How to Combine Widening and Narrowing for Non-monotonic Systems of Equations

Kalmer Apinis Helmut Seidl Vesal Vojdani
Technische Universität München
{apinis,seidl,vojdanig}@in.tum.de

Abstract

Non-trivial analysis problems require complete lattices with infinite ascending and descending chains. In order to compute reasonably precise post-fixpoints of the resulting systems of equations, Cousot and Cousot have suggested accelerated fixpoint iteration by means of widening and narrowing [6, 7].

The strict separation into phases, however, may unnecessarily give up precision that cannot be recovered later. While widening is also applicable if equations are non-monotonic, this is no longer the case for narrowing. A narrowing iteration to improve a given post-fixpoint, additionally, must assume that all right-hand sides are monotonic. The latter assumption, though, is not met in presence of widening. It is also not met by equation systems corresponding to context-sensitive interprocedural analysis, possibly combining context-sensitive analysis of local information with flow-insensitive analysis of globals [1].

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

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis; F.2.2 [*Analysis of Algorithms and Problem Complexity*]: Nonnumerical Algorithms and Problems—Computations on discrete structures

Keywords Static Program Analysis, Fixpoint Iteration, Constraint Solving

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 instances of a class. For abstract interpretation, often complete lattices are chosen as domains of (abstract) values [7]. Then variants of Kleene iteration can be applied to determine solutions. Right from the beginning of abstract interpretation, it has been observed that many interesting invariants are best expressed by means of lattices that have *infinite* strictly ascending chains. Possibly infinite strictly ascending chains, though, imply that naive Kleene iteration may not terminate. For that reason, Cousot and Cousot proposed a *widening* iteration, followed by a *narrowing* iteration in order to determine reasonably precise invariants or, technically speaking, reasonably small *post* solutions [6]. The widening phase can be considered as a Kleene iteration that is accelerated by means of a widening operator which enforces that only finitely many increases of values occur for an 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 by means of a *narrowing* operator.

Trying to recover precision once it has been thrown away, though, in general is not possible (see, e.g., [18] for a recent discussion). Some attempts try to improve precision by reducing the number of points where widening applied [4, 10], others rely on refined widening or narrowing operators (see, e.g., [5, 27]). Recent work has focused methods to guide or stratify the exploration of the state space [13–15, 19, 23], including techniques for automatic transformation of irregular loops [16, 26].

Our approach here is complementary to such techniques and can, possibly, be combined with these. Our idea is not to postpone narrowing to a second phase where all losses of information have already occurred and been propagated. Instead, an attempt is made to systematically improve the current information by narrowing immediately. In particular, this means that widening and narrowing are applied in an *interleaved* manner. A similar idea has been proposed for the static analyzer ASTRÉE [2].

Narrowing and the narrowing operator are only guaranteed to return meaningful results, i.e., a post solution of the given system of equations, if the right-hand sides of equations are *monotonic*. This assumption, however, is violated in presence of widening. This requirement is also not met by the equation systems for context-sensitive interprocedural analysis [1, 12]. Moreover, for more complicated abstract domains, these systems may be infinite and thus can be handled by *local* solvers only. Local solvers query the value of an interesting unknown and explore the space of unknowns only insofar as required for answering the query. For this type of algorithm, the set of evaluated unknowns is not known beforehand. In particular the values of new unknowns may be queried in the narrowing phase that have not yet been considered before. As a consequence, the strict separation into a widening and a narrowing phase can no longer be maintained. Although, narrowing iteration

PLDI'13, June 16–19, 2013, Seattle, WA, USA.

in case of non-monotonicity has been considered before [9], such approaches are not directly applicable for local solving.

In order to remedy these obstacles, we introduce an operator $\boxminus$ which is a generic combination of a given widening $\sqcup$ with a given narrowing operator $\sqcap$ and show that this new operator can be plugged into a generic solver of equation systems, be they monotonic or non-monotonic. As a result, solvers are obtained that return reasonably precise post solutions in one go—given that they terminate.

Termination, though, is indeed an issue. We present two simple example systems of monotonic equations where standard fixpoint algorithms such as round robin or work-list iteration, when enhanced with the new operator, fail to terminate. Therefore, we develop a variant of round robin as well as a variant of work-list iteration which in absence of widening and narrowing are not or at least not much worse than their standard counter parts—but which additionally are guaranteed to terminate for monotonic systems when the $\boxminus$-operator is used.

The idea of plugging the new operator $\boxminus$ into a generic *local* solver in principle, i.e., modulo termination, works as well. A local solver such as [21], however, is not generic in the sense of the present paper—meaning that a naive enhancement with the operator $\boxminus$ 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 then extended to a solver for *side-effecting* constraint systems. Such systems allow to conveniently specify analyses that combine context-sensitive analysis of local information with flow-insensitive analysis of globals [1] as provided, e.g., by the program analyzer GOBLINT [29]. Since the different contributions to a global unknown are generated during the evaluation of a subset of right-hand sides, which is not known before-hand and may vary during fixpoint iteration, further non-trivial changes are required to handle this situation.

The remaining paper is organized as follows. In section 2, we present the concept of generic solvers. In section 3, we show that any such solver, when instantiated with $\boxminus$, 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 generic variants of round-robin iteration as well as of work-list based fixpoint computation. Section 5 introduces the new generic local $\boxminus$-solver **SLR**, which then is generalized to equation systems with side effects in section 6. In section 7, we report about preliminary experiments with the novel $\boxminus$-solver within the analyzer framework GOBLINT and conclude in section 8.

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 D of values where the right-hand sides f_x are mappings $(X \to D) \to D$. Furthermore, let $\square : D \to D \to D$ be a binary operator to combine old values with the new contributions of the right-hand sides.

A $\square$-solution of S is a assignment $\rho : X \to D$ such that for all unknowns $x \in X$, $\rho[x] = \rho[x] \square f_x\, \rho$ holds. In the case that $\square$ is defined as $a \square b = b$, a $\square$-solution is an ordinary solution of the system, i.e., a mapping ρ with $\rho[x] = f_x\, \rho$ for all unknowns x. In case D is a lattice, and the $\square$-operator equals the least upper bound operator $\sqcup$, a $\square$-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 $\square$ equals the greatest lower bound operator $\sqcap$, a $\square$-solution is a *pre* solution of the system, i.e., a mapping ρ with $\rho[x] \sqsubseteq f_x\, \rho$ for all unknowns x.

The operator $\square$ can also be instantiated with widening and narrowing operators. According to [6, 7, 11], a widening operator $\sqcup$ for a lattice D must satisfy that $a \sqcup b \sqsupseteq a \sqcup b$ for all $a, b \in D$. This implies that a $\square$-solution then again provides a post solution of the original system S. The situation is slightly more complicated for narrowing operators. For a narrowing operator $\sqcap$, $a \sqsupseteq b$ implies that $a \sqsupseteq (a \sqcap b) \sqsupseteq b$. This means that narrowing can only be applied if the right-hand side of equations are guaranteed to return values that are less or equal than the values of the current left-hand sides. Thus a mapping ρ can only be a $\sqcap$-solution, if it is a post solution of the system.

A (chaotic) solver for systems of equations is an algorithm that maintains a mapping $\rho : X \to D$ and performs a sequence of *update steps*, starting from an initial mapping ρ_0. 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.,

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

Then the algorithm is a $\square$-solver if upon termination the final mapping (after completing n steps) ρ_n is a $\square$-solution of S. The algorithm is a generic solver, if it works for any binary update operator $\square$. In this sense, the round-robin iteration of Fig. 1 is a generic solver. 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 generic solver, i.e., upon termination returns a $\square$-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, $\rho_i[x] \neq \rho_i(x) \square f_x\, \rho_i$ implies that for some $j \geq i$, an update $\rho_{j+1}[x] \leftarrow \rho_j[x] \square f_x\, \rho_j$ occurs.

Not every solver algorithm, though, may consider right-hand sides of equations as black boxes, as the round-robin algorithm does. The worklist algorithm from Fig. 2 can only be used as generic solver—given that all dependences are provided before-hand. This means that for each right-hand side f_x a (super-)set dep_x of unknowns is 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) 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 re-computed. Note that whenever an update to a unknown y provides a new value, we re-schedule y for evaluation as well. This is a precaution for the case that the operator $\square$ is *not* (right) idempotent. Here, an operator $\square$ is called *idempotent* if the following equality:

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

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

3. Enhancing Narrowing

First, we observe:

FACT 1. *Assume that all right-hand sides of the system S of equations over a lattice D are monotonic and that ρ_0 is a post solution of S, and $\square$ is a narrowing operator $\sqcap$. Then the sequence $\rho_0, \rho_1, \ldots$ of mappings produced by a generic $\square$-solver, is defined and decreasing.* ■

```
do {
    dirty ← false;
    forall (x ∈ X) {
        new ← ρ[x] □ f_x ρ;
        if (ρ[x] ≠ new) {
            ρ[x] ← new;
            dirty ← true;
        }
    }
} while (dirty);
```

Figure 1. The solver **RR**.

```
W ← X;
while (W ≠ ∅){
    x ← extract(W);
    new ← ρ[x] □ f_x ρ;
    if (ρ[x] ≠ new) {
        ρ[x] ← new;
        W ← W ∪ infl_x;
    }
}
```

Figure 2. The Solver **W**.

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

Equation systems for context-sensitive interprocedural analysis, though, are not necessarily monotonic. In the following we show how to lift the the technical restrictions to the applicability of narrowing. Given a widening operator $\sqcup$ and a narrowing operator $\sqcap$, we define a new binary operator $\boxminus$ by:

$$a \boxminus b = \begin{cases} a \sqcap b, & \text{if } b \sqsubseteq a \\ a \sqcup b, & \text{otherwise.} \end{cases}$$

Note that the operator $\boxminus$ is not necessarily idempotent, but whenever narrowing is idempotent the following holds:

$$(a \boxminus b) \boxminus b = (a \boxminus b) \sqcap b$$

and therefore also

$$((a \boxminus b) \boxminus b) \boxminus b = (a \boxminus b) \boxminus b\,.$$

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

LEMMA 1. *Consider a finite system S of equations over a lattice D. Then every $\boxminus$-solution ρ of S is a post solution, i.e., for all unknowns x, $\rho[x] \sqsupseteq f_x\,\rho$.*

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

$$\rho[x] \;=\; \rho[x] \boxminus f_x\,\rho \;=\; \rho[x] \sqcup f_x\,\rho \;\sqsupseteq\; f_x\,\rho$$

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

Thus, every generic solver for lattices 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 $\boxminus$, has the additional advantage that values may also *shrink* in-between. Improving possibly too large values, thus, may take place immediately resulting in overall smaller, i.e., better post solutions. Moreover, no restriction is imposed any longer concerning monotonicity of right-hand sides.

4. Enforcing termination

For the new operator $\boxminus$, termination cannot generally be guaranteed for all solvers. In this section, we therefore present a modification of worklist iteration which is guaranteed to terminate—given that all right-hand sides of equations are monotonic.

EXAMPLE 1. *Consider the system:*

$$\begin{array}{lcl} x_1 & = & x_2 \\ x_2 & = & x_3 + 1 \\ x_3 & = & x_1 \end{array}$$

with $D = \mathbb{N} \cup \{\infty\}$, the lattice of non-negative integers, equipped with the natural ordering $\sqsubseteq$ given by $\leq$ and extended with ∞. Consider a widening $\sqcup$ where $a \sqcup b = a$ if $a = b$ and $a \sqcup b = \infty$ otherwise, together with a narrowing $\sqcap$ where for $a \geq b$, and $a \sqcap b = b$ if $a = \infty$, and $a \sqcap b = a$ otherwise. Round-robin iteration with the operator $\boxminus$ 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	...

thus does no not terminate—although right-hand sides are monotonic. ■

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

EXAMPLE 2. *Consider the two equations:*

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

using the same lattice as in example 1 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 as well as of the worklist solver for which termination can be guaranteed. The worst case complexity for the new round-robin solver turns out to be even by a factor of 2 faster than ordinary round-robin iteration. For the new worklist solver, the theoretical complexity is at least not far away from the classical iterator.

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 [3], has a significant impact on performance. Hence, the linear ordering should be chosen in a way that innermost loops would be evaluated before iteration on outer loops. For unknowns

x_i and the system of equations given by $x_i = f_i$, for $i = 1, \ldots, n$, the new round-robin algorithm is shown in Fig. 3.

Let us call the new algorithm **SRR** (*structured* round-robin). 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. Clearly, the resulting algorithm is a generic $\Box$-solver.

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

THEOREM 1. *Consider the algorithm* **SRR** *for a system of n equations over a complete lattice where all right-hand sides are monotonic and $\Box = \boxminus$. Then the following holds:*

1. *Assume that the lattice has bounded height h and $\uplus = \sqcup$. Then* **SRR** *when started with the initial mapping $\rho_0 = \{x_i \mapsto \bot \mid i = 1, \ldots, n\}$, terminates after at most $n + \frac{h}{2}n(n+1)$ evaluations of right-hand sides f_i.*
2. *Also in presence of unbounded ascending chains, the algorithm* **SRR** *will terminate for* every *initial mapping.*

Proof. Recall that ordinary round robin iteration 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 overhead

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

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

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

giving us statement 1. For the second statement, we proceed by induction on i. The case $i = 0$ is vacuously true. For the induction step assume $i > 0$. For a contradiction assume that solve i for the current mapping does not terminate. First assume that $f_i\ \rho$ returns a value smaller than $\rho[x_i]$ while for all $j < i$, $\rho[x_j] = \rho[x_j] \boxminus f_j\ \rho$ implying that $\rho[x_j] \sqsupseteq f_j\ \rho$ for all $j < i$. Then due to monotonicity, the subsequent iteration of solve i will produce a decreasing sequence of mappings implying that the operator $\boxminus$ during all occurring updates behaves like $\sqcap$. Since all decreasing chains produced by narrowing are ultimately stable, the call solve i will terminate—in contradiction to our assumption.

Therefore during the whole run of solve i the mapping ρ' when evaluating f_i, must always return a value that is not subsumed by $\rho[x_i]$. Since all calls solve $(i - 1)$ inbetween terminate by 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—in contradiction to our assumption. We conclude that solve i is eventually terminating. ■

EXAMPLE 3. *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:*

```
void solve i {
    if (i = 0) return;
    solve (i − 1);
    new ← ρ[x_i] □ f_i ρ;
    if (ρ[x_i] ≠ new) {
        ρ[x_i] ← new;
        solve i;
    }
}
```

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

```
Q ← ∅;
for (i ← 1; i ≤ n; i++) add Q x_i;
while (Q ≠ ∅) {
    x_i ← extract_min (Q);
    new ← ρ[x_i] □ f_i ρ;
    if (ρ[x_i] ≠ new) {
        ρ[x_i] ← new;
        add Q x_i;
        forall (x_j ∈ infl_i) add Q x_j;
    }
}
```

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

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 fix-point solving 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 we are given for each right-hand side f_i a (super-)set 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. As before for each unknown x_j, let infl_j denote the 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 resulting algorithm is a generic solver.

EXAMPLE 4. *Consider again the system from example 2. Structured worklist iteration using $\boxminus$ 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 2. *Assume the algorithm* **SW** *is applied to a system of equations over a complete lattice $\mathbb{D}$ and that each right-hand side is monotonic.*

1. *Assume that the maximal length of a strictly ascending chain is bounded by h. When instantiated with $\square = \sqcup$, and started with an initial mapping ρ mapping each unknown to $\bot$, the algorithm terminates after at most $h \cdot N$ evaluations of right-hand sides where $N = \sum_{i=1}^{n}(2 + |\mathsf{dep}_i|)$.*
2. *When instantiated with $\square = \boxminus$ and started on any mapping, the algorithm is guaranteed to terminate and, thus, always to return a post solution.*

The first statement of the theorem indicates that **SW** behaves complexity-wise like ordinary worklist iteration: 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 the operator $\boxminus$.

Proof. 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 $d_{i+1} = d_i \boxminus f_n\,\rho_i$ which is less or equal to ρ_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 therefore 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, $d_{i+1} = d_i \sqcup f_n\,\rho_i$ 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. ■

Since the algorithm **SW** is a generic solver, it can also be applied to non-monotonic systems. There, however, termination can no longer be guaranteed. One generic idea, though, to enforce termination for *all* $\boxminus$-solvers and in all cases, is to equip each unknown with a separate counter that counts how often the solver has switched from narrowing back to widening. That number then may be taken into account by the $\boxminus$-operator, e.g., by choosing successively less aggressive narrowing operators $\boxminus_0, \boxminus_1, \ldots$, and, ultimately, to give up improving the obtained values. The latter is achieved by defining $a \boxminus_k b = a$ for a certain threshold k.

5. Local generic solvers

Similar to generic solvers, we define generic *local* solvers. Use of local solvers can 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 [1, 8]. 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 [12, 22, 28]. For that, it seems convenient that the *dynamic* dependences between unknowns are approximated. 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 [20].

Essentially, purity for a right-hand side f means that evaluating $f\,\rho$ for a mapping ρ operationally consists of a finite sequence value lookups in ρ where the next unknown whose value 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.

A *partial* $\square$-solution of an (infinite) system of pure equations S is a set $\mathsf{dom} \subseteq X$ and a mapping $\rho : \mathsf{dom} \to D$ with the following two properties:

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

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

EXAMPLE 5. *The following equation system (for $n \in \mathbb{N} = D$)*

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

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* $\rho = \{y_1 \mapsto 2, y_2 \mapsto 2, y_4 \mapsto 2\}$. ■

A *local* generic solver instantiated with an operator $\square$, then, is an algorithm that, when given a system of pure equations S, and 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 $\square$-solution (dom, ρ), such that $x_0 \in \mathsf{dom}$.

At first sight, it may seem surprising that such local generic 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.

One more elaborate algorithm for local solving is formalized by Hofmann et al. [21], 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. However, when enhanced with an operator $\square$, this algorithm is *not* a generic 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 $\square$-solution—even if it terminates. Here, we therefore, provide a variant of **RLD** where right-hand sides (conceptually) are executed atomically.

Clearly, a local generic solver does not terminate if infinitely many unknowns are encountered. Therefore, a reasonable local

```
let rec solve x =
  if x ∉ stable then
    stable ← stable ∪ {x};
    tmp ← σ[x] ⊔ f_x(eval x);
    if tmp ≠ σ[x] then
      W ← infl[x];
      σ[x] ← tmp;
      infl[x] ← [];
      stable ← stable \ W;
      foreach x ∈ W do solve x
    end
  end

and eval x y =
  solve y ;
  infl[y] ← infl[y] ∪ {x};
  σ[y]

in
  stable ← ∅; infl ← ∅; σ ← ∅;
  solve x_0; σ
```

Figure 5. The solver **RLD** from [21].

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 solver is guaranteed to terminate for the operator $\boxdot$ at least for 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, **SLR** (*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. Moreover, a global assignment $\mathsf{infl} : \mathsf{dom} \to 2^X$ records for each unknown in $y \in \mathsf{dom}$, the unknown y itself together with 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.

Unknowns whose equations may possibly be no longer valid, will be scheduled for reevaluation. This means that they are inserted into the global priority queue Q. Finally, there is a global mapping $\rho : \mathsf{dom} \to \mathbb{D}$ that records the current values for the encountered unknowns.

As in the algorithm **RLD**, right-hand sides f_x are not directly evaluated for the current mapping ρ, but instead for a 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. Also, if the encountered unknown is new, then it tries not just to return the value $\bot$, but to compute the best possible value for the new unknown before-hand. 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. The function eval 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. Then eval

```
let rec solve x =
  if x ∉ stable then
    stable ← stable ∪ {x}
    tmp ← ρ[x] □ f_x(eval x);
    if tmp ≠ ρ[x] then
      W ← infl[x];
      foreach y ∈ W do add Q y;
      ρ[x] ← tmp;
      infl[x] ← {x};
      stable ← stable \ W;
      while (Q≠∅)∧(min_key Q ≤ key[x]) do
        solve (extract_min Q);
    end
  end

and init y =
  dom ← dom ∪ {y};
  key[y] ← −count; count++;
  infl[y] ← {y}; ρ[y] ← ρ_0[y]

and eval x y =
  if y ∉ dom then
    init y; solve y
  end;
  infl[y] ← infl[y] ∪ {x};
  ρ[y]

in
  stable ← ∅; infl ← ∅; ρ ← ∅;
  dom ← ∅;
  Q ← empty_queue(); count ← 0;
  init x_0; solve x_0; ρ
```

Figure 6. The new solver **SLR**.

records the fact that x depends on y, by adding x to the set $\mathsf{infl}[y]$. Only then is the corresponding value $\rho[y]$ returned.

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 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 $\rho[x'] \neq \rho[x'] \sqcap f_{x'}\,\rho$ holds. In the procedure solve for x, the call min_key Q 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 a set stable of all unknowns introduced so far such that for all unknowns x' in stable one of the following properties hold at each call of procedure solve:

- a call to the procedure solve x' has been started and the update of $\rho[x']$ has not yet occurred; or
- the equality $\rho[x'] = \rho[x'] \sqcap f_{x'}\,\rho$ 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 $\mathsf{key}[x]$. Since x_0 has received the largest key, the initial call $\mathsf{solve}\, x_0$ will result, upon termination, in an empty priority queue Q.

EXAMPLE 6. *Consider again the infinite equation system from example 5. The solver* **SLR***, 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 **SLR** 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 3. *1. When applied to any system of pure equations and interesting unknown x_0, the algorithm* **SLR** *returns a partial $\Box$-solution—whenever it terminates.*

2. *Assume that* **SLR** *is applied to a system of pure equations over a complete lattice $\mathbb{D}$ where each right-hand side is monotonic. If the operator $\Box$ is instantiated with $\boxminus$, then for any initial mapping ρ_0 and interesting unknown x_0,* **SLR** *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 $\mathsf{solve}\, y$ will never modify the value $\rho[x]$ of an unknown x with $\mathsf{key}[x] > \mathsf{key}[y]$. 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 $f_x\,(\mathsf{eval}\, x)$, will return the same value — but by a computation that does not change ρ and thus is atomic.

In order to prove that **SLR** 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 **SLR** 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 $\boxminus$ behaves like $\sqcup$. As soon as the same value $\rho[x]$ or a smaller value is obtained, the operator $\boxminus$ behaves like the operator $\sqcap$. 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 satisfying 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 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 $\sqcup$ 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 $\sqcap$. Due to the properties of $\sqcup$ and $\sqcap$, we conclude that the iteration must terminate. ■

6. Side-effecting systems of equations

Generic solving, as we have discussed in the preceding sections cannot generally be 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, which recasts *assert*-statements of PROLOG or DATALOG programs, has been advocated in [1] for an elegant treatment of interprocedural analysis using partial contexts and flow-insensitive unknowns and thus also of multi-threaded programs [25].

EXAMPLE 7. *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 asignments, 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 a *lattice* $\mathbb{D}$ and also that right-hand sides are *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 $\mathsf{side} : X \to \mathbb{D} \to \mathbf{unit}$, 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 also to each unknown $z \neq x$ at most one side effect. 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 a side-effecting variant $\mathbf{SLR}^+$ of the algorithm **SLR** from section 5 that for such systems returns a partial $\Box$-solution—whenever it terminates. Moreover, the enhanced solver $\mathbf{SLR}^+$ is guaranteed to terminate whenever all right-hand sides f_x are *monotonic*, i.e., the functions $\bar{f}_x$ all are monotonic.

EXAMPLE 8. *Consider again the analysis of example 7. 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* **SLR** *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 $\mathbf{SLR}^+$ therefore creates for each side effect to unknown z inside the right-hand side of x, a fresh unknown $\langle x, z\rangle$ 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, the side effect $\mathsf{side}\,z\,d$ is implicitly replaced with
$$\mathsf{side}\,\langle x, z\rangle\,d$$
while additionally, 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 $\langle z, x\rangle$, $z \in \mathsf{set}[x]$.

The $\boxminus$-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 **SLR**.

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 side is required for realizing the side-effects during an evaluation of a right-hand side. As the function eval, also function side receives the left-hand side of the equation under consideration as its first argument. We define:

```
side x y d = if ⟨x,y⟩ ∉ dom then
               ρ[⟨x,y⟩] ← ⊥;
             if d ≠ ρ[⟨x,y⟩] then
               ρ[⟨x,y⟩] ← d;
               if y ∈ dom then
                 set[y] ← set[y] ∪ {x};
                 stable ← stable \ {y};
                 add Q y
               else
                 init y;
                 set[y] ← {x};
                 solve y
               end
             end
```

When called with x, y, d, the function side first initializes the unknown $\langle x, y\rangle$ if it is not yet contained in dom. If the new value is different from the old value of ρ for $\langle x, y\rangle$, $\rho[\langle x, y\rangle]$ 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 $\mathsf{side}\,x$ as a second argument and additionally evaluates all unknowns collected in $\mathsf{set}[x]$. The corresponding new line reads:

$$\mathsf{tmp} \leftarrow \rho(x) \boxminus (f_x\,(\mathsf{eval}\,x)\,(\mathsf{side}\,x) \sqcup \bigsqcup\{\rho\,\langle z, x\rangle \mid z \in \mathsf{set}\,x\});$$

EXAMPLE 9. *Consider again interval analysis for the program from example 7. 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*

$$[0, 0] \boxminus [0, 3] = [0, 0] \sqcup [0, 3] = [0, \infty]$$

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

$$[0, \infty] \boxminus [0, 3] = [0, \infty] \sqcap [0, 3] = [0, 3]$$

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

Analogously to theorem 3 from the last section, we obtain:

THEOREM 4. *1. When applied to any system of pure equations with side effects and interesting unknown x_0, the algorithm $\mathbf{SLR}^+$ returns a partial post solution—whenever it terminates.*
2. Assume that $\mathbf{SLR}^+$ is applied to a system of pure equations over a complete lattice $\mathbb{D}$ where each right-hand side is monotonic. Then for any initial mapping ρ_0 and interesting unknown x_0, $\mathbf{SLR}^+$ is guaranteed to terminate and thus always to return a partial post solution—whenever only finitely many unknowns are encountered.

7. Experimental evaluation

We have implemented the generic local solver $\mathbf{SLR}^+$ and included into the analyzer GOBLINT for multi-threaded C programs. Goblint uses CIL as C front-end [24] and is written in OCAML. The tests were performed on 2.7GHz Intel Core i7 laptop, with 8GB DDR3 RAM, running OS X 10.8.2.

In order to compare the precision attained by two-phase solving according to [6] using widening and narrowing, and $\boxminus$-solving, we used the benchmark suite[1] from the Märdalen WCET research group [17] which collects a series of interesting small examples for WCET analysis, varying in size from about 40 lines to 4000 lines of code. On top of standard analyses of pointers, we performed an interval analysis where local variables are analyzed context-insensitively while global variables are treated flow-insensitively. Within this setting, we determined the precision achieved by the $\boxminus$-solver compared to a corresponding solver which realizes a distinct widening phase, followed by a distinct narrowing phase. The results of this comparison is displayed in fig. 7. Since the absolute runtimes are negligible (about 14 seconds for all programs together), we only display the relative precision. For that, we list, sorted

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

Program	⊔ without context		⊟ without context		⊔ with context		⊟ with context	
	Time(s)	Unknowns	Time(s)	Unknowns	Time(s)	Unknowns	Time(s)	Unknowns
401.bzip2	3.3	6 565	3.3	6 578	3.3	7 033	3.3	7 028
429.mcf	0.4	1 245	0.4	1 245	0.5	1 673	0.5	1 673
433.milc	9.3	8 206	9.3	8 206	13.5	36 720	13.4	36 711
456.hmmer	11.2	11 446	11.2	11 446	12.6	80 938	13.3	81 038
458.sjeng	49.1	13 306	52.1	13 306	121.0	97 785	131.0	108 371
470.lbm	0.3	784	0.3	784	0.4	1 577	0.2	899
482.sphinx	14.0	11 963	13.9	11 963	6.3	21 610	6.3	21 610

Table 1. Results for programs from the SpecCpu2006 benchmark suite.

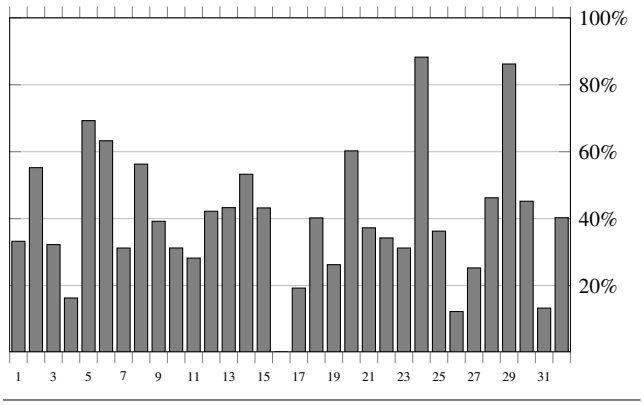

Figure 7. The percentage of program points with improvement.

by program size, the percentage of program points where the ⊟-solver returns better results. With the notable exception of the benchmark `qsort-exam.c` without improvement (number 16 in the list), all other benchmarks show significant improvements, with an weighted average improvement of 39%.

In a second experiment, we explored the efficiency of our implementation of a generic local ⊟-solver. Our collection of benchmarks consists of all C projects of the SpecCpu2006 benchmark suite that can be handled by CIL. The programs `400.perlbench` and `445.gobmk` could not be analyzed in this setting, as there was (to our best understanding) insufficient memory to do so. For the other programs, we performed interval analysis in two variants. The first variant (reported in the table 1 to the left) performs a context-insensitive analysis, while the second variant (reported to the right) analyzes local variables depending on a calling context which includes all non-interval values of locals. While the first variant of analysis can be realized by means of clearly separated phases of widening and narrowing, the second variant cannot, 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. In order to explore the extra costs incurred by the operator ⊟ for each variant, we then compared the run-times if just widening is applied or ⊟. While in size, the programs range from 1 to 33 kloc, the effective run-times essentially depend on the number of unknowns to be handled by the solvers.

Not surprisingly, the run-times for context-*insensitive* analysis, (as reported to the left) are faster than those if context is taken into account, with the exception of `482.sphinx`. As we can see here, sometimes the dependency-breaking property of context-sensitive analysis succeeds in performing the analysis faster—even when the context-sensitive case needs substantially more unknowns.

In absence of context, the ⊟-solver is only marginally slower than the corresponding ⊔-solver. The efficiency picture for context-*sensitive* analysis, however, is less clear. Here, it seems that the efficiency crucially depends on the number of needed unknowns.

For some programs the numbers of encountered unknowns do not depend greatly on the chosen solver, since the sets of occurring contexts are largely independent of the computed intervals. In these cases, the solvers essentially perform equally well. For the remaining benchmarks, the number of contexts may increase (456 and 458) or decrease (470) with ⊟, implying that the run-time of the ⊟-solver may increase or decrease compared to the ⊔-solver.

In summary, the ⊟-solver $\mathbf{SLR}^+$ turns out to be a robust algorithm with decent run-times. The new solver allows to significantly improve precision over the two-phase widening/narrowing approach and also is successfully applicable in more general analysis scenarios, where the two-phase approach was not applicable.

8. Conclusion

We have presented a generic combination of widening and narrowing into a single operator ⊟ and systematically explored solver algorithms which, when instantiated with ⊟ will solve general systems of equations. Perhaps surprisingly, standard versions of fixpoint algorithms, when enhanced with ⊟, 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.

Furthermore, we provided preliminary practical evidence that fixpoint iteration based on the combined operator ⊟ may increase precision significantly, and that it also behaves well for interprocedural analysis where right-hand sides of equations can no longer be considered as monotonic. Our experiments were performed independent of other methods of increasing the precision of the widening/narrowing approach. It remains for future work to explore how well different methods to increase precision may cooperate for computing precise analysis results at a decent cost.

Acknowledgments

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement n° 269335 and from the German Science Foundation (DFG). The last author is partially supported by the Estonian Science Foundation (EstSF) under grant n° 8421.

References

1. K. Apinis, H. Seidl, and V. Vojdani. Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis. In *APLAS*, pages 157–172. LNCS 7705, Springer, 2012.
2. Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *ACM SIGPLAN Notices*, volume 38, pages 196–207. ACM, 2003.
3. François Bourdoncle. 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*, volume 456 of *Lecture Notes in Computer Science*, pages 307–323. Springer-Verlag, 1990.
4. François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *In Proceedings of the International Conference on Formal Methods in Programming and their Applications*, pages 128–141. Springer-Verlag, 1993.
5. Agostino Cortesi and Matteo Zanioli. Widening and narrowing operators for abstract interpretation. *Computer Languages, Systems & Structures*, 37(1):24–42, 2011.
6. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In B. Robinet, editor, *Second International Symposium on Programming, Paris, France*, page 106—130. Dunod, Paris, 1976.
7. P. Cousot and R. Cousot. 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)*, pages 238–252. ACM Press, 1977.
8. P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Recursive Procedures. In *IFIP Conf. on Formal Description of Programming Concepts*, pages 237–277. North-Holland, 1977.
9. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the ASTRÉE static analyzer. In M. Okada and I. Satoh, editors, *Eleventh Annual Asian Computing Science Conference (ASIAN'06)*, pages 272–300, Tokyo, Japan, LNCS 4435, 2007. Springer, Berlin.
10. Patrick Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, page 303—342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, U.S.A., 1981.
11. Patrick Cousot and Radhia Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In Maurice Bruynooghe and Martin Wirsing, editors, *PLILP*, volume 631 of *LNCS*, pages 269–295. Springer, 1992.
12. Christian Fecht and Helmut Seidl. A Faster Solver for General Systems of Equations. *Science of Computer Programming*, 35(2):137–161, 1999.
13. Denis Gopan and Thomas Reps. Lookahead widening. In Thomas Ball and Robert Jones, editors, *Computer Aided Verification*, volume 4144 of *LNCS*, pages 452–466. Springer, 2006.
14. Denis Gopan and Thomas Reps. Guided static analysis. In Hanne Nielson and Gilberto Filé, editors, *Proc. of the 14th International Static Analysis Symposium (SAS)*, volume 4634 of *LNCS*, pages 349–365. Springer, 2007.
15. Bhargav Gulavani, Supratik Chakraborty, Aditya Nori, and Sriram Rajamani. Automatically refining abstract interpretations. In C. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *LNCS*, pages 443–458. Springer, 2008.
16. Sumit Gulwani, Sagar Jain, and Eric Koskinen. 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)*, page 375–385, June 2009.
17. Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. In Björn Lisper, editor, *WCET2010*, pages 137–147, Brussels, Belgium, July 2010. OCG.
18. Nicolas Halbwachs and Julien Henry. When the decreasing sequence fails. In Antoine Miné and David Schmidt, editors, *SAS*, volume 7460 of *LNCS*, pages 198–213. Springer, 2012. ISBN 978-3-642-33124-4.
19. Julien Henry, David Monniaux, and Matthieu Moy. Succinct representations for abstract interpretation. In Antoine Miné and David Schmidt, editors, *Static Analysis Symposium (SAS'12)*, volume 7460 of *LNCS*, pages 283–299. Springer Berlin / Heidelberg, 2012.
20. Martin Hofmann, Aleksandr Karbyshev, and Helmut Seidl. What is a pure functional? In *ICALP (2)*, pages 199–210. LNCS 6199, Springer, 2010.
21. Martin Hofmann, Aleksandr Karbyshev, and Helmut Seidl. Verifying a local generic solver in Coq. In *SAS'10*, pages 340–355. LNCS 6337, Springer, 2010.
22. B. Le Charlier and P. Van Hentenryck. A Universal Top-Down Fixpoint Algorithm. Technical Report 92–22, Institute of Computer Science, University of Namur, Belgium, 1992.
23. David Monniaux and Julien Le Guen. Stratified static analysis based on variable dependencies. In *The Third International Workshop on Numerical and Symbolic Abstract Domains*, 2011.
24. George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC'02*, volume 2304 of *LNCS*, pages 213–228. Springer, 2002.
25. Helmut Seidl, Varmo Vene, and Markus Müller-Olm. Global invariants for analyzing multithreaded applications. *Proc. of the Estonian Academy of Sciences: Phys., Math.*, 52(4):413–436, 2003.
26. Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. Simplifying loop invariant generation using splitter predicates. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 703–719. Springer, 2011.
27. Axel Simon and Andy King. Widening polyhedra with landmarks. In Naoki Kobayashi, editor, *APLAS*, volume 4279 of *LNCS*, pages 166–182. Springer, 2006. ISBN 3-540-48937-1.
28. B. Vergauwen, J. Wauman, and J. Lewi. Efficient fixpoint computation. In *SAS'94*, volume 864 of *LNCS*, pages 314–328. Springer, 1994.
29. Vesal Vojdani and Varmo Vene. Goblint: Path-sensitive data race analysis. *Annales Univ. Sci. Budapest., Sect. Comp.*, 30:141–155, 2009.