* iteration in case that no widening and narrowing is required. The second statement, on the other hand, provides us with a termination guarantee — whenever all right-hand sides are monotonic.

Proof. Recall that ordinary *round-robin* iteration for upward-directed sets of bounded height performs at most $h \cdot n$ rounds due to increases of values of unknowns plus one extra round to detect termination, giving in total

$$n + h \cdot n^2$$

evaluations of right-hand sides. In contrast for *structured round-robin iteration*, termination for unknown x_i requires one evaluation when solve i is called for the first time and then one further evaluation for every update of one of the unknowns $x_n, \ldots, x_{i+1}$. This sums up to $h \cdot (n-i) + 1$ evaluations throughout the whole iteration. This gives a overhead of

$$n + h \cdot \sum_{i=1}^{n} (n-i) = n + \frac{h}{2} \cdot n \cdot (n-1)\ .$$

Additionally, there are $h \cdot n$ evaluations that increase values. In total, the number of evaluations, therefore, is

$$n + \frac{h}{2} \cdot n \cdot (n-1) + h \cdot n = n + \frac{h}{2} \cdot n \cdot (n+1)$$

giving us statement 1.

For the second statement, we proceed by induction on i. The case $i = 0$ is vacuously true. For the inductive step, assume that $i > 0$ and for all $j < i$, solve j terminates for any mapping. To arrive at a contradiction, assume that solve i for the current mapping ρ does not terminate. First, consider the case where $f_i\ \rho$ returns a value smaller than $\rho[x_i]$. Since **SRR** is a $\Box$-solver for every $\Box$, we have for all $j < i$, $\rho[x_j] = \rho[x_j] \boxslash f_j\ \rho$, implying that $\rho[x_j] \sqsupseteq f_j\ \rho$ for all $j < i$. After $\rho[x_i]$ is updated, by monotonicity, it still holds that $\rho[x_j] \sqsupseteq f_j\ \rho$ for all $j < i$. Solving for the unknown $i-1$ will only cause further descending steps, where $\boxslash$ behaves like Δ. The subsequent iteration of solve i will produce a decreasing sequence of mappings. Since all decreasing chains produced by narrowing are ultimately stable, the call solve i will terminate—in contradiction to our assumption.

Therefore, non-termination is only possible if during the whole run of solve i, evaluating $f_i\ \rho$ must always return a value that is not subsumed by $\rho[x_i]$. Since all calls solve $(i-1)$ in-between terminate by the induction hypothesis, a strictly increasing sequence of values for x_i is obtained that is produced by repeatedly applying the widening operator. Due to the properties of widening operators, any such sequence is eventually stable—again in contradiction to our assumption. We thus conclude that solve i is eventually terminating. ■

Example 9. *Recall the equation system, for which* round-robin *iteration did not terminate. With* structured round-robin *iteration, however, we obtain the following sequence of updates:*

i		2	1	2	1	3	2	1
x_1	0	0	∞	∞	1	1	1	∞
x_2	0	∞	∞	1	1	1	∞	∞
x_3	0	0	0	0	0	∞	∞	∞

where the evaluations of unknowns not resulting in an update have been omitted. Thus, structured round-robin *quickly stabilizes for this example.* ■

The idea of *structured* iteration can also be lifted to *worklist* iteration. Consider again a system $x_i = f_i$, for $i = 1, \ldots, n$, of equations. As for the ordinary *worklist* algorithm, we assume that for each right-hand side f_i a superset dep_i of unknowns is given, such that for all mappings ρ, ρ', $f_i\,\rho = f_i\,\rho'$ whenever ρ and ρ' agree on all unknowns in dep_i.

```
$Q$ ← {1, ..., n};
while $Q \neq \emptyset$ do
  $x_i$ ← extract_min($Q$);
  new ← $\rho[x_i] \Box f_i\ \rho$;
  if $\rho[x_i] \neq$ new then
    $\rho[x_i]$ ← new;
    add $Q$ infl$_i$;
done
```

Figure 4: The new solver **SW**.

As before for each unknown x_j, let infl_j denote the set consisting of the unknown x_j together with all unknowns *influenced* by x_j. Instead of a plain worklist, the modified algorithm maintains the set of unknowns to be reevaluated, within a *priority queue* Q. In every round, not an arbitrary element is extracted from Q — but the unknown with the least index. The resulting algorithm is presented in Fig. 4.

Here, the function add inserts an element into the priority queue or leaves the queue unchanged if the element is already present. Moreover, the function extract_min removes the unknown with the smallest index from the queue and returns it as result.

Let us call the resulting algorithm **SW** (*structured worklist* iteration). Clearly, the algorithm **SW** is a $\Box$-solver for systems of equations where the dependences between unknowns are explicitly given.

Example 10. *Consider again the system from Example 7. Structured worklist iteration using $\boxslash$ for this system results in the following iteration:*

Q	$[x_1,x_2]$	$[x_1,x_2]$	$[x_1,x_2]$	$[x_2]$	$[x_1,x_2]$	$[x_1,x_2]$	$[x_2]$	$[]$
x_1	0	∞	1	1	1	∞	∞	∞
x_2	0	0	0	0	∞	∞	∞	∞

and thus terminates. ■

In general, we have:

Theorem 11. *Assume the algorithm **SW** is applied to a system of n equations over an upward-directed set $\mathbb{D}$.*

1. *If $\mathbb{D}$ has bounded height h and $\Box = \sqcup$, then **SW** terminates with a post solution after at most $h \cdot N$ evaluations of right-hand sides, where $N = \sum_{i=1}^{n}(2 + |\mathsf{dep}_i|)$.*
2. *In presence of possibly unbounded ascending chains, when instantiated with $\Box = \boxslash$, **SW** terminates with a post solution—whenever each right-hand side is monotonic.*

The first statement of the theorem indicates that **SW** behaves complexity-wise like ordinary *worklist* iteration: in case that the upward-directed set $\mathbb{D}$ has finite height, the only overhead to be paid for is an extra logarithmic factor for maintaining the priority queue. The second statement, perhaps, is more surprising: it provides us with a termination guarantee for arbitrary lattices and the warrowing operator — whenever only all right-hand sides are monotonic.

Proof. We only consider the second statement. We proceed by induction on the number n of unknowns. The case $n = 1$ is true by definition of widening and narrowing. For the induction step assume that the assertion holds for systems of equations of $n-1$ unknowns. Now consider a system of equations for a set X of cardinality n, and assume that x_n is the unknown which is larger than all other unknowns in X.

For a contradiction assume that **SW** does not terminate for the system of equations for X. First assume that the unknown x_n is extracted from the queue Q only finitely many times, say k times where d is the last value computed for x_n. This means that after the last extraction, an infinite iteration occurs on the subsystem on the unknowns $X' = X \setminus \{n\}$ where for $x_r \in X'$, the right-hand side is given by $f'_r\,\rho = f_r\,(\rho \oplus \{x_n \mapsto d\})$. By inductive hypothesis, however, the algorithm **SW** for this system terminates — in contradiction to our assumption.

Therefore, we may assume that the unknown x_n is extracted infinitely often from Q. Let $\rho_i, i \in \mathbb{N}$, denote the sequence of mappings at these extractions. Since Q is maintained as a priority queue, we know that for all unknowns x_r with $r < n$, the inequalities $\rho_i[x_r] \sqsupseteq f_r\,\rho_i$ hold. Let $d_i = \rho_i[x_n]$. If for any i, $f_n\,\rho_i \sqsubseteq d_i$, the next value d_{i+1} for x_n then is obtained by $d_{i+1} = d_i \,\Delta\, f_n\,\rho_i$ which is less or equal to d_i. By monotonicity, this implies that in the subsequent iteration, the values for all unknowns $x_r, r \leq n$, may only decrease. The remaining iteration is a pure narrowing iteration and therefore terminates. In order to obtain an infinite sequence of updates for z, we conclude that for no i, $f_n\,\rho_i \sqsubseteq d_i$. Hence for every i, $d_{i+1} = d_i \,\nabla\, f_n\,\rho_i$ where $d_i \sqsubseteq d_{i+1}$. This, however, is impossible due to the properties of the widening operator. In summary, we conclude that x_n is extracted only finitely often from Q. Hence the fixpoint iteration terminates. ∎

The algorithm **SW** can also be applied to non-monotonic systems. There, however, termination can no longer be guaranteed. In fact, the assumption of monotonicity is not a defect of our solvers **SRR** or **SW**, but inherent to *any* terminating fixpoint iteration which intertwines widening and narrowing.

Example 12. *Consider the single equation:*

$$\mathsf{x} = \textbf{if } (\mathsf{x} = 0) \textbf{ then } 1 \textbf{ else } 0$$

over the lattice of naturals (with infinity) with $a \,\nabla\, b = \infty$ whenever $a < b$ and $a \,\Delta\, b = b$ whenever $a = \infty$. The right-hand side of this equation is not monotonic. An iteration, defined by $\mathsf{x}_0 = 0$ and $\mathsf{x}_{i+1} = \mathsf{x}_i \boxslash f \mathsf{x}_i$ for $i \geq 0$ (f the right-hand side function of the equation) will produce the sequence:

$$0 \to \infty \to 0 \to \infty \to 0 \to \infty \to \ldots$$

and thus will not terminate. We conclude that in absence of monotonicity, we cannot hope for termination—at least, without further assumptions on the right-hand sides of the equations. ∎

Clearly at the price of extra imprecision, termination for *all* $\boxslash$-solvers and all monotonic or non-monotonic systems of equations can always be enforced. One generic idea to achieve this, is to count for each unknown how often the solver has switched from narrowing back to widening. This number then may be taken into account by the $\boxslash$ operator, e.g., by choosing successively less aggressive narrowing operators $\Delta_0, \Delta_1, \ldots$, and, ultimately, to give up improving the obtained values. The latter is achieved by defining $a \, \Delta_k \, b = a$ for a certain threshold k.

5. Local generic solvers

Similar to $\Box$-solvers, we define *local* $\Box$-solvers. Local solvers should be considered if systems of equations are infeasibly large or even infinite. Such systems are, e.g., encountered for context-sensitive analysis of procedural languages (Cousot and Cousot, 1977b; Apinis et al., 2012). Local solvers query the system of equations for the value of a given unknown of interest and try to evaluate only the right-hand sides of those unknowns that are needed for answering the query (Le Charlier and Van Hentenryck, 1992; Vergauwen et al., 1994; Fecht and Seidl, 1999). As the dependences between unknowns may change during fixpoint iteration, such a solver should keep track of the *dynamic* dependences between unknowns. For a mapping ρ, a set $X' \subseteq X$ subsumes all dynamic dependences of a function $f : (X \to \mathbb{D}) \to \mathbb{D}$ (w.r.t. ρ) in the case that $f\,\rho = f\,\rho'$ whenever $\rho'|_{X'} = \rho|_{X'}$. Such sets can be constructed on the fly whenever the function f is *pure* in the sense of Hofmann et al. (2010b).

Essentially, purity for a right-hand side f means that evaluating $f\,\rho$ for a mapping ρ operationally consists of a finite sequence of value lookups in ρ where the next unknown whose value has to be looked up may only depend on the values that have already been queried. Once the sequence of lookups has been completed, the final value is determined depending on the sequence of values and finally returned. In this case, the set X' can be chosen as the set of all variables y for which the value $\rho\,y$ is queried when evaluating (an implementation of) the function f for the argument ρ. Let us denote this set by $\mathsf{dep}_x\,\rho$.

A *partial* $\Box$-solution of a (finite or infinite) system of pure equations S consists of a set $\mathsf{dom} \subseteq X$ together with a mapping $\rho : \mathsf{dom} \to \mathbb{D}$ with the following two properties:

1. $\rho[x] = \rho[x] \,\Box\, f_x\,\rho$ for all $x \in \mathsf{dom}$; and
2. $\mathsf{dep}_x\,\rho \subseteq \mathsf{dom}$ for all $x \in \mathsf{dom}$

In essence, this means that a partial $\Box$-solution is a $\Box$-solution of the subsystem of S restricted to unknowns in dom.

Example 13. *The following equation system (for $n \in \mathbb{N} = \mathbb{D}$)*

$$\begin{aligned} y_{2n} &= \max(y_{y_{2n}}, n) \\ y_{2n+1} &= y_{6n+4} \end{aligned}$$

is infinite as it uses infinitely many unknowns, but has at least one finite partial max*-solution—the set* $\mathsf{dom} = \{y_1, y_2, y_4\}$ *together with the mapping* $\rho = \{y_1 \mapsto 2, y_2 \mapsto 2, y_4 \mapsto 2\}$ *where* $\mathsf{dep}_{y_1}\,\rho = \{y_4\}$, $\mathsf{dep}_{y_2}\,\rho = \{y_2\}$ *and* $\mathsf{dep}_{y_4}\,\rho = \{y_4, y_2\}$. ■

```
let rec solve $x$ =
  if $x \notin$ stable then
    stable $\leftarrow$ stable $\cup \{x\}$;
    tmp $\leftarrow$ $\rho[x] \sqcup f_x$ (eval $x$);
    if tmp $\neq \rho[x]$ then
      W $\leftarrow$ infl$[x]$;
      $\rho[x]$ $\leftarrow$ tmp;
      infl$[x]$ $\leftarrow$ $\emptyset$;
      stable $\leftarrow$ stable $\setminus$ W;
      foreach $x \in$ W do solve $x$
    end
  end
  and eval $x$ $y$ =
    solve $y$;
    infl$[y]$ $\leftarrow$ infl$[y]$ $\cup \{x\}$;
    $\rho[y]$

  in
    stable $\leftarrow$ $\emptyset$;
    infl $\leftarrow$ $\emptyset$;
    $\rho \leftarrow$ $\rho_0$;
    solve $x_0$;
    $\rho$
```

Figure 5: The solver **RLD** (Hofmann et al., 2010a).

A *local* $\Box$-solver then, is an algorithm that, when given a system of pure equations S, an initial mapping ρ_0 for all unknowns, and an unknown x_0, performs a sequence of update operations that, upon termination, results in a partial $\Box$-solution (dom, ρ), such that $x_0 \in \mathsf{dom}$.

At first sight, it may seem surprising that *generic* local $\Box$-solvers may exist. In fact, one such instance can be derived from the *round-robin* algorithm. For that, the evaluation of right-hand sides is instrumented in such a way that it keeps track of the set of accessed unknowns. Each round then operates on a growing set of unknowns. In the first round, just x_0 alone is considered. In any subsequent round all unknowns are added whose values have been newly accessed during the last iteration.

A more elaborate algorithm for local solving is formalized by Hofmann et al. (2010a), namely the solver **RLD**, as shown in Figure 5. This algorithm has the benefit of visiting nodes in a more efficient order, first stabilizing innermost loops before iterating on outer loops. The global assignment $\mathsf{infl} : X \to 2^X$ records, for each encountered unknown y, the set of unknowns $x \in \mathsf{dom}$ with the following two properties:

- the last evaluation of f_x has accessed the unknown y;
- since then, the value of the unknown y has not changed.

The right-hand sides f_x are not directly applied to the current mapping ρ, but instead to a (partially applied) helper function eval which in the end, returns values for unknowns. Before that, however, the helper function eval provides extra book-keeping of the encountered dependence between unknowns. In order to be able to track dependences between unknowns, the helper function eval receives as a first argument the unknown x whose right-hand side is under evaluation. Given that this partial application is applied to another unknown y, first the best possible value for y is computed by calling the procedure solve for y. Then the fact that the right-hand side of x depends on y, is recorded by adding x to the set $\mathsf{infl}[y]$. Only then is the corresponding value $\rho[y]$ returned.

The main fixpoint iteration is implemented by the procedure solve. It requires a set stable of unknowns such that, if x is in stable, a call to the procedure solve x has been started and no unknowns influencing x have been updated.

This algorithm correctly determines a post solution of the set of equations upon termination. However, when enhanced with an operator $\Box$, it is *not* a generic $\Box$-solver in our sense, since it is not guaranteed to execute as a sequence of *atomic* updates. Due to the recursive call to procedure solve at the beginning of eval, one evaluation of a right-hand side may occur nested into the evaluation of another right-hand side. Therefore, conceptually, it may happen that an evaluation of a right-hand side uses the values of unknowns from several different mappings ρ_i from the sequence $\rho_0, \rho_1, \ldots, \rho_n$, instead of the latest mapping ρ_n. Accordingly, the solver **RLD** is not guaranteed to return a $\Box$-solution—even if it terminates. We therefore provide a variant of **RLD** where right-hand sides (conceptually) are executed atomically.

Clearly, a local $\Box$-solver does not terminate if infinitely many unknowns are encountered. Therefore, a reasonable local $\Box$-solver will try to consider as few unknowns as possible. Our solver, thus, explores the values of unknowns by recursively descending into solving unknowns *newly* detected while evaluating a right-hand side. Certain equation systems, though, introduce infinite chains of dependences for the unknowns of interest. Those systems then cannot be solved by any local solver. Here, we show that the new generic solver is guaranteed to terminate for the warrowing operator at least when applied to equation systems which are monotonic and either finite or infinite but where only finitely many unknowns are encountered.

Let us call the new solver, on Fig. 6, $\mathbf{SLR}_1$ (*structured local recursive* solver). The new algorithm maintains an explicit set $\mathsf{dom} \subseteq X$ of unknowns that have already been encountered. Beyond **RLD**, it additionally maintains a counter count which counts the number of elements in dom, and a mapping $\mathsf{key} : \mathsf{dom} \to \mathbb{Z}$ that equips each unknown with its priority. Unknowns whose equations may possibly be no longer valid will be scheduled for reevaluation. This means that they are inserted into a global priority queue Q.

As in the algorithm **RLD**, right-hand sides f_x are applied to the helper function eval partially applied to x. The call eval x y first checks whether the unknown y is already contained in the domain dom of ρ. If this is not the case, y is first initialized by calling the procedure init. Subsequently, the best possible value for y is computed by calling the procedure solve for y.

Initialization of a fresh unknown y means that y is inserted into dom where it receives a key less than the keys of all other unknowns in dom. For that, the variable count is used. Moreover, $\mathsf{infl}[y]$ and $\rho[y]$ are initialized with $\{y\}$ and $\rho_0[y]$, respectively. Thus, the given function eval differs from the corresponding function in **RLD** in that solve is recursively called only for *fresh* unknowns, and also in that every unknown y always depends on itself.

The main fixpoint iteration is implemented by the procedure solve. When solve is called for an unknown x, we assume that there is currently no unknown $x' \in \mathsf{dom}$ with $\mathsf{key}[x'] < \mathsf{key}[x]$ that violates its equation, i.e., for which $\rho[x'] \neq \rho[x'] \;\Box\; f_{x'}\,\rho$ holds. In the procedure solve for x, the call min_key Q

```
let rec solve $x$ =
  if $x \notin$ stable then
    stable $\leftarrow$ stable $\cup \{x\}$;
    tmp $\leftarrow$ $\rho[x] \,\square\, f_x$ (eval $x$);
    if tmp $\neq \rho[x]$ then
      W $\leftarrow$ infl$[x] \cup \{x\}$;
      add $Q$ W;
      $\rho[x]$ $\leftarrow$ tmp;
      infl$[x]$ $\leftarrow$ $\emptyset$;
      stable $\leftarrow$ stable $\setminus$ W;
      while $(Q \neq \emptyset) \wedge$
            (min_key $Q \leq$ key$[x]$) do
        solve (extract_min $Q$);
    end
  end

and init $y$ =
  dom $\leftarrow$ dom $\cup \{y\}$;
  key$[y]$ $\leftarrow$ $-$count;
  count++;
  infl$[y]$ $\leftarrow$ $\emptyset$;
  $\rho[y]$ $\leftarrow$ $\rho_0[y]$

and eval $x$ $y$ =
  if $y \notin$ dom then
    init $y$;
    solve $y$;
  end;
  infl$[y]$ $\leftarrow$ infl$[y] \cup \{x\}$;
  $\rho[y]$

in
  stable $\leftarrow$ $\emptyset$; infl $\leftarrow$ $\emptyset$;
  $\rho \leftarrow \emptyset$; dom $\leftarrow$ $\emptyset$;
  $Q$ $\leftarrow$ empty_queue();
  count $\leftarrow$ 0; init $x_0$;
  solve $x_0$;
  $\rho$
```

Figure 6: The new solver $\mathbf{SLR}_1$.

returns the minimal key of an element in Q, and extract_min Q returns the unknown in Q with minimal key and additionally removes it from Q. Besides the global priority queue Q, the procedure solve also requires the set stable as for **RLD**. Due to the changes in eval and the fact that x is always added to W during the execution of solve x, at each call of the procedure solve, if $x \in$ stable then either

- a call to the procedure solve x has been started and the update of $\rho[x']$ has not yet occurred; or
- the equality $\rho[x] = \rho[x] \,\square\, f_x\,\rho$ holds.

The new function solve essentially behaves like the corresponding function in **RLD** with the notable exception that not necessarily all unknowns that have been found unstable after the update of the value for x in ρ, are recursively solved right-away. Instead, all these unknowns are inserted into the global priority queue Q and then solve is only called for those unknowns x' in Q whose keys are less or equal than key$[x]$. Since x_0 has received the largest key, the initial call solve x_0 will result, upon termination, in an empty priority queue Q.

Example 14. *Consider again the infinite equation system from Example 13. The solver* $\mathbf{SLR}_1$*, when solving for* y_1*, will return the partial* max*-solution* $\{y_0 \mapsto 0, y_1 \mapsto 2, y_2 \mapsto 2, y_4 \mapsto 2\}$. ■

The modifications of the algorithm **RLD** to obtain algorithm $\mathbf{SLR}_1$ allow us not only to prove that it is a generic local solver, but also a strong result concerning termination. Our main theorem is:

Theorem 15. *Assume the algorithm* $\boldsymbol{SLR}_1$ *is applied to a system of pure equations over a set* $\mathbb{D}$*, with initially queried* interesting *unknown* x_0.

1. $\boldsymbol{SLR}_1$ *returns a partial* $\square$*-solution whose domain contains* x_0*—whenever it terminates.*
2. *If* $\mathbb{D}$ *is an upward-directed set, each right-hand side of the equation system is monotonic and the operator* $\square$ *is instantiated with* $\boxslash$*, then* $\boldsymbol{SLR}_1$ *is guaranteed to terminate and thus always to return a partial post solution—whenever only finitely many unknowns are encountered.*

PROOF. We first convince ourselves that, upon termination, each right-hand side can be considered as being evaluated atomically. For that, we notice that a call solve y will never modify the value $\rho[x]$ of an unknown x with key$[x] >$ key$[y]$. During evaluation of right-hand sides, a recursive call to solve may only occur for an unknown y that has not been considered before, i.e., is fresh. Therefore, it will not affect any unknown that has been encountered earlier. From that, we conclude that reevaluating a right-hand side f_x for ρ immediately after a call f_x (eval x), will return the same value — but by a computation that does not change ρ and thus is atomic.

In order to prove that $\mathbf{SLR}_1$ is a local generic solver, it therefore remains to verify that upon termination, ρ is a partial $\Box$-solution with $x_0 \in \mathsf{dom}$. Since x_0 is initialized before $\mathsf{solve}\,x_0$ is called, x_0 must be contained in dom. Upon termination, evaluation of no unknown is still in process and the priority queue is empty. All unknowns in $\mathsf{dom} \setminus \mathsf{stable}$ are either fresh and therefore solved right-away, or non-fresh and then inserted into the priority queue. Therefore, we conclude that the equation $\rho[x] = \rho[x] \;\Box\; f_x\,\rho$ holds for all $x \in \mathsf{dom}$. Furthermore, the invariant for the map infl implies that upon termination, $x \in \mathsf{infl}[y]$ whenever $x = y$ or $y \in \mathsf{dep}_x\,\rho$. In particular, infl is defined for y implying that $y \in \mathsf{dom}$.

In summary, correctness of the algorithm $\mathbf{SLR}_1$ follows from the stated invariants. The invariants themselves follow by induction on the number of function calls. Therefore, statement 1 holds.

For a proof of statement 2, assume that all equations are monotonic and only finitely many unknowns are encountered during the call $\mathsf{solve}\,x_0$. Let dom denote this set of unknowns. We proceed by induction on key values of unknowns in dom. First consider the unknown $x \in \mathsf{dom}$ with minimal key value. Then for all mappings ρ and infl, the call $\mathsf{solve}\,x$ will perform a sequence of updates to $\rho[x]$. In an initial segment of this sequence, the operator $\boxslash$ behaves like ∇. As soon as the same value $\rho[x]$ or a smaller value is obtained, the operator $\boxslash$ behaves like the operator Δ. Due to monotonicity, the remaining sequence may only consist of narrowing steps. By the properties of widening and narrowing operators, the sequence therefore must be finite.

Now consider a call $\mathsf{solve}\,x$ for an unknown $x \in \mathsf{dom}$ where by inductive hypothesis, $\mathsf{solve}\,y$ terminates for all unknowns y with smaller keys, and all mappings ρ, infl, sets stable and priority queue Q satisfy the invariants of the algorithm. In particular, this means that every recursive call to a fresh unknown terminates.

Assume for a contradiction that the assertion were wrong and the call to $\mathsf{solve}\,x$ would not terminate. Then this means that the unknown x must be destabilized after every evaluation of f_x ($\mathsf{eval}\,x$). Upon every successive call to $\mathsf{solve}\ x$, all unknowns with keys smaller than $\mathsf{key}[x]$ are no longer contained in Q and therefore are stable. Again we may deduce that the successive updates for $\rho[x]$ are computed by ∇ applied to the former value of $\rho[x]$ and a new value provided by the right-hand side for x, until a narrowing phase starts. Then, however, again due to monotonicity a decreasing sequence of values for $\rho[x]$ is encountered where each new value now is combined with the former value by means of Δ. Due to the properties of ∇ and Δ, we conclude that the iteration must terminate. ■

One limitation of using local solvers such as $\mathbf{SLR}_1$, is that nontermination may be encountered due to an ever growing number of explored unknowns. Multiple remedies have been suggested to cope with this situation. From some point on, the solver may stop to further explore newly appearing unknowns and decide to constantly return for these a safe value, e.g., the largest value $\top$ of $\mathbb{D}$ (if such a value exists). Alternatively, the solver may also introduce a partition of the unknowns and provide only a single value for each equivalence class. The latter solution has been proposed by Bourdoncle (1992). It may at least work for inter-procedural analysis where unknowns whose right-hand sides are closely related can naturally be grouped together.

6. Localized warrowing in SLR

So far we have applied the warrowing operator at every right-hand side. It has been long known for the 2-phase widening and narrowing approach, however, that precision can be gained by applying widening and thus also narrowing only at selected unknowns. These unknowns may be chosen freely, provided they form an *admissible set*, i.e., at least one unknown is selected for each loop in the dependence graph of the equations. When intertwining widening and narrowing by means of *structured round-robin* or *worklist iteration*, restricting $\boxslash$ to an admissible set of widening points may, however, no longer ensure termination of the resulting solvers.

Example 16. *Consider the same set of equations in the Example 6. According to our definition, the singleton set $\{x_2\}$ is admissible. Now assume that the $\boxslash$ operation is performed for the unknown x_2 only. With **SRR** we obtain the following sequence of updates:*

i		2	1	2	1	3	2	1	2	1	
x_1	0	0	∞	∞	1	1	1	∞	∞	2	...
x_2	0	∞	∞	1	1	1	∞	∞	2	2	...
x_3	0	0	0	0	0	1	1	1	1	1	...

*Whenever the value for x_3 increases, x_2 and x_1 receive the value $x_3 + 1$, implying that subsequently, x_3 further increased. A stable post-solution is never attained. A similar behavior can also be observed for **SW** on this example.* ■

Example 16 indicates that we cannot ignore the ordering on the unknowns x_i when selecting the points of application for $\boxslash$. Therefore, we refine the notion of admissibility as follows. Assume that we are given a system of equations $x_i = e_i, i = 1, \ldots, n$ where sets $\mathsf{dep}(x_i)$ of variable dependences are explicitly given. Then the set W of unknowns is called an *admissible set of $\boxslash$-points* if, in each cycle in the dependence graph of the equations, the unknown with the *highest index* is in W. We obtain:

Theorem 17. *Given a system of equations and an admissible set W of $\boxslash$-points, both the algorithm **SRR** and the algorithm **SW** are guaranteed to terminate when instantiated with $\square = \boxslash$, even when restricting the application of $\boxslash$ to unknowns in W only.*

Proof. The proofs are similar to those for the Theorems 8 and 11, respectively. Here, we only consider the assertion for **SW**. For the base case, note that if x_1 is the only unknown, either the right hand size of x_1 is a constant, or it refers to x_1 itself, in which case x_1 is in the set of $\boxslash$-points. In both cases, **SW** terminates. For the inductive case, assume x_n is extracted infinitely many times. First assume that x_n is contained in W. In this case, the proof proceeds as in Theorem 11. Now assume that x_n is not contained in W. Then there is no loop containing x_n which consists of variables with index at most n. In particular, this means that the set $\{x_1, \ldots, x_{n-1}\}$ can be split into disjoint subsets X_1, X_2, X_3. X_1 consists of all unknowns directly or indirectly depending on x_n, X_2 consists of the unknowns onto which x_n directly or indirectly depends, and X_3 contains the remaining unknowns. As soon as x_n is evaluated for the first time, the evaluation of the unknowns in X_2 and X_3 have already terminated. Therefore following an update of the unknown x_n, only unknowns from X_1 may be added to the worklist. Since none of these ever will cause x_n to be added to the worklist again, fixpoint iteration terminates by the inductive hypothesis. ■

Example 18. *According to the refined definition, the set $\{x_2\}$ in Example 16 is no longer admissible, whereas the set $\{x_3\}$ is. When restricting $\boxslash$ to the latter set, we obtain:*

i		2	1	3	2	1	3
x_1	0	0	1	1	1	∞	∞
x_2	0	1	1	1	∞	∞	∞
x_3	0	0	0	∞	∞	∞	∞

and the algorithm terminates. ■

In applications where dependences between unknowns may change, we cannot perform any pre-computation on the dependence graph between unknowns. In order to conveniently deal with these nonetheless, methods are required which determine admissible sets of $\boxslash$-points *on the fly.* Assume that we are given an assignment key of unknowns to priorities which are *linearly* ordered. Such an assignment enables us to dynamically identify *back-edges.* Here, a back-edge $y \to x$ consists of unknowns x, y where the value of y is queried in the right-hand side of x where $\mathsf{key}[x] \leq \mathsf{key}[y]$. Note that this does not correspond to the standard definition of back-edge, but we use the same terminology since both may be used to identify the head of loops. When y is the unknown with the highest priority in some loop, then a back-edge $y \to x$ along this loop is encountered. Therefore y is included into the set of admissible unknowns, i.e., those where $\boxslash$ is going to be applied. In all other cases, we may omit the application of $\boxslash$ and directly use the value of the right-hand side to determine the next value for the left-hand side. The resulting improvement to the solver, as shown in Fig. 7, is called $\mathbf{SLR}_2$.

Interestingly for our suite of benchmark programs, the algorithm $\mathbf{SLR}_2$ did not significantly improve the precision of the resulting interval analysis. Consider, e.g., the program in Fig. 8. The control-flow graph corresponding to this program is shown in Fig. 9 where each node v is marked with the priority assigned to v when the function solve of $\mathbf{SLR}_1$ is called for the endpoint of the program for an interval analysis. We are looking for nodes that influence nodes with smaller priority. In the example, these are the nodes with priorities -1 and -5, respectively, i.e., exactly the loop heads. After the first iteration for interval

```
let rec solve $x$ =
  wpx $\leftarrow$ if $x \in$ wpoint
    then true else false;
  if $x \notin$ stable then
     stable $\leftarrow$ stable $\cup \{x\}$;
     tmp $\leftarrow$ if wpx
       then $\rho[x] \boxslash f_x$ (eval $x$)
       else $f_x$ (eval $x$);
     if tmp $\neq \rho[x]$ then
       $\rho[x]$ $\leftarrow$ tmp;
       W $\leftarrow$ if wpx
         then infl$[x] \cup \{x\}$
         else infl$[x]$;
       add $Q$ W;
       infl$[x]$ $\leftarrow$ $\emptyset$;
       stable $\leftarrow$ stable $\setminus$ W;
       while $(Q \neq \emptyset) \wedge$
          (min_key $Q \leq$ key$[x]$) do
         solve (extract_min $Q$);
     end
  end
and init $y$ =
  dom $\leftarrow$ dom $\cup \{y\}$;
  key$[y]$ $\leftarrow$ $-$count;
  count++;
  infl$[y]$ $\leftarrow$ $\emptyset$;
  $\rho[y]$ $\leftarrow$ $\rho_0[y]$
and eval $x$ $y$ =
  if $y \notin$ dom then
    init $y$; solve $y$;
  if key$[x] \leq$ key$[y]$ then
    wpoint $\leftarrow$ wpoint $\cup \{y\}$;
  infl$[y]$ $\leftarrow$ infl$[y] \cup \{x\}$;
  $\rho[y]$
in
  wpoint $\leftarrow$ $\emptyset$
  stable $\leftarrow$ $\emptyset$; infl $\leftarrow$ $\emptyset$;
  $\rho \leftarrow \emptyset$; dom $\leftarrow$ $\emptyset$;
  $Q$ $\leftarrow$ empty_queue();
  count $\leftarrow$ 0; init $x_0$;
  solve $x_0$;
  $\rho$
```

Figure 7: The algorithm **SLR**$_2$, which is **SLR** with plain localized widening. Colored in red are changes w.r.t. **SLR**$_1$.

```
$i$ = 0;
while ($i$ < 100) {
  $j$ = 0;
  while ($j$ < 10) {
    // Inv: $0 \leq i \leq 99$
    $j = j + 1$;
  }
  $i = i + j$;
}
```

Figure 8: Example program with nested loops.

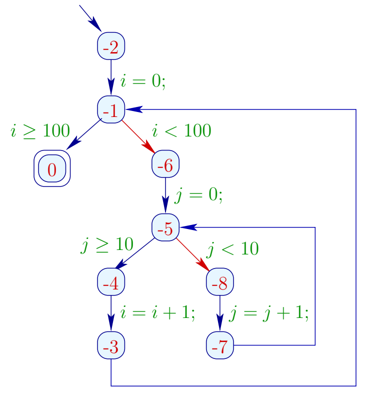

Figure 9: The control-flow graph of the program.

analysis on this program, the interval $[0,0]$ has been established for the program variable i at all program points of the inner loop. Then a second iteration of the outer loop is performed. Even if warrowing is only applied at the loop heads, we obtain the interval $[0,\infty]$ for i at the loop head of the outer loop. In the subsequent iteration of the inner loop, the new interval for variable i at the inner loop head is $[0,99]$. Since warrowing is meant to be applied at that program point, the interval $[0,0] \boxslash [0,99] = [0,\infty]$ is recorded for i and subsequently also propagated to all the other program points of the inner loop, and no subsequent narrowing will take place to recover from the loss of the upper bound for i.

This kind of loss of precision is avoided if we allow the set wpoint of unknowns where to apply $\boxslash$ not only to grow monotonically, but also to *shrink*. Our second idea therefore is to *remove* an unknown x from wpoint before the right-hand side of x is evaluated. The resulting algorithm $\mathbf{SLR}_3$ is shown in Fig. 10. Note that back-edges are detected by the call $\mathsf{eval}\ x\ y$ which therefore may insert y into the set wpoint, while the unknown x is removed from wpoint inside the call $\mathsf{solve}\ x$.

Theorem 19. *Assume the algorithm $\boldsymbol{SLR}_3$ is applied to a system of pure equations over an upward-directed set $\mathbb{D}$, with initially queried* interesting *unknown x_0.*

1. *$\boldsymbol{SLR}_3$ returns a partial post solution whose domain contains x_0—whenever it terminates.*
2. *If each right hand side is monotonic, then $\boldsymbol{SLR}_3$ is guaranteed to terminate—whenever only finitely many unknowns are encountered.*

PROOF. The considerations in the original proof for $\mathbf{SLR}_1$ regarding atomicity of evaluation of right-hand sides still hold. The same is true for partial correctness. The only difference w.r.t. $\mathbf{SLR}_1$ is that, upon termination, for an unknown x either $\rho[x] = \rho[x] \boxslash f_x\rho$ or $\rho[x] = f_x\rho$. In any case, ρ is a post-solution.

The most interesting part is the proof of termination. So, assume that all right hand sides are monotonic and only finitely many unknowns are encountered during the call of $\mathsf{solve}\ x_0$. Assume the algorithm does not terminate. It means there are unknowns x whose values $\rho[x]$ are updated infinitely many times. Let x denote one of these unknowns, namely the one with maximum priority. From a certain point in the execution of the algorithm, no fresh unknown is encountered and no $\rho[y]$ for an unknown y with key value exceeding $\mathsf{key}[x]$ is ever updated.

Assume we have reached this point in the execution of the algorithm. Moreover, assume that x is extracted. This means that in the queue there are no unknowns with key value less than $\mathsf{key}[x]$. Since all unknowns with key values greater than $\mathsf{key}[x]$ are not subject to update (hence their evaluation does not add elements to the queue), for x to be extracted repeatedly, the only possibility is that in every subsequent call $\mathsf{solve}\ x$, we have:

1. $tmp \neq \rho[x]$; and
2. there is some unknown $y \in \mathsf{infl}[x]$ with $\mathsf{key}[y] \leq \mathsf{key}[x]$ so that the evaluation of the right-hand side of y queries the value of the unknown x.

```
let rec solve $x$ =
  wpx ← if $x \in$ wpoint
     then true else false;
  wpoint ← wpoint $\setminus \{x\}$;
  if $x \notin$ stable then
     stable ← stable $\cup \{x\}$;
     tmp ← if wpx
       then $\rho[x] \boxslash f_x$ (eval $x$)
       else $f_x$ (eval $x$);
     if tmp $\neq \rho[x]$ then
       $\rho[x]$ ← tmp;
       W ← if wpx
         then infl$[x] \cup \{x\}$
         else infl$[x]$;
       add $Q$ W;
       infl$[x]$ ← $\emptyset$;
       stable ← stable $\setminus$ W;
       while $(Q \neq \emptyset) \wedge$
          (min_key $Q \leq$ key$[x]$) do
         solve (extract_min $Q$);
     end
  end
and init $y$ =
  dom ← dom $\cup \{y\}$;
  key$[y]$ ← −count;
  count++;
  infl$[y]$ ← $\emptyset$;
  $\rho[y]$ ← $\rho_0[y]$
and eval $x$ $y$ =
  if $y \notin$ dom then
    init $y$; solve $y$;
  if key$[x] \leq$ key$[y]$ then
    wpoint ← wpoint $\cup \{y\}$;
  infl$[y]$ ← infl$[y] \cup \{x\}$;
  $\rho[y]$
in
  wpoint ← $\emptyset$
  stable ← $\emptyset$; infl ← $\emptyset$;
  $\rho$ ← $\emptyset$; dom ← $\emptyset$;
  $Q$ ← empty_queue();
  count ← 0; init $x_0$;
  solve $x_0$;
  $\rho$
```

Figure 10: The algorithm **SLR**$_3$, which is **SLR**$_2$ with localized warrowing. Colored in red are changes w.r.t. **SLR**$_2$.

Assume for a contradiction, that the second assertion were wrong. Then the evaluation of the right-hand side of every unknown y with key$[y] \leq$ key$[x]$ returns the current value of y. Therefore, the next evaluation of the right-hand side of x also will return the same value as the last evaluation the right-hand side of x — implying that the call solve x terminates: in contradiction to our assumption. Therefore, the second assertion holds. Now, due to the second assertion, x is inserted into the set wpoint before the procedure solve is tail-recursively called for x. Accordingly, wpx will always be true when evaluating solve x. However, by the properties of $\boxslash$, this means that x cannot be updated infinitely many times: contradiction. Therefore the algorithm terminates. ■

Let us again consider the program from Fig. 8. The solver **SLR**$_3$ iterates through the program points of the inner loop until stabilization before the next iteration on the program points of the outer loop is performed. After this iteration, the interval $[0, 0]$ has been established for the program variable at all program points of the inner loop. Since the unknown corresponding to the loop head of the inner loop is now stable, it is no longer contained in the set wpoint. Therefore, when during the next iteration of the outer loop the interval $[0, 99]$ arrives for program variable i, this interval will replace the current interval $[0, 0]$ for i (without application of the warrowing operator). Accordingly, the

```
i = 0;
while (TRUE) {
  i = i + 1;
  j = 0;
  while (j < 10) {
    // Inv: 1 ≤ i ≤ 10
    j = j + 1;
  }
  if (i > 9) i = 0;
}
```

Figure 11: Example program hybrid from Halbwachs and Henry (2012).

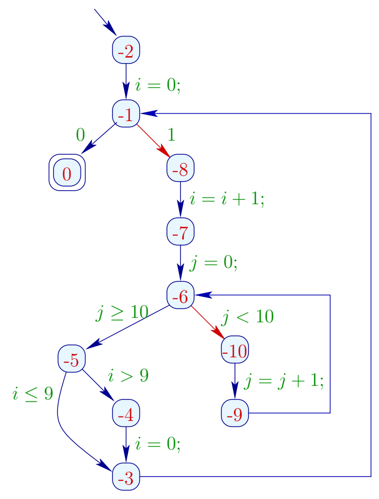

Figure 12: The control-flow graph for the program from Fig. 11.

subsequent iteration on the inner loop will propagate this interval throughout the inner loop without change. Therefore no upper bound ∞ for i is ever generated within the inner loop. This effect is comparable to the concept of localized widening as proposed by Amato and Scozzari (2013).

7. Restarting in SLR

Besides localization of widening and narrowing, Amato and Scozzari (2013) present a second idea to improve precision of fixpoint iteration in presence of infinite increasing chains. Consider the program in Fig. 11 whose control-flow graph is given in Fig. 12. In this example, the program variable i takes values from the interval $[0, 10]$ whenever the inner loop is entered. The upper bound 10, though, is missed both by the vanilla version of **SLR** as well as by $\mathbf{SLR}_3$. The reason is that the inner loop is iterated with the interval $[1, \infty]$ for i until stabilization before that, triggered by a narrowing iteration of the outer loop, the value $[1, 10]$ for i arrives at the entry point of the inner loop. Since $[1, 10] \sqcup [1, \infty] = [1, \infty]$, the finite upper bound of i at the entry point cannot be recovered.

In order to improve on this and similar kinds of precision loss, Amato and Scozzari propose to *restart* the iteration for sub-programs. The restart could be triggered, e.g., for the body of a loop as soon as the value for the head has decreased.

In the following, we indicate how this strategy may be integrated into the generic solver $\mathbf{SLR}_3$ (see Fig. 13). The resulting algorithm $\mathbf{SLR}_4$ requires a function restart. This function, when called with a priority r and an unknown x,

```
let rec restart r $y$ =
    add $Q$ $y$;
    stable $\leftarrow$ stable $\setminus \{y\}$;
    if key$[y]$ < r then
      $\rho[y]$ $\leftarrow$ $\rho_0[y]$;
      M $\leftarrow$ infl$[y]$;
      infl$[y]$ $\leftarrow$ $\emptyset$;
      foreach $z \in$ M do
        restart r $z$
in
let rec solve $x$ =
  wpx $\leftarrow$ if $x \in$ wpoint
     then true else false;
  wpoint $\leftarrow$ wpoint $\setminus \{x\}$;
  if $x \notin$ stable then
     stable $\leftarrow$ stable $\cup \{x\}$;
     tmp $\leftarrow$ if wpx
       then $\rho[x] \boxslash f_x$ (eval $x$)
       else $f_x$ (eval $x$);
    if tmp $\neq \rho[x]$ then
      if wpx $\wedge$ tmp $\sqsubseteq \rho[x]$ then
       foreach $z \in$ infl$[x] \cup \{x\}$
         do restart key$[x]$ $z$;
      else
        W $\leftarrow$ if wpx
          then infl$[x] \cup \{x\}$
          else infl$[x]$;
        add $Q$ W;
        stable $\leftarrow$ stable $\setminus$ W;
        infl$[x]$ $\leftarrow$ $\emptyset$;
        $\rho[x]$ $\leftarrow$ tmp;
        while $(Q \neq \emptyset) \wedge$
          (min_key $Q \leq$ key$[x]$) do
           solve (extract_min $Q$);
      end
    end
and init $y$ =
  dom $\leftarrow$ dom $\cup \{y\}$;
  key$[y]$ $\leftarrow$ $-$count;
  count++;
  infl$[y]$ $\leftarrow$ $\emptyset$;
  $\rho[y]$ $\leftarrow$ $\rho_0[y]$
and eval $x$ $y$ =
  if $y \notin$ dom then
    init $y$; solve $y$;
  if key$[x] \leq$ key$[y]$ then
    wpoint $\leftarrow$ wpoint $\cup \{y\}$;
  infl$[y]$ $\leftarrow$ infl$[y] \cup \{x\}$;
  $\rho[y]$
in
  wpoint $\leftarrow$ $\emptyset$
  stable $\leftarrow$ $\emptyset$; infl $\leftarrow$ $\emptyset$;
  $\rho \leftarrow \emptyset$; dom $\leftarrow$ $\emptyset$;
  $Q$ $\leftarrow$ empty_queue();
  count $\leftarrow$ 0; init $x_0$;
  solve $x_0$;
  $\rho$
```

Figure 13: Solver $\mathbf{SLR}_4$. Colored in red are changes w.r.t. $\mathbf{SLR}_3$.

traverses variables y that are recursively influenced by x such that the priority of y is less than r. As the main effect of restart, the value $\rho[y]$ is reset to $\rho_0[y]$, for each found unknown y. Additionally, to force re-computation, all influenced unknowns are added to the priority queue Q, removed them from the stable set, and have their their influence sets cleared. The function restart then is called within the function solve for an unknown x whenever x is currently contained in wpoint and the new value tmp for x is less than the current value for x. In this case, all unknowns in the set infl$[x]$ are restarted (w.r.t. the priority of x). Otherwise, the algorithm behaves like the algorithm $\mathbf{SLR}_3$.

Consider again the program from Fig. 11. As soon as narrowing at the head of the outer loop recovers the interval $[0, 9]$ for the program variable i, recursively the values for the reachable program points with lower priorities are reset to $\bot$. This refers to all program points in the body of the outer loop and thus also to the complete inner loop. Reevaluation of all these program points with the value $[0, 9]$ for i at the outer loop head provides us with the invariant $1 \leq i \leq 10$ throughout the inner loop.

The algorithm will return a $\boxslash$-solution whenever it terminates. A guarantee, however, of termination is no longer possible even if right-hand sides are monotonic and only finitely many unknowns are visited. Intuitively, the reason is the following. Assume that the value for an unknown x has decreased. Then we might expect that restarting the iteration for lower priority unknowns results in a smaller next approximation for x. Due to the non-monotonicity introduced by widening, this need not necessarily be the case. Accordingly, we are no longer able to bound the number of switches between increasing and decreasing phases for x. There are simple practical remedies for nontermination, though. We may, for example, bound for each unknown the number of restarts which do not lead to the same value or a decrease. This behaviour is somewhat different from the restart policy of Amato and Scozzari (2013) where nontermination cannot happen, due to the fact that the algorithm keeps track of which (ascending or descending) phase is executed in a given program point, and a descending phase cannot turn into an ascending phase just because the current values of some unknowns have been decreased.

8. Side-effecting systems of equations

In the following, generic solving, as we have discussed in the preceding sections, is extended to right-hand sides f_x that not only return a value for the left-hand side x of the equation $x = f_x$, but additionally may produce *side effects* to other unknowns. This extension to equation systems corresponds to *assert*-statements of Prolog or Datalog programs. Side effects may arbitrarily be dispersed over right-hand sides of unknowns. This kind of extension to right-hand sides has been advocated by Apinis et al. (2012) for an elegant specification of inter-procedural analysis using partial contexts and flow-insensitive unknowns and thus also of multi-threaded programs (Seidl et al., 2003). In that paper it is argued that, at least for very large or infinite systems, side effects cannot easily be simulated by ordinary systems of equations.

Example 20. *Consider the following program.*

```
int g = 0;
void f(int b) {
  if (b) g = b + 1;
  else g = −b − 1;
}
int main() {
  f(1);
  f(2);
  return 0;
}
```

The goal is to determine a tight interval for the global program variable g. A flow-insensitive analysis of globals aims at computing a single interval which should comprise all values possibly assigned to g. Besides the initialization with 0, this program has two assignments, one inside the call $f(1)$, the other inside the call $f(2)$. A context-sensitive analysis of the control-flow should therefore collect the three values $0, 2, 3$ and combine them into the interval $[0, 3]$ for g. This requires to record for which contexts the function f is called. This task can nicely be accomplished by means of a local solver. That solver, however, has to be extended to deal with the contributions to global unknowns. ■

In general, several side effects may occur to the same unknown z. Over an arbitrary domain of values, though, it remains unclear how the multiple contributions to z should be combined. Therefore in this section, we assume that the values of unknowns are taken from an upward-directed set $\mathbb{D}$ with a least element, which is denoted by $\bot$. Also, right-hand sides are again assumed to be *pure*. For side-effecting constraint systems this means that evaluating a right-hand side f_x applied to functions $\mathsf{get} : X \to \mathbb{D}$ and $\mathsf{side} : X \to \mathbb{D} \to \mathbf{unit}$ (**unit** being the type consisting of the empty tuple only) consists of a sequence of value lookups for unknowns by means of calls to the first argument function get and *side effects* to unknowns by means of calls to the second argument function side which is terminated by returning a contribution in $\mathbb{D}$ for the corresponding left-hand side.

Subsequently, we assume that each right-hand side f_x produces no side effect to x itself and at most one side effect to each unknown $z \neq x$. Technically, the right-hand side f_x of x with side effects can be considered as a succinct representation of a function $\bar{f}_x$ that takes a mapping ρ and does not return just a single value, but again another mapping ρ' where $\rho'[x]$ equals the return value computed by f_x for $\mathsf{get} = \rho$, and for $z \neq x$, $\rho'[z] = d$ if during evaluation of f_x get side, side is called for z and d. Otherwise, i.e., if no side effect occurs to z, $\rho'[z] = \bot$. A post solution of a system $x = f_x, x \in X$, of equations with side effects then is a mapping $\rho : X \to \mathbb{D}$ such that for every $x \in X$, $\rho \sqsupseteq \bar{f}_x\,\rho$. A *partial* post solution with domain $\mathsf{dom} \subseteq X$ is a mapping $\rho : \mathsf{dom} \to \mathbb{D}$ such that for every $x \in \mathsf{dom}$, evaluation of f_x for ρ accesses only unknowns in dom and also produces side effects only to unknowns in dom; moreover, $\bar{\rho} \sqsupseteq \bar{f}_x\,\bar{\rho}$ where $\bar{\rho}$ is the total variable assignment obtained from ρ by setting $\bar{\rho}[y] \leftarrow \bot$ for all $y \notin \mathsf{dom}$.

In the following, we present the side-effecting variant $\mathbf{SLR}_1^+$ of the algorithm $\mathbf{SLR}_1$ from Section 5 that for such systems returns a partial $\Box$-solution—whenever it terminates. Moreover, the enhanced solver $\mathbf{SLR}_1^+$ is guaranteed to terminate whenever all right-hand sides f_x are *monotonic*, i.e., the functions $\bar{f}_x$ all are monotonic.

Example 21. *Consider again the analysis of Example 20. The contributions to the global program variable g by different contexts may well be combined individually by widening to the current value of the global. When it comes to narrowing, though, an individual combination may no longer be sound. Therefore, the extension of the local solver $\boldsymbol{SLR}_1$ should collect all occurring contributions* *into a* set*, and use the* joint value *of all these to possibly improve the value of g.* ■

Conceptually, the algorithm $\mathbf{SLR}_1^+$ therefore creates for each side effect to unknown z inside the right-hand side of x, a fresh unknown $s_{x,z}$ which receives that single value during evaluation of the right-hand side f_x. Furthermore, the algorithm maintains for every unknown z an auxiliary set $\mathsf{set}[z]$ which consists of all unknowns x whose right-hand sides may possibly contribute to the value of z by means of side effects. Accordingly, the original system of side-effecting equations is (implicitly) transformed in the following way:

1. Inside a right-hand side f_x, when a side effect $\mathsf{side}\,z\,d$ is encountered, the value d is stored inside the auxiliary unknown $s_{x,z}$ while the unknown x is added to the set $\mathsf{set}[z]$.

2. The new right-hand side for an unknown x is extended with a least upper bound of all $s_{z,x}$, $z \in \mathsf{set}[x]$.

The warrowing operator is applied whenever the return value of the new right-hand side for x is combined with the previous value of x. Let us now list the required modifications of the algorithm $\mathbf{SLR}_1$.

First, the function $\mathsf{init}\,y$ is extended with an extra initialization of the set $\mathsf{set}[y]$ with $\emptyset$. The function eval remains unchanged. Additionally, a function evalSide is required for realizing the side effects during an evaluation of a right-hand side. As eval, the function evalSide also receives the left-hand side of the equation under consideration as its first argument. We define:

```
let evalSide $x$ $y$ $d$ =
            if $s_{x,y} \notin$ dom then $\rho[s_{x,y}] \leftarrow \bot$;
            if $d \neq \rho[s_{x,y}]$ then
              $\rho[s_{x,y}] \leftarrow d$;
              if $y \in$ dom then
                set[$y$] $\leftarrow$ set[$y$] $\cup \{x\}$;
                stable $\leftarrow$ stable $\setminus \{y\}$;
                add $Q$ $y$
              else
                init $y$; set[$y$] $\leftarrow \{x\}$;
                solve $y$
              end
            end
```

When called with x, y, d, the function evalSide first initializes the unknown $s_{x,y}$ if it is not yet contained in dom. If the new value is different from the old value of ρ for $s_{x,y}$, $\rho[s_{x,y}]$ is updated. Subsequently, the set $\mathsf{set}[y]$ receives the unknown x, and the unknown y is triggered for reevaluation. If y has not yet been encountered, y is initialized, $\mathsf{set}[y]$ is set to $\{x\}$, and $\mathsf{solve}\,y$ is called. Otherwise, x is only added to $\mathsf{set}[y]$, and y is scheduled for reevaluation by destabilizing y first and then inserting y into the priority queue Q.

The third modification concerns the procedure solve. There, the call of the right-hand side f_x now receives $\mathsf{evalSide}\,x$ as a second argument and additionally evaluates all unknowns collected in $\mathsf{set}[x]$. The corresponding new line reads:

$$\mathsf{tmp} \leftarrow \rho[x] \boxslash (f_x\,(\mathsf{eval}\,x)\;(\mathsf{evalSide}\,x) \sqcup \bigsqcup\{\rho[s_{z,x}] \mid z \in \mathsf{set}[x]\});$$

Example 22. *Consider again interval analysis for the program from Example 20. Concerning the global program variable g, the initialization $g = 0$ is detected first, resulting in the value $\rho[g] = [0,0]$. Then g is scheduled for reevaluation. This occurs immediately, resulting in no further change. Then the calls $f(1), f(2)$ are analyzed, the side effects of 2 and 3 are recorded and g is rescheduled for evaluation. When that happens, the value $\rho[g]$ is increased to*

$$[0,0] \boxslash [0,3] = [0,0] \nabla [0,3] = [0,\infty]$$

if the standard widening for intervals is applied. Since $\rho[g]$ has changed, z again is scheduled for evaluation resulting in the value

$$[0,\infty] \boxslash [0,3] = [0,\infty] \Delta [0,3] = [0,3]$$

Further evaluation of g will not change this result any more. ■

Analogously to Theorem 15 from the last section, we obtain:

Theorem 23. *Assume the algorithm $\boldsymbol{SLR}_1^+$ is applied to a system of pure equations with side effects over an upward-directed set $\mathbb{D}$ with bottom, with initially queried* interesting *unknown x_0.*

1. *$\boldsymbol{SLR}_1^+$ returns a partial post solution whose domain contains x_0—whenever it terminates.*

2. *If each right hand side is monotonic and $\sqcup$ is monotonic as well, then $\boldsymbol{SLR}_1^+$ is guaranteed to terminate—whenever only finitely many unknowns are encountered and side effects of low priority variables' right-hand sides always refer to higher priority variables.*

Note that in the proof of termination we also require the upper bound operator $\sqcup$ to be monotone. The property trivially holds when $\mathbb{D}$ is a join semi-lattice and $\sqcup$ is the least upper bound. However, there are some abstract domains which are not join semi-lattices, such as zonotopes (Goubault et al., 2012) or parallelotopes (Amato and Scozzari, 2012).

The proof of Theorem 23 is analogous to the proof of Theorem 15. It is worthwhile noting, though, that the argument there breaks down if the assumption on the priorities in side effects is not met: in that case, any reevaluation of a high-priority variable x may have another effect onto a low-priority variable y — even if x does not change. No guarantee therefore can be given that the overall sequence of values for y will eventually become stable. If on the other hand, the side-effected variable y has priority greater than x, at reevaluation time of y, the evaluation of x has already terminated where only the final contributions to y are taken into account. Since only finitely many such contributions are possible, the algorithm is overall guaranteed to terminate.

The extra condition on the side effects incurred during fixpoint computation is indeed crucial for enforcing termination — as can be seen from the following example.

Example 24. *Consider the following program:*

```
int g = 0;
int main() {
  g = g + 1;
  return 0;
}
```

where the global is meant to be analyzed flow-insensitively. Consider an interval analysis by means of solver $\boldsymbol{SLR}_1^+$*, and assume that the unknown for the global* g *has lesser priority than the unknown for the endpoint of the assignment to* g*. The first side effect to* g *is the interval* $[1,1]$ *resulting in the new value* $[0,1]$ *which is combined with the old value* $[0,0]$ *by means of* $\boxslash$ *and then again by means of* $\boxslash$*. Since*

$$([0,0] \boxslash [0,1]) \boxslash [0,1] = [0,\infty] \boxslash [0,1] = [0,1]$$

the widening is immediately compensated by the consecutive narrowing. The same phenomenon occurs at every successive update of the value for g*, implying that* $\boldsymbol{SLR}_1^+$ *will not terminate.*

The solver $\boldsymbol{SLR}_1^+$ *behaves differently if the priority of the unknown for* g *exceeds the priority of the unknown for the endpoint of the assignment. In this case after the first application of* $\boxslash$ *at* g*, the assignment is processed again. Since the first application of* $\boxslash$ *behaves like a widening, this means that the second side effect to* g *is with the interval* $[1,\infty]$*. Accordingly, the following recomputation of the new value for* g *will be*

$$[0,\infty] \boxslash ([0,0] \sqcup [1,\infty]) = [0,\infty] \boxslash [0,\infty] = [0,\infty]$$

and the fixpoint computation terminates.■

In practical applications where the side-effected unknowns correspond to globals, the extra condition on priorities in Theorem 23 can be enforced, e.g., by ensuring that the initializers of globals are always analyzed *before* the call to the procedure main.

Theorem 23 only discusses the extension of the base version of the algorithm $\mathbf{SLR}_1$ to systems of equations with side effects. A similar extension is also possible to the solvers with localized application of $\boxslash$. In order to ensure termination also in this case, however, we additionally must insert every side-effected unknown into the set wpoint of unknowns where the operation $\boxslash$ is to be applied. For the side-effecting version of $\mathbf{SLR}_3$, we therefore define:

```
let evalSide $x$ $y$ $d$ =
            wpoint $\leftarrow$ wpoint $\cup \{y\}$;
            if $s_{x,y} \notin$ dom then $\rho[s_{x,y}] \leftarrow \bot$;
            if $d \neq \rho[s_{x,y}]$ then
              $\rho[s_{x,y}] \leftarrow d$;
              if $y \in$ dom then
                set$[y] \leftarrow$ set$[y] \cup \{x\}$;
                stable $\leftarrow$ stable $\setminus \{y\}$;
                add $Q$ $y$
              else
                init $y$; set$[y] \leftarrow \{x\}$;
                solve $y$
              end
            end
```

With this definition, termination of the algorithm $\mathbf{SLR}_3^+$ can be guaranteed under the same assumptions as for the algorithm $\mathbf{SLR}_1^+$.

9. Experimental evaluation

We have implemented the proposed generic local solvers and included them into the analyzer GOBLINT for multi-threaded C programs. GOBLINT uses CIL as C front-end (Necula et al., 2002) and is written in OCAML. The tests were performed on 2.7GHz Intel Core i7 laptop, with 8GB DDR3 RAM, running OS X 10.9.

In a first series of experiments we tried to clarify the increase of precision possibly attained by means of the various $\boxslash$-solvers w.r.t. the two-phase solving using widening and narrowing according to (Cousot and Cousot, 1976). For these experiments, we used the benchmark suite[1] from the Märdalen WCET research group (Gustafsson et al., 2010) which collects a series of interesting small examples for WCET analysis, varying in size from about 40 lines to 4000 lines of code. We have extended this benchmark suite with four tricky programs from (Amato and Scozzari, 2013): a) `hh.c`, b) `hybrid.c`, c) `nested.c`, and d) `nested2.c`. The basis of our analysis is constant propagation together with a may-points-to analysis of pointer variables, which is flow-sensitive for locals and flow-insensitive for globals. On top of that, we performed an interval analysis. In contrast to the preliminary experiments of Apinis et al. (2013), we now use an interval analysis which soundly approximates 32bit integers with wrap-around semantics. This means that the abstract semantics of arithmetic operations now returns $\top = [\mathsf{minint}, \mathsf{maxint}]$ if an overflow may occur. Likewise, the widening operator widens finite lower and upper bounds to minint and maxint, respectively. Additionally, we have considered the relational abstract numerical domains of

[1]available at `www.mrtc.mdh.se/projects/wcet/benchmarks.html`

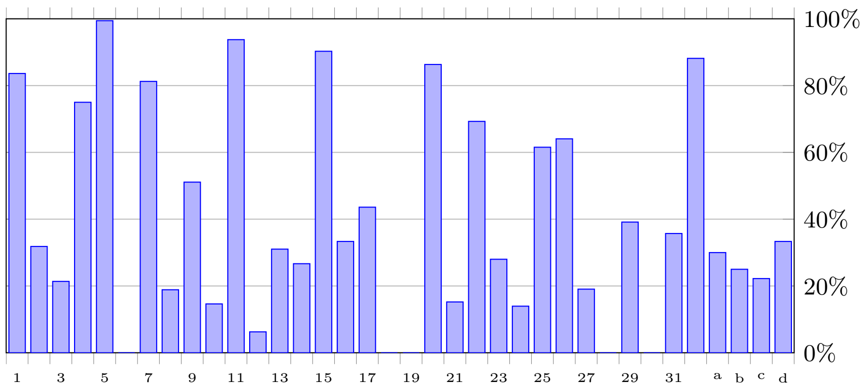

Figure 14: The relative improvement of $\mathbf{SLR}_1$ over two-phase solving.

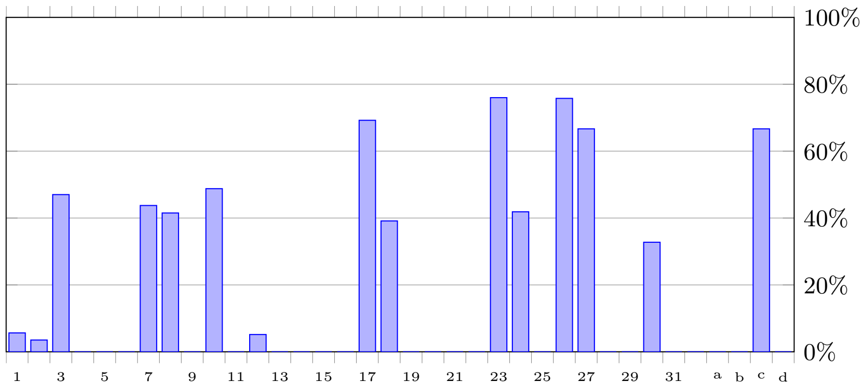

Figure 15: The relative improvement of $\mathbf{SLR}_3$ over $\mathbf{SLR}_2$.

octagons and polyhedra, respectively. The implementation of these is based on the Apron library (Jeannet and Miné, 2009). First, we determined the relative precisions achieved by the various solvers w.r.t. the interval domain only. The results of this comparison is displayed in Figs. 14, 15, and 16. Since the absolute run-times are negligible (about 14 seconds for all programs together), we only display the relative precision. Fig. 14 reports the percentage of program points where solver $\mathbf{SLR}_1$ returns better results than two-phase solving. In the vast majority of cases, $\mathbf{SLR}_1$ returned significantly better results—supporting the claim that $\boxslash$-solving may improve the precision. Interestingly, placing the warrowing operator at widening points only, as implemented by solver $\mathbf{SLR}_2$, results in an improvement in the first two benchmarks (28.81% and 1.01%, respectively) only. The reason might be that, applying narrowing intertwined

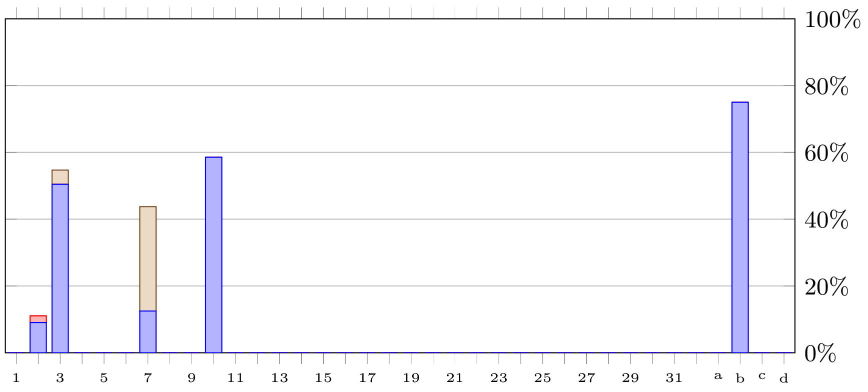

Figure 16: Comparison of $\mathbf{SLR}_4$ with $\mathbf{SLR}_3$ indicating the percentage of program points where the results are incomparable (brown), better (blue) or worse (red).

with widening can quite often recover some of the precision lost by the superfluous widenings.

Fig. 15 reports the relative further improvement when additionally widening points can dynamically be removed during solving. In 15 of 36 cases, we again obtain an improvement, in some cases even for over 70% of program points! This strategy therefore seems highly recommendable to achieve good precision. Fig. 16 finally explores the impact of restarting. Here, the picture is not so clear. For the benchmark program 2, restarting resulted even in a loss of precision for a small fraction of program points, while still for a larger fraction improvements were obtained. In two further benchmarks, program points with incomparable results where found. For benchmark program 3, these make up about 4% of the program points, while for program 7, the fraction goes even up to 31%. In principle such a behavior is not surprising, considering the non-monotonicity of widening. Still, for two more example programs, drastic improvements are found. One of these comes from the WCET benchmark suite, while the other has been provided by (Amato and Scozzari, 2013), admittedly, as an example where restarting is beneficial.

In a second experiment, we explored the relative efficiencies of our implementation of the generic local $\boxslash$-solvers. For that, we performed interval analysis where local variables are analyzed depending on a calling context which includes all non-interval values of locals, while the values of globals are analyzed flow-insensitively. Such kinds of analysis cannot be performed by the two-phase approach, since right-hand sides are not monotonic and the sets of contexts and thus also the sets of unknowns encountered during the widening and narrowing phases may vary.

The analysis was run on all benchmarks from the SpecCpu2006 benchmark suite which can be handled by the C front-end CIL used in our analyzer. The set of selected benchmarks consist of seven programs in the range of 1 to 33

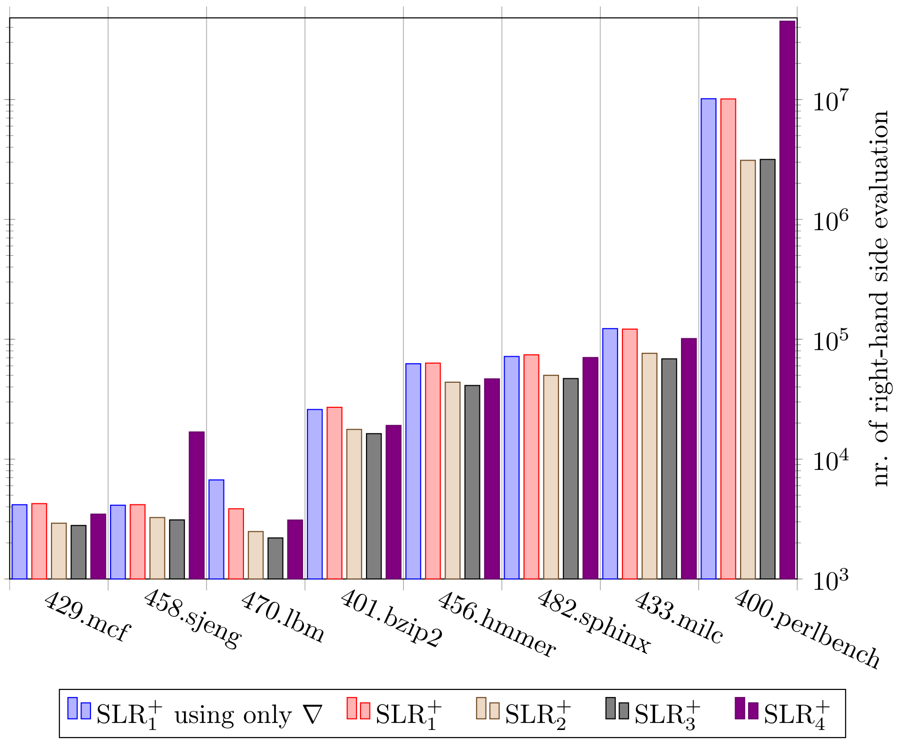

Figure 17: Context sensitive interval analysis of SpecCpu2006 programs.

kloc, `400.perlbench` with 175 kloc, and `445.gobmk` with 412 kloc of C code. The results for the side-effecting versions of $\mathbf{SLR}_1$ to $\mathbf{SLR}_4$ are reported in Fig. 17 where the numbers of evaluations of right-hand sides are displayed on a logarithmic scale. For a comparison we also included the numbers of evaluations if the solver $\mathbf{SLR}_1^+$ uses plain widening instead of $\boxslash$.

The analysis of the seven smaller programs could be handled in less than 13 seconds. The large program `400.perlbench` (175 kloc of C code) could be handled by our solvers — but with running times between 18 minutes (using $\mathbf{SLR}_3^+$) and 4 hours (using $\mathbf{SLR}_4^+$), while context-sensitive analysis did not terminate for the largest benchmark `445.gobmk` (412 kloc) within 5 hours.

The first observation is that $\mathbf{SLR}_1^+$ is only marginally slowed down, if widening is enhanced to $\boxslash$, i.e., narrowing is added. The second observation is that the efficiency of fixpoint computation is greatly improved when restricting the application of $\boxslash$ to widening points. Improvements of about 30% could consistently be obtained. For the large program `400.perlbench`, the speedup even was by a factor of 3. Enhancing solver $\mathbf{SLR}_2^+$ to solver $\mathbf{SLR}_3^+$, on the other hand, which comes with a significant improvement in precision, additionally results in another slight reduction of the number of evaluated right-hand sides. To us, these numbers came at a surprise, since even in those scenarios where we could theoretically establish termination of the algorithms, we expected drastically worse running times of iteration with $\boxslash$ when compared to iteration with widening alone.

Restarting, finally, adds another dimension of potential inefficiency to fixpoint iteration. Yet, our numbers for $\mathbf{SLR}_4^+$ on the benchmark suite show that the practical slowdown over the fastest solver $\mathbf{SLR}_3^+$ is in many cases still better than solving with $\mathbf{SLR}_1^+$ with widening alone. For the programs `458.sjeng` and `400.perlbench`, however, $\mathbf{SLR}_4^+$ is slower by a factor of 5 and 14, respectively.

As a third experiment, we compared the two-phase algorithm with the $\boxslash$-solver $\mathbf{SLR}_3^+$ for octagons and polyhedra. Since the results for these two relational domains are quite similar on our benchmark suite, we only report the numbers for polyhedra. This domain is quite expressive. On very small programs, we found virtually no differences in precision for the two fixpoint algorithms (up to two notable exceptions in favor of $\mathbf{SLR}_3$). Fig. 18 therefore only lists results for programs with at least fifty program points. On average, the number of computed invariants per program is then about 229. Improvements of 21.87% and 14.49% of $\mathbf{SLR}_3$ over the classical two-phase approach occur in the peak, while improvements around 5% are more common. No differences can be observed for more than half of the programs. In rare cases, a few invariants found by the two-phase approach could not be detected by $\mathbf{SLR}_3$. Again, due to the non-monotonic behavior of widening, such phenomena are to be expected.

10. Related work

Numerous attempts have been made to face the problem of the loss of precision introduced by widening operators. Some authors propose to avoid widening and compute a fixpoint of the Kleene iteration by using strategy/policy

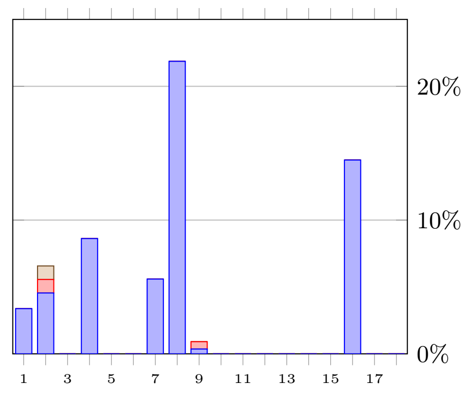

Figure 18: Comparison of 2-phase widening and narrowing with $\mathbf{SLR}_3$ indicating the percentage of program points where the results are incomparable (brown), where $\mathbf{SLR}_3$ is better (blue) or worse (red).

iteration (Costan et al., 2005; Gawlitza and Seidl, 2011) or acceleration operators (Gonnord and Halbwachs, 2006), but these methods are applicable only to specific abstract domains or under syntactical restrictions to the program syntax. In contrast, our approach is generally applicable, independently from the choice of the abstract domain and operators used in the analysis or any syntactical restrictions.

Another domain-independent approach is to design enhanced widening operators such as delayed widening, widening with threshold (Blanchet et al., 2003), widening with landmarks (Simon and King, 2006) and lookahead widening (Gopan and Reps, 2006). These are tailored for specific settings and abstract domains. They are orthogonal to our approach. They may be plugged into the warrowing operator, and thus be used together with our fixpoint algorithms.

Due to the presence of widening operators, it has been observed that the entire analysis fails to be monotonic. Therefore, selecting a different starting point of the analysis, other than the bottom of the abstract domain, may improve the overall result. In practice, this idea has been exploited by various techniques, all of which have in common to repeat the entire analysis multiple times with some variations, and to combine the obtained results. The proposal of Halbwachs and Henry (2012) is to iterate the analysis starting from a different initial value. After each widening/narrowing phase, the result is perturbed in order to get a new value to restart the widening/narrowing phase. The intersection of all obtained results is guaranteed to be a post-fixpoint. There are several approaches to choose the perturbation, but only the simplest one has been implemented so far. The main difference to our restarting solver is that the algorithm of Halbwachs and Henry (2012) restarts the widening/narrowing phase only after that a post fixpoint has been reached. On the contrary, in our approach the ascending and descending phases, perhaps together with restarting are intertwined and different program points can be in different phases during the same iteration.

Thus, these two approaches are technically incomparable. Still, computing a post solution before the restarting as well as after the restarting according to Halbwachs and Henry (2012) could very well be implemented, e.g., by one of our solvers in order to obtain the benefits of both approaches. (Amato and Scozzari, 2013) provide experimental evidence that localized widening with a standard separated narrowing is competitive with respect to this approach. These ideas are generalized by our solvers $\mathbf{SLR}_2$ and $\mathbf{SLR}_3$.

Guided static analysis as proposed by Gopan and Reps (2007) applies a standard program analysis to a sequence of program restrictions, each obtained by disabling some program paths. Each restriction is analyzed starting from the result of the previous one, until the original program is analyzed. Henry et al. (2012a) further enhanced this approach by combining it with path-focusing (Monniaux and Gonnord, 2011), in order to avoid merging infeasible paths and thus to find precise disjunctive invariants. Lookahead widening (Gopan and Reps, 2006) may be viewed as a variant of guided static analysis where the machinery needed to explore the different program restrictions is embedded, with some limitations, inside the widening operator. Amato and Scozzari (2013) give some evidence, though, that guided static analysis does not help in those cases where localized widening and intertwined widening and narrowing are beneficial.

Static analysis stratified by variable dependency, as proposed by Monniaux and Guen (2012), is similar to guided static analysis in that successive approximations of the program are considered, where later approximations consider more variables than former ones. The result of one approximation is used within the successive approximations to improve the results. All these techniques treat the equation solver as a black box, and try to execute different analyses to improve the result. In this sense, they are orthogonal to our engineering of solver algorithms and may therefore benefit from our improvements as well.

A recent proposal by Cousot (2015) is to add a third phase of iteration to the analysis, subsequently to the ascending (widening) and descending (narrowing) phases. In order to further improve on that post solution, another ascending iteration from the initial value is performed, now with a *dual-narrowing* operator. The dual narrowing operator allows to obtain a terminating iteration whose result is bounded above by the previously found post solution. Thus, similar to the approach of Halbwachs and Henry (2012), the third phase takes place only after a post solution has been reached. Currently, our approach only intertwines the ascending and descending phases — implying that for each unknown of the system, the solver may switch between widening and narrowing several times. In a technical sense, our setting is incomparable with the setting of Cousot (2015), as we do not require the initial value to be a pre solution. Accordingly, also the computed (partial) post solutions need not necessarily exceed the initial values. Still, it would be interesting to explore how far our solvers could be enhanced so that a third mode of iteration could be intertwined which uses a dual narrowing operator instead of $\boxslash$ — at least, whenever sensible.

11. Conclusion

We have presented a combination of widening and narrowing into a single operator $\boxslash$ and systematically explored solver algorithms which, when instantiated with $\boxslash$ will solve general systems of equations. Perhaps surprisingly, standard versions of fixpoint algorithms, when enhanced with $\boxslash$, may fail to terminate even for finite systems of monotonic equations. Therefore, we presented variants of *round-robin* iteration, of ordinary *worklist* iteration as well as of recursive local solving with and without side effects where for monotonic equations and finitely many unknowns, termination can be guaranteed whenever only finitely many unknowns are encountered, and side effects are to higher-priority unknowns only. In order to enforce termination, we assigned static priorities to the unknowns of the system. In order to construct *generic* solvers for arbitrary systems of equations, we heavily relied on self-observation of the solvers. Thus, priorities are assigned in the ordering in which the unknowns are encountered. We also let the fixpoint iterator itself determine the dependencies between unknowns. Together with the priorities, also the places where to apply warrowing are dynamically determined.

It has not been clear before-hand, though, how well the resulting algorithms behave for real-world program analyses. In order to explore this question, we have provided an implementation within the analysis framework Goblint. In our experimental set-up, we considered inter-procedural interval analysis where the monotonicity assumption is not necessarily met. Our experiments confirm that fixpoint iteration based on the combined warrowing operator still terminates and may increase precision considerably. This holds true already for the local solver $\mathbf{SLR}_1^+$ which has been presented by Apinis et al. (2013). Beyond that, we demonstrated that the add-on of localizing warrowing operators increases precision further, while efficiency is improved at the same time. An equally clear picture could not be identified for the extra add-on of restarting. While we found improvements in selected cases and generally still an acceptable efficiency, we also found exceptional cases where a (minor) loss of precision occurs at some program points or where the performance is degraded considerably.

In the end, we think that the two most important benefits of using the warrowing operator are:

- the increase in precision w.r.t. standard analysis with separate widening and narrowing phases;
- simpler implementation of solvers w.r.t. other solutions with separate and (especially) interleaved widening and narrowing phases (compare, for example, the complexity of the solver based on localized narrowing of Amato and Scozzari (2013) with the solver **SRR**).

The most significant improvements were achieved for standard interval analysis with the obvious widening and narrowing operators. For more expressive (and more expensive) domains such as octagons and polyhedra, still improvements could be observed in several cases. It remains for future work to explore how well our methods work also for other domains, for more sophisticated widening and narrowing operators and also for larger classes of programs. It would also be interesting to see whether other approaches such as static guided analysis by Gopan and Reps (2007) may benefit from our enhanced algorithms, and also in how far our ideas can be extended so that the third iteration phase by Cousot (2015) is incorporated into our solvers.

References

Amato, G., Scozzari, F., 2012. The abstract domain of parallelotopes. Electr. Notes Theor. Comput. Sci. 287, 17–28.

Amato, G., Scozzari, F., 2013. Localizing widening and narrowing. In: Logozzo, F., Fändrich, M. (Eds.), Static Analysis, LNCS 7935. Springer, pp. 25–42.

Apinis, K., Seidl, H., Vojdani, V., 2012. Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis. In: APLAS. LNCS 7705, Springer, pp. 157–172.

Apinis, K., Seidl, H., Vojdani, V., 2013. How to combine widening and narrowing for non-monotonic systems of equations. In: PLDI'13. ACM, pp. 377–386.

Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X., 2003. A static analyzer for large safety-critical software. In: ACM SIGPLAN Notices. Vol. 38. ACM, pp. 196–207.

Bourdoncle, F., 1990. Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity. In: Programming Language Implementation and Logic Programming, 2nd International Workshop PLILP'90. Vol. 456 of Lecture Notes in Computer Science. Springer-Verlag, pp. 307–323.

Bourdoncle, F., 1992. Abstract interpretation by dynamic partitioning. J. Funct. Program. 2 (4), 407–423.

Bourdoncle, F., 1993. Efficient chaotic iteration strategies with widenings. In: In Proceedings of the International Conference on Formal Methods in Programming and their Applications. Springer-Verlag, pp. 128–141.

Carl, S., Heikkilä, S., 2010. Fixed Point Theory in Ordered Sets and Applications. Springer New York, Dordrecht, Heidelberg, London.

Cortesi, A., Zanioli, M., 2011. Widening and narrowing operators for abstract interpretation. Computer Languages, Systems & Structures 37 (1), 24–42.

Costan, A., Gaubert, S., Goubault, E., Martel, M., Putot, S., 2005. A policy iteration algorithm for computing fixed points in static analysis of programs. In: Etessami, K., Rajamani, S. K. (Eds.), Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings. Vol. 3576 of LNCS. Springer, pp. 462–475.

Cousot, P., 1981. Semantic foundations of program analysis. In: Muchnick, S., Jones, N. (Eds.), Program Flow Analysis: Theory and Applications. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, U.S.A., Ch. 10, p. 303—342.

Cousot, P., 2015. Abstracting induction by extrapolation and interpolation. In: D'Souza, D., Lal, A., Larsen, K. G. (Eds.), Verification, Model Checking, and Abstract Interpretation, 16th International Conference (VMCAI). LNCS 8931, Springer, pp. 19–42.

Cousot, P., Cousot, R., 1976. Static determination of dynamic properties of programs. In: Robinet, B. (Ed.), Second International Symposium on Programming, Paris, France. Dunod, Paris, p. 106—130.

Cousot, P., Cousot, R., 1977a. Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4th ACM Symp. on Principles of Programming Languages (POPL'77). ACM Press, pp. 238–252.

Cousot, P., Cousot, R., 1977b. Static Determination of Dynamic Properties of Recursive Procedures. In: IFIP Conf. on Formal Description of Programming Concepts. North-Holland, pp. 237–277.

Cousot, P., Cousot, R., Aug. 1992a. Abstract interpretation frameworks. Journal of Logic and Computation 2 (4), 511–547.

Cousot, P., Cousot, R., 1992b. Comparing the galois connection and widening/-narrowing approaches to abstract interpretation. In: Bruynooghe, M., Wirsing, M. (Eds.), PLILP. Vol. 631 of LNCS. Springer, pp. 269–295.

Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X., 2007. Combination of abstractions in the ASTRÉE static analyzer. In: Okada, M., Satoh, I. (Eds.), Eleventh Annual Asian Computing Science Conference (ASIAN'06). Springer, Berlin, Tokyo, Japan, LNCS 4435, pp. 272–300.

Cousot, P., Halbwachs, N., 1978. Automatic discovery of linear restraints among variables of a program. In: POPL'78. ACM Press, pp. 84–96.

Fecht, C., Seidl, H., 1999. A Faster Solver for General Systems of Equations. Science of Computer Programming 35 (2), 137–161.

Gawlitza, T. M., Seidl, H., Apr. 2011. Solving systems of rational equations through strategy iteration. ACM Trans. Prog. Lang. Syst. 33 (3), 1–48.

Ghorbal, K., Goubault, E., Putot, S., 2009. The zonotope abstract domain taylor1+. In: Bouajjani, A., Maler, O. (Eds.), Computer Aided Verification, 21st International Conference (CAV). Springer, LNCS 5643, pp. 627–633.

Gonnord, L., Halbwachs, N., 2006. Combining widening and acceleration in linear relation analysis. In: Yi, K. (Ed.), Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006. Proceedings. Vol. 4134 of LNCS. Springer, Berlin Heidelberg, pp. 144–160.

Gopan, D., Reps, T., 2006. Lookahead widening. In: Ball, T., Jones, R. (Eds.), Computer Aided Verification. Vol. 4144 of LNCS. Springer, pp. 452–466.

Gopan, D., Reps, T., 2007. Guided static analysis. In: Nielson, H., Filé, G. (Eds.), Proc. of the 14th International Static Analysis Symposium (SAS). Vol. 4634 of LNCS. Springer, pp. 349–365.

Goubault, E., Putot, S., Védrine, F., 2012. Modular static analysis with zonotopes. In: Miné and Schmidt (2012), pp. 24–40.

Gulavani, B., Chakraborty, S., Nori, A., Rajamani, S., 2008. Automatically refining abstract interpretations. In: Ramakrishnan, C., Rehof, J. (Eds.), Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08). Vol. 4963 of LNCS. Springer, pp. 443–458.

Gulwani, S., Jain, S., Koskinen, E., Jun. 2009. Control-flow refinement and progress invariants for bound analysis. In: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI'09). p. 375–385.

Gustafsson, J., Betts, A., Ermedahl, A., Lisper, B., Jul. 2010. The Mälardalen WCET benchmarks – past, present and future. In: Lisper, B. (Ed.), WCET2010. OCG, Brussels, Belgium, pp. 137–147.

Halbwachs, N., Henry, J., 2012. When the decreasing sequence fails. In: Miné and Schmidt (2012), pp. 198–213.

Henry, J., Monniaux, D., Moy, M., 2012a. PAGAI: A path sensitive static analyser. Electronic Notes in Theoretical Computer Science 289, 15–25.

Henry, J., Monniaux, D., Moy, M., 2012b. Succinct representations for abstract interpretation. In: Miné, A., Schmidt, D. (Eds.), Static Analysis Symposium (SAS'12). Vol. 7460 of LNCS. Springer Berlin / Heidelberg, pp. 283–299.

Hofmann, M., Karbyshev, A., Seidl, H., 2010a. Verifying a local generic solver in Coq. In: SAS'10. LNCS 6337, Springer, pp. 340–355.

Hofmann, M., Karbyshev, A., Seidl, H., 2010b. What is a pure functional? In: ICALP (2). LNCS 6199, Springer, pp. 199–210.

Jeannet, B., Miné, A., 2009. Apron: A library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (Eds.), Computer Aided Verification (CAV). Springer, LNCS 5643, pp. 661–667.

Le Charlier, B., Van Hentenryck, P., 1992. A Universal Top-Down Fixpoint Algorithm. Tech. Rep. 92–22, Institute of Computer Science, University of Namur, Belgium.

Miné, A., Schmidt, D. (Eds.), 2012. Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings. Vol. 7460 of LNCS. Springer.

Monniaux, D., Gonnord, L., 2011. Using bounded model checking to focus fixpoint iterations. In: Yahav, E. (Ed.), Static Analysis, 18th International Symposium, SAS 2011. Vol. 6887 of LNCS. Springer, Berlin Heidelberg, pp. 369–385.

Monniaux, D., Guen, J. L., 2012. Stratified static analysis based on variable dependencies. In: The Third International Workshop on Numerical and Symbolic Abstract Domains. Vol. 288 of Electronic Notes in Theoretical Computer Science. p. 61–74.

Necula, G. C., McPeak, S., Rahul, S. P., Weimer, W., 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In: CC'02. Vol. 2304 of LNCS. Springer, pp. 213–228.

Seidl, H., Vene, V., Müller-Olm, M., 2003. Global invariants for analyzing multithreaded applications. Proc. of the Estonian Academy of Sciences: Phys., Math. 52 (4), 413–436.

Sharma, R., Dillig, I., Dillig, T., Aiken, A., 2011. Simplifying loop invariant generation using splitter predicates. In: Gopalakrishnan, G., Qadeer, S. (Eds.), Computer Aided Verification (CAV'11). Vol. 6806 of LNCS. Springer, pp. 703–719.

Simon, A., King, A., 2006. Widening polyhedra with landmarks. In: Kobayashi, N. (Ed.), APLAS. Vol. 4279 of LNCS. Springer, pp. 166–182.

Vergauwen, B., Wauman, J., Lewi, J., 1994. Efficient fixpoint computation. In: SAS'94. Vol. 864 of LNCS. Springer, pp. 314–328.

Vojdani, V., Vene, V., 2009. Goblint: Path-sensitive data race analysis. Annales Univ. Sci. Budapest., Sect. Comp. 30, 141–155.