Enhancing Top-down Solving with Widening and Narrowing

Kalmer Apinis[1], Helmut Seidl[2], and Vesal Vojdani[1]

[1] Deptartment of Computer Science, University of Tartu,
J. Liivi 2, EE-50409 Tartu, Estonia
{kalmer.apinis,vesal.vojdani}@ut.ee
[2] Lehrstuhl für Informatik II, Technische Universität München,
Boltzmannstraße 3, D-85748 Garching b. München, Germany
seidl@in.tum.de

Abstract. We present an enhancement of the generic fixpoint algorithm **TD** which can deal with widening and narrowing even for non-monotonic systems of equations. In constrast to corresponding enhancements proposed for other standard fixpoint algorithms, no extra priorities on variables are required. Still, a mechanism can be devised so that occurrences of the widening/narrowing operator are inserted as well as removed dynamically.

1 Introduction

Many analysis problems can be formalized as (post)-solutions to systems of equations $x = f_x, x \in V$, where V is a set of unknowns each denoting and f_x is a function specifying how x depends on the values of other unknowns in V. In the simplest setting of context-insensitive analysis of sequential imperative programs, the set of unknowns is given by the set of program points, for which the equation system provides a specification of valid invariants. In the more elaborate case of context-sensitive analysis, though, the unknowns are no longer plain program points but also incorporate information about the calling contexts of the respective functions.

A *solver* of systems of equations is an algorithm which determines such post-solutions. It is *local* if it is started with a query to the value of some unknown and then tries to explore the system only as much as is necessary to determine the answer to the query. Local solving has attracted attention in particular for inter-procedural analysis of recursive programs [3, 9, 15, 21, 23, 24] where the potential number of abstract calling contexts can be extremely high, if not *infinite*, while the number of those contexts which are really required for describing all occurring contexts for each function may be quite small. Local solvers are also the method of choice to realize *incremental* program analysis, e.g., when updating the analysis result after an incremental modification of the source program in question [14, 22]. Particularly useful, from a software engineering perspective, are *generic* local solvers [12, 13, 16, 19, 21] which make only minimal assumptions about the domain

of values to compute with, e.g., that it has an ordering relation $\sqsubseteq$ as well as an upper bound operation $\sqcup$, while right-hand sides of equations are taken as black boxes. This means that the right-hand side f_x of a variable x is *not* considered as some kind of syntactical expression which can be inspected beforehand and must be interpreted by the solver. Instead, the right-hand side is considered as a function implemented in some programming language which can be called for another function σ representing the current information about the unknowns and returns a contribution to the left-hand side x of the equation. Interestingly, quite competitive generic solvers have been proposed. Among these, the *top-down* solver **TD** [19] or the solver from GOBLINT [16]. Since they are completely ignorant of the concrete analysis problem in question, they conveniently allow the analysis designer to separate the algorithmic concerns of solving from the design of a suitable domain of abstract values (corresponding to potential invariants) and the realization of the abstract semantics by means of equations. Accordingly, they are at the heart of modern analysis frameworks such as the CIAO system [15] or GOBLINT [4, 24].

One key problem for solving systems of equations is that many interesting analysis problems require lattices with *infinite* strictly ascending chains. This is already the case for interval analysis [7] which tries to determine for each numerical program variable and program point a suitably small interval superset of all runtime values, but also the case for more elaborate numerical properties such as octagons [20] or polyhedra [11]. One general technique to deal with such problems is the widening and narrowing approach as proposed by Cousot and Cousot [7]. The idea is to accelerate the Kleene-type fixpoint iteration for the system of equations by rapidly increasing the values of the unknowns through a *widening* operator. In this way, a guarantee of termination is traded against a severe loss in precision—some of which may later be recovered by means of a subsequent *narrowing* iteration. Technically, a widening produces a larger value than the ordinary least upper bound operator. Thus it may reach a post-solution more quickly (and hopefully in finitely many steps), whereas narrowing when applied to a post-solution (perhaps produced by over-enthusiastic widening) may return a better post-solution.

While local solvers such as the *top-down* solver **TD** or the solver from GOBLINT can easily be extended to work with widening, it has been observed in [5], that they do not go well with narrowing. There are two reasons for this behavior. First, during the narrowing phase, further unknowns may be encountered which may not yet been considered so far. More severely, however, is that the application of context-sensitive analysis may result in *non-monotonic* systems of equations, while narrowing in the original sense can only be applied when all right-hand sides are monotonic. As a remedy, therefore, Apinis et al. introduce a new operator $\boxtriangledown$ which combines a widening operator ∇ with a narrowing operator Δ into one [5]. By means of this operator, variants of several standard solvers are derived and sufficient conditions are provided for which these algorithms are guaranteed to terminate. In particular, variants of the generic local solvers underlying the GOBLINT system are presented. The key idea for enforcing termination is to

maintain an ordering on the names of the unknowns which is respected during fixpoint iteration.

In this paper, we present a variant of the solver **TD** which also supports widening and narrowing. It turns out that **TD** iterates according to an ordering on unknowns as provided by the iteration strategy. This means that it suffices to insert the operator $\boxtriangledown$ into the base algorithm for combining old values with newly computed values and additionally always trigger reevaluation of an unknown, whenever its value has changed. Already for this minimalistic enhancement, termination can be guaranteed—at least for equation systems where all right-hand sides are monotonic, and only finitely many unknowns are encountered. Beyond that, we enhance the algorithm so that the operator $\boxtriangledown$ is not applied for each equation when its right-hand side is evaluated, but only for a small subset of these. This subset is dynamically established by means of the set of unknowns under evaluation by the solver, which is explicitly maintained by **TD** anyway.

2 Equation System

Assume that D is a set of values. Usually, we assume that D is a complete lattice, but weaker assumptions would do as well. The *minimal* requirements are that D is equipped with a partial ordering $\sqsubseteq$, that there is a designated least element $\bot$ with $\bot \sqsubseteq d$ for all $d \in D$, and that there is a binary upper bound operation $\sqcup$, i.e., $a \sqsubseteq a \sqcup b$ and $b \sqsubseteq a \sqcup b$. Let V denote a set of variables or unknowns. Then a system $\mathcal{C}$ of equations over the values D with variables from V is a collection of equations:

$$x = f_x\,, \quad x \in V$$

where the right-hand side f_x of an unknown x specifies how the value of x depends on the values of all other unknowns in the system. Thus, f_x can be understood as a function $f_x : (V \to D) \to D$, which for every assignment $\sigma : V \to D$ of values to unknowns, returns a value in D for the left-hand side x. A mapping $\sigma : V \to D$ is a *post*-solution to S if the values of σ for the left-hand sides are upper bounds to the values returned by their respective right-hand sides for σ, i.e., if $\sigma\, x \sqsupseteq f_x\, \sigma$ for all $x \in V$.

Example 1. As a running example, consider the following equation system with two equations

$$x = (x < 2^{32}\ ?\ y\ :\ 2^{32})$$
$$y = x + 1$$

where the set D of values is given by $D = \mathbb{N} \cup \{\infty\}$, equipped with the natural ordering and extended with ∞. The right-hand side of x returns the value of y, if x is less than 2^{32}, otherwise it returns 2^{32}. The right-hand side of y, however, always returns the value of $x + 1$. Then the mappings $\{x \mapsto 2^{32}, y \mapsto 2^{32} + 1\}$ and $\{x \mapsto \infty, y \mapsto \infty\}$ are post-solutions for the given equation system. ∎

In static program analysis, equation systems are used for specifying data flow [18] or the abstract semantics of a programs [3, 11]. The value domains in such

cases typically are (complete) lattices where the right-hand side functions are monotonic.

In the practical application within a program analyzer, the function f_x is not given as a mathematical object, but as a piece of code realizing the mathematical function. This code can be implemented in any language. We only make the assumption that the realized function is terminating and *pure* in the sense of [17]. This means that operationally, every evaluation of $f_x\,\sigma$ consists of a finite sequence of *steps* which eventually returns a value, where each step consists of a look-up of the value of an unknown, followed by some computation solely depending on the sequence of values read so far.

For some analysis problems, the set of potential unknowns and thus the resulting systems of equations may be very large, or even *infinite*. In order to deal with such a situation, the system of equations is more conveniently assumed to be represented implicitly by means of a single function $f \in V \to (V \to D) \to D$ so that $f\,x$ returns the right-hand side f_x for each unknown x. Using this representation, the mapping $\sigma \in V \to D$ is a post-solution if $\sigma x \sqsupseteq f\,x\,\sigma$ for all $x \in V$.

A *solver* for a class of equation systems is an algorithm that for each system $\mathcal{C}$ of equations in that class, upon termination, returns a post-solution. Various solvers have been proposed for equation systems where the set of unknowns is finite and the partial ordering D is *Noetherian*, i.e., has no infinite strictly increasing chains. One example of such a solver is *Round Robin* iteration with accumulation. For other solvers, such as the *worklist* iterator, further information about the right-hand side functions is required—namely the set of unknowns whose values are queried during their evaluations. In some cases, though, the system of equations is queried for the values of a few *interesting* unknowns only. Consider, e.g., inter-procedural analysis in the style of [9], e.g., for C. In this application, the unknowns are pairs of program points and abstract calling contexts in which these points are analyzed. The equation system then is queried for the value of the *end point* of the call to the *main* function for the *initial* abstract calling context. From the remaining unknowns only those must be inspected which directly or indirectly contribute to the result of the initial query. A *local* solver is an algorithm meant to deal with such queries. When started with the initial query to an unknown x, it returns a *partial* solution σ. The mapping σ is a partial solution if it provides values for a subset $V' \subseteq V$ of unknowns such that the following holds:

- $x \in V'$;
- For every unknown $y \in V'$, f_y when evaluated on σ, only queries the values of unknowns in V';
- $\sigma x \sqsupseteq f_x\sigma$ for all $x \in V'$.

Note that the equation system in Example 1, when started with a query of x, has a partial solution of only $\sigma = \{x \mapsto 2^{32}\}$. This is because the short-circuit evaluation of the ternary operator :? does not require to inspect the value of y to determine the value of the right-hand side of x for σ.

In this paper we are concerned with the Top-Down local solver **TD** from Le Charlier and Van Hentenryck [19] as depicted in Figure 1. The solver **TD** operates on a partial assignment σ of unknowns to values which initially is empty.

```
void solve(var x) {
  D eval(var y) {
    solve(y);
    infl y ← infl y ∪ {x};
    return σ y;
  }
  if (x ∈ stable ∪ called) return;
  else {
    called ← called ∪ {x};
    stable ← stable ∪ {x};
    tmp ← σ x ⊔ f_x (eval);
    called ← called \ {x};
    if (tmp = σ x) return;
    else {
      σ x ← tmp;
      destabilize(x);
      solve(x);
    }
  }
}

void destabilize(var x) {
  W ← infl x;
  infl x ← ∅ ;
  forall (y ∈ W) {
    if (y ∈ stable) {
      stable ← stable \ {y};
      destabilize(y);
    }
  }
}
```

Fig. 1. The original solver **TD**.

Each unknown for which σ does not provide a value, implicitly is assumed to be mapped to the least value $\bot$. The solver **TD** consists of two functions: solve, and destabilize. The main function of **TD** is the function solve—which when called with an unknown x, is meant to compute a partial solution σ that provides a value for x. Furthermore, **TD** maintains a subset called of unknowns which consists of all unknowns for which the evaluation of the right-hand side has been started but is not yet completed. It also maintains a set stable receiving the unknown x as soon as solving for x has started, where x is only removed when some unknown onto which x recursively depends has changed its value. Initially, both called and stable are empty. A call to solve for the unknown x first checks if x is contained in stable or called. If this is the case, solve immediately returns. Otherwise, x is inserted into stable and called to indicate that solving of x has now started. Then

the right-hand side f_x for the unknown x is evaluated for the local function eval (instead of σ directly).

When within a call solve x, the argument function eval is queried for the value of an unknown y, it ultimately returns the value of y. Before that, however, the solver tries to compute the best possible value for y by calling the function solve for y. Furthermore, eval y keeps track of detected influences between unknowns. All currently known dependences are maintained by **TD** in a mapping infl which, initially, is empty. The function eval records the fact that the variable y was required for computing the value for x, by adding the unknown x in the mapping infl to the value for y. Only then is the value of y (as stored in σ) returned.

In the next step, the function solve joins the old value of σ for x with the new value returned by $f\,x$ eval. Since now the evaluation of the right-hand side is finished, x is removed from the set called, and the joined new value is compared to the old value for x as provided by σ. If these two values are equal, no increase of x has occurred and solve returns. Otherwise, the value of x in σ is updated to the new value. Since the value of x has changed, all unknowns which directly or indirectly may be influenced by x, can no longer be considered as stable and therefore are marked for potential reevaluation. This is the task of the function destabilize.

The function destabilize when called for an unknown x, iterates through all unknowns in the set infl x. Each of the unknowns y which are found to be in stable are removed from stable and then recursively destabilized. Moreover, the value of infl for x is updated to the empty set. In particular this means that before every call of solve, all infl sets of unstable unknowns, i.e., unknown not stable, are empty. Once destabilization has terminated, function solve re-evaluates x by calling itself tail-recursively.

Assume that initially, σ is the empty map, all sets stable, called and infl x for all $x \in V$ are empty. Then we consider the following invariant I:

1. Whenever $y \notin \mathsf{stable} \cup \mathsf{called}$, then $\mathsf{infl}\,y = \varnothing$;
2. Whenever $y \in \mathsf{stable} \backslash \mathsf{called}$, then $\sigma\,y \sqsupseteq f_y\,\sigma$, and for every unknown z whose value is queried during the evaluation of f_y w.r.t. the current σ, $z \in \mathsf{stable} \cup \mathsf{called}$ and $y \in \mathsf{infl}\,z$.

Then we have the following properties:

Lemma 1. 1. The invariant I holds in the beginning and is re-established by each call to solve or eval. Likewise, the set called is preserved and only increased intermediately.
2. After each call of eval y inside a call of solve x, $x \in \mathsf{infl}\,y$, $y \in \mathsf{stable} \cup \mathsf{called}$ and the current value of $\sigma\,y$ is returned.
3. After each call of solve x, the variable x is in the set stable. Moreover, if $\sigma\,x$ has been updated, then x does not recursively influence any unknown in stable $\cap$ called. ∎

By Lemma 1, the program **TD** when started with a call solve x, returns a partial post-solution σ for some set V' of unknowns which contains x — whenever it

terminates. That set V' then is given by all unknowns x which are accessed when recursively re-evaluating the right-hand side of x starting from σ. Technically, this re-evaluation can be triggered by re-setting the set stable to the empty set and again calling solve x.

Lemma 1 also implies that the call solve x is guaranteed to terminate whenever the domain is Noetherian and only finitely many distinct unknowns are encountered during the evaluation.

We remark that one important step in proving Lemma 1 is to prove that conceptually, the evaluation of $f_x\,\sigma$ of the right-hand side of x in a call to solve x for the present mapping σ can be considered as if it happened atomically after evaluation of the unknowns whose values are queried during the evaluation of f_x. For that, it suffices to convince ourselves that every direct query to an unknown $y \notin$ called, during this evaluation, will add y and all unknowns by which it is influenced and that are not in called, into the stable set— ensuring that a second query of y or any unknowns by which y is influenced will return exactly the same result. Note that the unknowns in called are not changed during the evaluation of $f_x\,\sigma$.

Solving the equation system of Example 1 with **TD** produces the following sequence of updates:

	y	x	y	x	$\cdots$	x
x	0	0	1	1	2	$\cdots$	2^{32}
y	0	1	1	2	2	$\cdots$	$2^{32}+1$

We notice that the solving process, although it theoretically terminates after 2^{33} updates, is not efficient. Such inefficiency is typical for equation systems where the value domain contains long increasing chains.

3 Widening and Narrowing

Solving systems of equations usually is based on some form of Kleene iteration, meaning that it consists of a sequence of evaluations of right-hand sides, followed by updates of the unknowns on the corresponding left-hand sides—until all equations in question are satisfied. In case when the partial ordering D of values is not Noetherian, termination of the iteration, though, can no longer be guaranteed. For such cases, Cousot and Cousot [7] propose to first accelerate the iteration by introducing another upper bound operator $\nabla : D \to D \to D$ (the *widening*) to accumulate intermediate values. Conceptually, the idea can be seen as replacing the accumulating version:

$$x = x \sqcup f_x$$

of each equation with the equation:

$$x = x \nabla f_x$$

where ∇ is an upper bound operator which guarantees that every (post) solution of the new system is also a post-solution of the original system. Beyond that,

the widening operator must ensure that values only be increased finitely often. Accordingly, if the set of unknowns which is encountered is finite, **TD** equipped with ∇ (instead of $\sqcup$) will be guaranteed to terminate.

Solving the equation system in Example 1 with **TD** where widening is defined as

$$x \nabla y = \begin{cases} \infty & \text{if } y > x \\ x & \text{otherwise} \end{cases}$$

produces the following sequence of updates:

	y	x
x	0	∞
y	0	∞	∞

Accordingly, the solving process terminates already after two updates, the resulting values for x and y, though, seem unnecessarily large.

In general, many heuristics have been proposed for various domains widening operators which guarantee termination, while at the same time retain enough precision to return useful results. Still, in many cases the results obtained by widening alone, are unsatisfactory. Therefore, Cousot and Cousot propose to perform a second iteration on the system of equation which subsequently may improve a given post-solution [8,10]. The second iteration starts at a post-solution of the system. Given that all right-hand sides represent monotonic functions, the second iteration will result in a *decreasing* sequence of assignments to unknowns—each of which now forms a post-solution. In order to enforce termination of this second iteration, a *narrowing* operator $\Delta \in D \to D \to D$ is introduced. Again, the first argument of this operator is meant to be the former value of an unknown, while the second argument corresponds to the value newly provided by evaluating the corresponding right-hand side. Then the following property should hold:

$$b \sqsubseteq a \quad \Rightarrow \quad b \sqsubseteq a \,\Delta\, b \sqsubseteq a$$

As before, the narrowing operator should enforce that all possibly resulting decreasing chains are finite.

For our running example, we use the following narrowing operator:

$$x \Delta y = \begin{cases} y & \text{if } x = \infty \\ x & \text{otherwise} \end{cases}$$

Starting with the post-solution $\{x \mapsto \infty, y \mapsto \infty\}$, downward iteration produces the following sequence of updates:

	x	y
x	∞	2^{32}
y	∞	∞	$2^{32}+1$

Thus, the solving process terminates already after two updates.

In general, narrowing operators can only be applied if the evaluation of a right-hand side returns less or equal value than currently provided by the left-hand side. If right-hand sides are *not* monotonic, this is not necessarily the case. Non-monotonic right-hand sides, however, inevitably occur in the systems of equations for inter-procedural analysis in the style of Cousot in [9].

Example 2. Consider the equation for a call to a procedure g at an edge in the control-flow graph of the calling procedure f from program point u to program point v. For simplicity, assume that all procedures operate on a global state (full treatment of this kind of constraint system together with a discussion of variations, e.g., for partial contexts is discussed in [3]). For every abstract calling context α of f, we then obtain the equation:

$$\langle v, \alpha \rangle = \langle g, \langle u, \alpha \rangle \rangle$$

where $\langle u, \alpha \rangle, \langle v, \alpha \rangle$ are unknowns representing the abstract values attained at program point u, v when analyzing f for context α, and $\langle g, \beta \rangle$ is the abstract state attained at the exit of the procedure g when called in the abstract context β. Note that in this equation, the context β for which $\langle g, \beta \rangle$ provides the value for the left-hand side $\langle v, \alpha \rangle$, equals the current abstract value for $\langle u, \alpha \rangle$. This means that in a first evaluation of $\langle u, \alpha \rangle$ could return a value β_1, while a later evaluation might return another value β_2, and there is no reason why the values of the two distinct unknowns $\langle g, \beta_1 \rangle$ and $\langle g, \beta_2 \rangle$ should always be related.

Likewise, as elaborated by Apinis et al. in [5], local solving and the two-phase approach to widening/narrowing does not go well together. As a remedy, Apinis et al. propose to *combine* the two operators into one update operator $\boxtriangledown : D \to D \to D$:

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

Let us call this new operator a *warrowing*. Plugging the warrowing operator into a local solver results in a fixpoint iteration which not necessarily performs a single widening iteration followed by a single narrowing iteration. Instead, widening and narrowing is applied in an intertwined manner—with the additional benefit of increasing precision.

We recall from Apinis et al. [5] that every variable assignment σ such that

$$\sigma\, x = \sigma\, x \;\boxtriangledown\; f_x\, \sigma \qquad (x \in V)$$

is also a post-solution of the original system. In general, though, when plugging the combined operator into an arbitrary solver, termination can no longer be guaranteed—even if the original system of equations has monotonic right-hand sides only.

The following sequence of updates that may be exhibited by some solver (not **TD**) for the equation system in Example 1, when the warrowing operator $\boxtriangledown$ is applied for every right-hand side.

	y	y	x	x	y	y	...	y
x	0	0	0	∞	1	1	1	...	2^{32}
y	0	∞	1	1	1	∞	2	...	$2^{32}+1$

Although the iteration terminates, the solving process turns out to be even less efficient than if no widening/narrowing were used. In general, even termination of the iteration can no longer be guaranteed. Therefore, Apinis et al. [4, 5] provide a modifications to several standard solvers so that termination guarantees are retained. The key idea for these modifications is to introduce some kind of ordering on the unknowns which is obeyed during fixpoint iteration. In the following, we show that the enhancement of the top-down solver **TD** by means of the warrowing operator $\boxslash$ is possible — without resorting to such artificial change in the iteration ordering.

4 Enhancing TD

Intuitively, adding an extra ordering on the unknowns for **TD** can be omitted as top-down iteration already imposes an ordering by which unknowns are re-evaluated: no unknown, once called, will be reevaluated before each of the unknowns onto which it depends are stabilized. Surprisingly, this already suffices to ensure termination for equation systems with monotonic right-hand sides, given that the least upper bound operator in **TD** is replaced with the warrowing operator $\boxslash$.

Consider again the equation system in Example 1. When x is solved with the enhanced **TD**, we obtain:

	y	y	x	y	x
x	0	0	0	∞	∞	2^{32}
y	0	∞	1	1	∞	∞

Now, the solving process terminates after a few steps. Note that **TD** does not update the value for y as the value of y is not required for verifying the answer to the initial query of x.

Practical evidence shows, however, that performing widening (and narrowing) for every program point throws away too much information [6]. Therefore, the set of unknowns in whose right-hand sides $\boxslash$ is applied, should be chosen to be as small as possible. Fig. 2 shows the proposed modified top-down solver where the additions are highlighted. The first point to note is that accumulation with $\sqcup$ is now replaced with accumulation by means of the warrowing operator $\boxslash$. Also, once $\boxslash$ is involved for computing the next value for an unknown x, the unknown x is also added to the set $\mathsf{infl}\ x$ in order to trigger reevaluation of x once x changes its value. The second point to note is that, by default, the new values provided by the respective right-hand sides are *directly* used to update the value of the unknown on the right-hand side. The warrowing operator $\boxslash$ is used to combine old values of unknowns with the new values only for dedicated unknowns, namely, those from the set wpoint.

```
void solve(V x) {
  D eval(V y) {
    if (y ∈ called) wpoint ← wpoint ∪ {y};
    solve(y);
    infl[y] ← infl[y] ∪ {x};
    return σ[y];
  }
  if (x ∈ stable ∪ called) return;
  called ← called ∪ {x};
  stable ← stable ∪ {x};
  if (x ∈ wpoint) {
    wpoint ← wpoint \ {x};
    tmp ← σ[x] ⊡ f (x) (eval);
    infl[x] ← infl[x] ∪ {x};
  }
  else tmp ← f (x) (eval);
  called ← called \ {x};
  if (tmp = σ[x]) return;
  else {
    σ[x] ← tmp;
    destabilize(x);
    solve(x);
  }
}

void destabilize(V x) {
  W ← infl[x];
  infl[x] ← ∅ ;
  forall (y ∈ W) {
    if (y ∈ stable) {
      stable ← stable \ {y};
      destabilize(y);
    }
  }
}
```

Fig. 2. The enhanced solver $\mathbf{TD}^{\boxdot}$.

The insight is that in order to enforce termination, widening (and likewise also narrowing) need not be applied everywhere in the system but only for at least one unknown within each cyclic influences of unknowns [6]. For systems of equations originating from control-flow graphs of programs (without recursive procedures), a reasonable choice is to use loop heads as widening points only. In our setting, though, the solver is unaware of the application where the system of equations originates from. Moreover, preprocessing of influences between unknowns is not possible—also due to potential changes of influences between unknowns during

the iteration of the solver. This means that a *dynamic* method must be provided which detects a sufficiently large set of widening points.

In our modification of **TD**, detection of widening points happens inside of the local function eval. Assume that eval is called for unknown y inside a call of solve for an unknown x. Then the variable y is added to wpoint whenever y is found to be already in called. As **TD** is a demand-driven local solver, we have, therefore, dynamically detected a cycle in the dependency graph for the unknowns of the equation system.

As a second improvement, the variable y is *removed* again from wpoint as soon as the iteration on y has stabilized. Such dynamic shrinking of the set wpoint not only accounts for dynamic changes of influences between unknowns, but also may significantly increase precision. Consider, e.g., the unknowns corresponding to a loop in a control-flow graph. Assume that the loop head has been removed from stable, but is no longer contained in wpoint. Then the prior iteration on the loop must have stabilized. Thus, the destabilization of x must have been triggered from outside the loop—implying that applying $\boxtriangle$ in the right-hand side of x is not necessary [1, 2].

Theorem 1. Consider $\mathbf{TD}^{\boxtriangle}$ for a system $\mathcal{C}$ of equations with set V of unknowns. Assume that initially, both stable and called are empty, and solve x has been called for some unknown $x \in V$. Then the following holds:

1. Upon termination, a partial solution for $\mathcal{C}$ is obtained for some subset $V' \subseteq V$ with $x \in V'$.
2. The call is guaranteed terminate if only finitely many unknowns are encountered and one of the following assumptions are met:
   (a) $a \,\Delta\, b = a$, i.e., narrowing is effectively switched off, or
   (b) all right-hand sides are monotonic.

Clearly, Theorem 1 is unsatisfactory, as it does not provide a termination guarantee for the case where right-hand sides are *not* monotonic. With contrived non-monotonic systems, virtually every solver using $\boxtriangle$ as is, can be forced into non-termination. In that sense, the second assertion for monotonic right-hand sides cannot easily be improved. It gives an indication, though, that *practically* termination can be hoped for. At the price of giving up some opportunities for further narrowing steps, we can always *enforce* termination. We could, e.g., modify the warrowing operator $\boxtriangle$ so that the number of switches from widening to narrowing is bounded at each occurrence of $\boxtriangle$ in the equation system.

For the proof of Theorem 1, we remark that the properties stated in Lemma 1 for solver **TD** also hold true for the program $\mathbf{TD}^{\boxtriangle}$. From that, the statement 1 of Theorem 1 immediately follows. Therefore here we concentrate on the proof of termination.

5 Proof of termination

We perform an induction on the number n of unknowns queried during the evaluation of the unknown x and which are either equal to x or not contained

in the set called. For $n = 0$, the evaluation immediately terminates. Now assume that $n > 0$. To establish a contradiction, assume that the call solve x does not terminate. Since by inductive hypothesis, each recursive call to eval terminates, the tail-recursive call to solve x is executed infinitely often. This means that the value of x must be updated infinitely often, and thus its right-hand side also be re-evaluated infinitely often.

We claim that then during each evaluation of the right-hand side of x, x is added to the set wpoint—implying that each new value is obtained by application of the operator $\boxtriangle$. Assume for a contradiction that this were not the case. Let V_i denote the set of unknowns which are not in called which are queried during the ith evaluation of the right-hand side of x. If before the $(i+1)$th evaluation, x is *not* contained in wpoint, then for none of the unknowns $y \in S_i$, the evaluation of their right-hand sides may have queried the value of x. After the ith evaluation of the right-hand side of x, each unknown $y \in S_i$ is stable and none of them is contained in the set infl(x). Therefore, none of them is removed from the set stable when the value of x in σ is updated. Therefore, the next as well as any subsequent evaluation of the right-hand side will query always the same set of unknowns, i.e., $S_j = S_i$ for all $j \geqslant i$, and all the unknowns in there will be stable. But then the $(i+2)$th value returned for x will equal the $(i+1)$th value for x—in contradiction to our assumption.

Let $d_1 \neq d_2 \neq \ldots$ be the sequence of values for x after the ith update. In particular, between any two updates, x must have been destabilized (otherwise, solve x would have terminated immediately), implying that x recursively has been found to influence itself. To establish this influence, x will have been inserted into the set wpoint, the latest during the first evaluation of its right-hand side and will stay there until its value has stabilized. Therefore, for each $i \geqslant 1$ it holds that $d_{i+1} = d_i \boxtriangle b_i$ for suitable values b_i.

First assume that narrowing returns its first argument, i.e., is effectively switched off. Then $d_{i+1} = d_i \nabla b_i$ (for all $i \geqslant 1$). Since the operator ∇ is a widening, the sequence d_i must eventually be stable—contradicting the assertion that $d_i \neq d_{i+1}$ for all i.

Therefore, now consider the second sufficient condition for termination as stated in the theorem, namely, that all right-hand sides are monotonic. Let m denote the maximal index such that for all $i < m$, $d_{i+1} = d_i \nabla b_i$. Since ∇ is a widening, such an m exists. For that m, we claim:

Claim. For all $j \geqslant m$, $d_j \sqsupset d_{j+1}$.

Given that the claim holds, $d_{j+1} = d_j \Delta b_j$ for all $j \geqslant m$. Now since Δ is a narrowing operation, the sequence $d_j, j \geqslant m$ must become ultimately stable—again contradicting the assertion that $d_i \neq d_{i+1}$ for all i.

It remains to prove the claim. In order to do so, we introduce a few extra notions. For a set V and a lattice $(D, \sqsubseteq)$, let g be a function $(V \to D) \to D$. Given a mapping $\sigma \in V \to D$, the function g *depends* on the set $V' \subseteq V$ of unknowns (relative to σ) if for all $\sigma' \in V \to D$ such that V' is the smallest subset such that $\sigma|_{V'} = \sigma'|_{V'}$ implies that $g\,\sigma = g\,\sigma'$. We say that g *references* unknowns from

$R(g,\sigma) \subseteq V$ w.r.t. the mapping σ if the evaluation of the strategy tree [17] for g for the mapping σ queries exactly the unknowns $R(g,\sigma)$. It can be shown that referencing is an over-approximation of dependency, i.e., all mappings σ' such that $\sigma|_{R(g,\sigma)} = \sigma'|_{R(g,\sigma)}$ implies $g\,\sigma = g\,\sigma'$. We have:

Lemma 2. If g is monotonic then for all $\sigma, \sigma' : V \to D$,

$$\forall x \in R(g,\sigma).\ \sigma'\,x \sqsubseteq \sigma\,x$$

then

$$g\,\sigma' \sqsubseteq g\,\sigma\ .$$

Proof. Assume for a contradiction that there are mappings σ, σ' such that $\forall x \in R(g,\sigma).\ \sigma'\,x \sqsubseteq \sigma\,x$, but $g\,\sigma' \not\sqsubseteq g\,\sigma$. We construct

$$\sigma''\,x = \begin{cases} \sigma\,x & \text{if } x \in R(g,\sigma) \\ \sigma'\,x & \text{otherwise.} \end{cases}$$

We have

$$\forall x \in V.\ \sigma'\,x \sqsubseteq \sigma''\,x$$

and therefore, by monotonicity of g,

$$g\,\sigma' \sqsubseteq g\,\sigma''\ .$$

Because reference is an over-approximation of dependence we also have that $g\,\sigma'' = g\,\sigma$. Thus, we conclude that $g\,\sigma' \sqsubseteq g\,\sigma$ holds—in contradiction to our assumption.∎

Now we are ready to prove our claim:

Lemma 3. During a call to $\mathsf{solve}(x)$, for some unknown x, once the sequence of values u_j provided by evaluating the right-hand side for the unknown x starts to descend, it will stabilize or keep descending, i.e., $d_j \sqsupseteq d_{j+1}$ for all $j \geqslant m$.

Proof. Evaluation of the right-hand side of the unknown x, when solving x generates a sequence $(z_1,\sigma_1),\ldots,(z_r,\sigma_r)$, where the first components z_i are the unknowns that are re-evaluated and the second components σ_i are the respective mappings at the time when the evaluation of the right-hand side of z_i has been completed, and the ordering is the ordering in which the new values for the unknowns are determined. In particular, the last unknown in this sequence z_r equals x. Let us call this the *trace* of the evaluation of x.

Assume that the evaluation of the right-hand side of the unknown x returned a smaller value u than the value currently stored in σ. At that point in time, all referenced unknowns $R(f\ z_i,\sigma)$ that are not in the set called are stable. Assume that $\sigma\ x\ \Delta\ u\ \sqsubset\ \sigma\ x$. After evaluation of x has been completed, $\mathsf{destabilize}$ is called for x, as we assume that the value for x continuously changes. The function $\mathsf{destabilize}$ will remove all unknowns from stable that might need to be updated, as they are (transitively) influenced by x. Subsequently, solve is again called for the unknown x.

As before, evaluation of the right-hand side of x will generate a trace (q_1, σ'_1), $\ldots, (q_n, \sigma'_n)$. Recall that the last unknown to be updated again will be x. Now we show that those unknowns q_i that have already occurred in the sequence $z_1, \ldots, z_r$ will receive a smaller value or stay the same. For the proof, we perform induction over the prefixes of the trace $(q_1, \sigma'_1), \ldots, (q_n, \sigma'_n)$.

Base Trivial.

Step Assume that the values of $q_1, \ldots, q_{i-1}$ that occurred already in the sequence $z_1, \ldots, z_r$ stayed the same or decreased—according to the induction hypothesis. As for the update to q_i, we only need to consider the case that q_i has already occurred, i.e., that $q_i = z_j$ for some index j.

As TD solves all unknowns occurring in the right-hand side of q_i beforehand, except when they are in called or in stable, only unknowns which did not occur among the $z_1, \ldots, z_r$ may receive a larger or incomparable value. This means that the last evaluation of z_j did not depend on these unknowns. By Lemma 2, however, the value returned for the right-hand side of $q_i = z_j$ then will be smaller or stay the same. This concludes the proof of the claim and hence of Theorem 1. ∎

6 Conclusion

We have presented a moderate improvement of the generic local solver **TD** which enables the solver to use widening and narrowing in a convenient way. Upon termination, the resulting algorithm always returns a partial solution from which a partial post-solution can be extracted. Moreover, termination can be guaranteed whenever only finitely many unknowns are encountered and either no narrowing is used or right-hand sides are all monotonic. During fixpoint iteration, it dynamically not only detects dependences between unknowns but also those unknowns which require widening/narrowing. Compared to the solvers presented in [2, 4, 5], the solver $\mathbf{TD}^{\boxtimes}$ is simpler as no explicit priorities of unknowns need to be maintained. The latter solvers, however, can be enhanced to deal with *side-effects* inside systems of equations. Side-effects allow to generate contributions to unknowns different from the left-hand side. This mechanism is convenient, e.g., for combining flow-insensitive analysis with inter-procedural analysis [3]. It is still open whether the solver $\mathbf{TD}^{\boxtimes}$ can be enhanced to deal with such systems as well. Also, in the application of **TD** inside the CIAO system, extra measures are taken to limit the number of unknowns to be considered [14]. It would be interesting to see how the plain version considered here works together with such extra methods.

Acknowledgements

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). This work was funded by institutional research grant IUT2-1 from the Estonian Research Council.

References

1. Gianluca Amato and Francesca Scozzari. Localizing widening and narrowing. In F. Logozzo and M. Fändrich, editors, *Static Analysis - 20th Int. Symp. (SAS)*, pages 25–42. LNCS 7935, Springer, 2013.
2. Gianluca Amato, Francesca Scozzari, Helmut Seidl, Kalmer Apinis, and Vesal Vojdani. Efficiently intertwining widening and narrowing. March 2015, arXiv:1503.00883 [cs.PL].
3. K. Apinis, H. Seidl, and V. Vojdani. Side-effecting constraint systems: A Swiss army knife for program analysis. In *Programming Languages and Systems - 10th Asian Symp. (APLAS)*, pages 157–172. LNCS 7705, Springer, 2012.
4. Kalmer Apinis. *Frameworks for Analyzing Multi-Threaded C.* PhD thesis, Institut für Informatik, Technische Universität München, June 2014.
5. Kalmer Apinis, Helmut Seidl, and Vesal Vojdani. How to combine widening and narrowing for non-monotonic systems of equations. In *34th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 377–386. ACM, 2013.
6. François Bourdoncle. Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity. In *Programming Language Implementation and Logic Programming, 2nd Int. Workshop (PLILP)*, pages 307–323. LNCS 456, Springer, 1990.
7. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In B. Robinet, editor, *2nd Int. Symp. on Programming, Paris, France*, page 106—130. Dunod, Paris, 1976.
8. P. Cousot and R. Cousot. Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Annual ACM Symp. on Principles of Programming Languages (POPL)*, pages 238–252. ACM Press, 1977.
9. 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.
10. Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen, editors, *6th Annual ACM Symp. on Principles of Programming Languages (POPL)*, pages 269–282, 1979.
11. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th Annual ACM Symp. on Principles of Programming Languages (POPL)*, pages 84–96. ACM Press, 1978.
12. Christian Fecht and Helmut Seidl. Propagating Differences: An Efficient New Fixpoint Algorithm for Distributive Constraint Systems. *Nord. J. Comput.*, 5(4):304–329, 1998.
13. Christian Fecht and Helmut Seidl. A faster solver for general systems of equations. *Science of Computer Programming*, 35(2):137–161, 1999.
14. M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(2):187–223, March 2000.
15. Manuel V Hermenegildo, Germán Puebla, Francisco Bueno, and Pedro López-García. Integrated program debugging, verification, and optimization using abstract interpretation (and the CIAO system preprocessor). *Science of Computer Programming*, 58(1):115–140, 2005.

16. Martin Hofmann, Aleksandr Karbyshev, and Helmut Seidl. Verifying a Local Generic Solver in Coq. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis*, pages 340–355. LNCS 6337, Springer, September 2010.
17. Martin Hofmann, Aleksandr Karbyshev, and Helmut Seidl. What is a pure functional? In *37th Int. Conf. on Automata, Languages and Programming (ICALP (2))*, pages 199–210. LNCS 6199, Springer, 2010.
18. G.A. Kildall. A unified approach to global program optimization. In *ACM Symp. on Principle of Programming Languages (POPL)*, pages 194–206. ACM, 1973.
19. 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.
20. Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
21. K. Muthukumar and M. Hermenegildo. Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, April 1990.
22. G. Puebla and M. Hermenegildo. Optimized algorithms for the incremental analysis of logic programs. In *Static Analysis - Third Int. Symp. (SAS)*, pages 270–284. LNCS 1145, Springer, 1996.
23. Helmut Seidl and Christian Fecht. Interprocedural Analyses: A Comparison. *J. Log. Program.*, 43(2):123–156, 2000.
24. Vesal Vojdani. *Static Data Race Analysis of Heap-Manipulating C Programs*. PhD thesis, University of Tartu., December 2010.