Mathematical Structures in Computer Science (2021), **31**, pp. 1090–1134
doi:10.1017/S0960129521000499

CAMBRIDGE UNIVERSITY PRESS

PAPER

Three improvements to the top-down solver

Helmut Seidl* and Ralf Vogler

TU München, Garching, Germany
*Corresponding author. Email: seidl@in.tum.de

(Received 31 July 2020; revised 20 December 2021; accepted 21 December 2021; first published online 3 February 2022)

Abstract

The local solver **TD** is a generic fixpoint engine which explores a given system of equations on demand. It has been successfully applied to the interprocedural analysis of procedural languages. The solver **TD** gains efficiency by detecting dependencies between unknowns on the fly. This algorithm has been recently extended to deal with *widening* and *narrowing* as well. In particular, it has been equipped with an automatic detection of widening and narrowing points. That version, however, is only guaranteed to terminate under two conditions: only finitely many unknowns are encountered, and all right-hand sides are *monotonic*. While the first condition is unavoidable, the second limits the applicability of the solver. Another limitation is that the solver maintains the current abstract values of *all* encountered unknowns instead of a minimal set sufficient for performing the iteration. By consuming unnecessarily much space, interprocedural analyses may not succeed on seemingly small programs. In the present paper, we therefore extend the *top-down* solver **TD** in three ways. First, we indicate how the restriction to monotonic right-hand sides can be lifted without compromising termination. We then show how the solver can be tuned to store abstract values only when their preservation is inevitable. Finally, we also show how the solver can be extended to *side-effecting* equation systems. Right-hand sides of these may not only provide values for the corresponding left-hand side unknowns but at the same time produce contributions to other unknowns. This practical extension has successfully been used for a seamless combination of context-sensitive analyses (e.g., of local states) with flow-insensitive analyses (e.g., of globals).

Keywords: Static analysis; local solvers; abstract interpretation

1. Introduction

Static analysis tools based on abstract interpretation are complicated software systems. They are complicated due to the complications of programming language semantics and the subtle invariants required to achieve meaningful results. They are also complicated when dedicated analysis algorithms are required to deal with certain types of properties, for example, one algorithm for inferring points-to information and another for checking array-out-of-bounds accesses. Thus, static analysis tools themselves are subject to subtle programming errors. From a software engineering perspective, it is therefore meaningful to separate the *specification* of the analysis as much as possible from the *algorithm* solving the analysis problem for a given program. In order to be widely applicable, this algorithm therefore should be as *generic* as possible.

In abstract interpretation-based static analysis, the analysis of a program, notably an imperative or object-oriented program, can naturally be compiled into a system of abstract equations. The unknowns represent places where invariants regarding the reaching program states are desired. Such unknowns could be, for example, plain program points or, in case of interprocedural analysis,

program points together with abstract calling contexts. The sets of possible abstract values for these unknowns then correspond to classes of possible invariants and typically form complete lattices. Since for infinite complete lattices of abstract values the number of calling contexts is also possibly infinite, interprocedural analysis has generally to deal with *infinite* systems of equations. It turns out, however, that in this particular application, only the values of those unknowns are of interest that directly or indirectly influence some initial unknown. Here, *local* solving comes as a rescue: a local solver can be started with one initial unknown of interest. It then explores only those unknowns whose values contribute to the value of this initial unknown.

One such generic local solver is the *top-down solver* **TD** (Charlier and Van Hentenryck 1992; Muthukumar and Hermenegildo 1990). Originally, the **TD** solver has been conceived for goal-directed static analysis of PROLOG programs (Hermenegildo and Muthukumar 1989, 1992) while some basic ideas can be traced back to Bruynooghe et al. (1987). The same technology as developed for PROLOG later turned out to be useful for imperative programs as well (Hermenegildo 2000) and also was applied to other languages via translation to CLP programs (Gallagher and Henriksen 2006; Hermenegildo et al. 2007). A variant of it is still at the heart of the program analysis system CIAO (Hermenegildo et al. 2012, 2005). The **TD** solver is interesting in that it completely abandons specific data structures such as work-lists, but solely relies on recursive evaluation of (right-hand sides of) unknowns.

Subsequently, the idea of using generic local solvers has also been adopted for the design and implementation of the static analysis tool GOBLINT (Vojdani et al. 2016) – now targeted at the abstract interpretation of multi-threaded C. Since the precise analysis of programming languages requires to employ abstract domains with possibly infinite ascending or descending chains, the solvers provided by GOBLINT were extended with support for *widening* as well as *narrowing* (Amato et al. 2016). Recall that widening and narrowing operators have been introduced in Cousot and Cousot (1977, 1992) as an iteration strategy for solving finite systems of equations: in a first phase, an upward Kleene fixpoint iteration is accelerated to obtain some post-solution, which then, in a second phase, is improved by an accelerated downward iteration. This strict separation into phases is given up by the algorithms from Amato et al. (2016). Instead, the ascending iteration on one unknown is possibly intertwined with the descending iteration on another. The idea is to avoid unnecessary loss of precision by starting an improving iteration on an unknown as soon as an over-approximation of the least solution for this unknown has been reached. On the downside, the solvers developed in Amato et al. (2016) are only guaranteed to terminate for *monotonic* systems of equations. Systems for *interprocedural* analysis, however, are not necessarily monotonic. The problems concerning termination as encountered by the non-standard local iteration strategy from Amato et al. (2016) were resolved in Frielinghaus et al. (2016) where the switch from a widening to a narrowing iteration for an unknown was carefully redesigned.

When reviewing advantages and disadvantages of local generic solvers for GOBLINT, we observed in Apinis et al. (2016) that the extension with widening and narrowing from Amato et al. (2016) could nicely be applied to the solver **TD** as well – it was left open, though, how termination can be enforced not only for monotonic, but for arbitrary systems of equations. In this paper, we settle this issue and present a variant of the **TD** solver with widening and narrowing which is guaranteed to terminate for *all* systems of equations – whenever only finitely many unknowns are encountered.

Besides termination, another obstacle for the practical application of static analysis is the excessive space consumption incurred by storing abstract values for all encountered unknowns. Storing all these values allows interprocedural static analysis tools like GOBLINT only to scale to medium-sized programs of about 100k LOC. This is in stark contrast to the tool ASTRÉE, which – while also being implemented in OCAML – succeeds in analyzing much larger input programs (Cousot et al. 2009). One reason is that ASTRÉE only keeps the abstract values of a *subset* of unknowns in memory, which is sufficient for proceeding with the current fixpoint iteration. As our second contribution, we therefore show how to realize a space-efficient iteration

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

strategy within the framework of generic local solvers. Unlike ASTRÉE which iterates over the syntax, generic local solvers are application-independent. They operate on systems of equations – no matter where these are extracted from. As right-hand sides of equations are treated as black boxes, inspection of the syntax of the input program is out of reach. Since for local solvers the set of unknowns to be considered is only determined during the solving iteration, also the subset of unknowns sufficient for reconstructing the analysis result must be identified *on the fly*. Our idea for that purpose is to *instrument* the solver to observe when the value of an unknown is queried, for which an iteration is already on the way. For equation systems corresponding to standard control-flow graphs, these unknowns correspond to the loop heads and are therefore ideal candidates for widening and narrowing to be applied. That observation was already exploited in Apinis et al. (2016) to identify a set of unknowns where widening and narrowing should be applied. The values of these unknowns also suffice for reconstructing the values of all remaining unknowns without further iteration. Finally, we present an extension of the **TD** solver with *side effects*. Side effects during solving means that, while evaluating the right-hand side for some unknown, contributions to other unknowns may be triggered. This extension allows to nicely formulate partial context-sensitivity at procedure calls and also to combine flow-insensitive analysis, for example, of global data, with context-sensitive analysis of the local program state (Apinis et al. 2012). The presented solvers have been implemented as part of GOBLINT, a static analysis framework written in OCAML.

This paper is organized as follows: First, we recall the basics of abstract interpretation. In Section 2, we show how the concrete semantics of a program can be defined using a system of equations with monotonic right-hand sides. In Section 3, we describe how abstract equation systems can be used to compute *sound* approximations of the collecting semantics. Since the sets of unknowns of concrete and abstract systems of equations may differ, we argue that soundness can only be proven relative to a *description relation* between concrete and abstract unknowns. We also recall the concepts of *widening* and *narrowing* and indicate in which sense these can be used to effectively solve abstract systems of equations. In Section 4, we present the generic local solver $\mathbf{TD_{term}}$. In Section 5, we show that it is guaranteed to terminate even for abstract equation systems with non-monotonic right-hand sides. In Section 6, we prove that it will always compute a solution that is a sound description of the least solution of the corresponding concrete system. In Section 7, we present the solver $\mathbf{TD_{space}}$, a space-efficient variation of $\mathbf{TD_{term}}$ that only keeps values at widening points. In Section 8, we prove that it terminates and similar to $\mathbf{TD_{term}}$ computes a sound description of the least solution of the corresponding concrete system. In Section 9, we introduce side-effecting systems of equations, and the solver $\mathbf{TD_{side}}$, a variation of $\mathbf{TD_{term}}$. In order to argue about soundness in presence of side effects, it is now convenient to consider description relations between concrete and abstract unknowns which are not static, but *dynamically computed* during fixpoint iteration, that is, themselves depend on the analysis results. In Section 10, we discuss the results of evaluating the solvers on a set of programs. In Section 11, we summarize our main contributions. Throughout the presentation, we exemplify our notions and concepts by small examples from interprocedural program analysis.

2. Concrete Systems of Equations

Solvers are meant to provide solutions to systems of equations over some complete lattice. Assume that $\mathcal{X}$ is a (not necessarily finite) set of *unknowns* and $\mathbb{D}$ a complete lattice. Then, a system of equations $\mathcal{E}$ (with unknowns from $\mathcal{X}$ and values in $\mathbb{D}$) assigns a right-hand side F_x to each unknown $x \in \mathcal{X}$. Since we not only interested in the values assigned to unknowns but also in the *dependencies* between these, we assume that each right-hand side F_x is a function of type $(\mathcal{X} \to \mathbb{D}) \to \mathbb{D} \times \mathcal{P}(\mathcal{X})$.

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

```
  main() {
0     h1();
1     p();
2 }

  p() {
3     if (*) {
4         h1();
5         p();
      } else
6         h2();
7 }
```

Figure 1. An example program with procedures.

- The first component of the result of F_x is the *value* for x;
- The second component is a set of unknowns which is *sufficient to determine* the value.

More formally, we assume for the second component of F_x that for every assignment $\sigma : \mathcal{X} \to \mathbb{D}$, $F_x\,\sigma = (d, X)$ implies that for any other assignment $\sigma' : \mathcal{X} \to \mathbb{D}$ with $\sigma|_X = \sigma'|_X$, that is, which agrees with σ on all unknowns from $\mathcal{X}$, $F_x\,\sigma = F_x\,\sigma'$ holds as well.

The set X thus can be considered as a superset of all unknowns onto which the unknown x *depends* – w.r.t. the assignment σ. We remark this set very well may be different for different assignments. For convenience, we denote these two components of the result $F_x\,\sigma$ as $(F_x\,\sigma)_1$ and $(F_x\,\sigma)_2$, respectively.

Systems of equations can be used to formulate the *concrete* (accumulating) semantics of a program. In this case, the complete lattice $\mathbb{D}$ is of the form $\mathcal{P}(\mathcal{Q})$ where $\mathcal{Q}$ is the set of states possibly attained during program execution. Furthermore, all right-hand sides of the concrete semantics should be *monotonic* w.r.t. the natural ordering on pairs. This means that on larger assignments, the sets of states for x as well as the set of contributing unknowns may only increase.

Example 1. For $\mathcal{X} = \mathcal{Q} = \mathbb{Z}$, a system of equations could have right-hand sides $F_x : (\mathbb{Z} \to \mathcal{P}(\mathbb{Z})) \to (\mathcal{P}(\mathbb{Z}) \times \mathcal{P}(\mathbb{Z}))$ where, for example,

$$F_1\,\sigma = (\{0\}, \emptyset)$$
$$F_2\,\sigma = (\sigma\ 1 \cup \sigma\ 3, \{1, 3\})$$

Thus, F_1 always returns the value $\{0\}$ and accordingly depends on no other unknown, F_2 on the other hand returns the union of the values of unknowns 1 and 3. Therefore, it depends on both of them. □

Example 2. As a running example, consider the program from Figure 1 which consists of the procedures main and p. Assume that these operate on a set $\mathcal{Q}$ of program states where the functions $h_1, h_2 : \mathcal{Q} \to \mathcal{P}(\mathcal{Q})$ represent the semantics of basic computation steps. The collecting semantics for the program provides subsets of states for each program point $u \in \{0, \ldots, 7\}$ and each possible calling context $q \in \mathcal{Q}$. The right-hand side function $F_{\langle u,q\rangle}$ for each such pair $\langle u, q\rangle$ is given by:

$$\begin{aligned}
F_{\langle 0,q\rangle}\,\sigma &= (\{q\}, \emptyset)\\
F_{\langle 1,q\rangle}\,\sigma &= (\cup \{\, h_1(q') \mid q' \in \sigma\ \langle 0, q\rangle \,\},\ \{\, \langle 0, q\rangle \,\})\\
F_{\langle 2,q\rangle}\,\sigma &= (\{\, \mathsf{combine}\ q_1\ q_2 \mid q_1 \in \sigma\ \langle 1, q\rangle, q_2 \in \sigma\ \langle 7, q_1\rangle \,\},\\
&\qquad \{\, \langle 1, q\rangle \,\} \cup \{\, \langle 7, q_1\rangle \mid q_1 \in \sigma\ \langle 1, q\rangle \,\})\\
F_{\langle 3,q\rangle}\,\sigma &= (\{\, q \,\}, \emptyset)\\
F_{\langle 4,q\rangle}\,\sigma &= (\sigma\ \langle 3, q\rangle,\ \{\, \langle 3, q\rangle \,\})
\end{aligned}$$

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

$$
\begin{aligned}
F_{\langle 5,q\rangle}\,\sigma &= (\,\cup\,\{\, h_1(q_1) \mid q_1 \in \sigma\ \langle 4,q\rangle \,\}\,,\,\{\, \langle 4,q\rangle \,\}\,)\\
F_{\langle 6,q\rangle}\,\sigma &= (\sigma\ \langle 3,q\rangle,\ \{\, \langle 3,q\rangle \,\}\,)\\
F_{\langle 7,q\rangle}\,\sigma &= (\,\{\, \mathsf{combine}\ q_1\ q_2 \mid q_1 \in \sigma\ \langle 5,q\rangle, q_2 \in \sigma\ \langle 7,q_1\rangle \,\}\\
&\qquad \cup\ \{\, h_2(q_1) \mid q_1 \in \sigma\ \langle 6,q\rangle \,\}\,,\\
&\qquad \{\, \langle 5,q\rangle \,\} \cup \{\, \langle 7,q_1\rangle \mid q_1 \in \sigma\ \langle 5,q\rangle \,\} \cup \{\, \langle 6,q\rangle \,\}\,)
\end{aligned}
$$

Recall that in presence of local scopes of program variables, the state after a call may also depend on the state before the call. Accordingly, we use an auxiliary function $\mathsf{combine} : \mathcal{Q} \to \mathcal{Q} \to \mathcal{Q}$ which determines the program state after a call from the state before the call and the state attained at the end of the procedure body. For simplicity, we have abandoned an extra function enter for modeling passing of parameters as considered, e.g., in Apinis et al. (2012) and thus assume that the full program state of the caller before the call is passed to the callee. If the set $\mathcal{Q}$ is of the form $\mathcal{Q} = \mathcal{A} \times \mathcal{B}$ where $\mathcal{A}, \mathcal{B}$ are the sets of local and global states, respectively, the function combine could, for example, be defined as

$$\mathsf{combine}\ (a_1, b_1)\ (a_2, b_2) = (a_1, b_2)$$

We remark that the sets of unknowns onto which the right-hand sides for $\langle 2, q\rangle$ as well as $\langle 7, q\rangle$ depend themselves depend on the assignment σ. □

Besides the concrete values provided for the unknowns, we also would like to determine the subset of unknowns which contribute to a particular subset of unknowns of *interest*. Restricting to this subset has the practical advantage that calculation may be restricted to a perhaps quite small number of unknowns only. Also, that subset in itself contains some form of *reachability* information. For interprocedural analysis, for example, of the program in Example 2, we are interested in all pairs $\langle \mathsf{ret}_{main}, q\rangle, q \in Q_0$, that is, the endpoint of the initially called procedure *main* for every initial calling context $q \in Q_0$. In the Example 2, these would be of the form $\langle 2, q\rangle, q \in Q_0$. In order to determine the sets of program states for the unknowns of interest, it suffices to consider only those calling contexts for each procedure p (and thus each program point within p) in which p is possibly called. The set of all such pairs is given as the *least* subset of unknowns which (directly or indirectly) *influences* any of the unknowns of interest.

More technically for a system $\mathcal{E}$ of equations, consider an assignment $\sigma : \mathcal{X} \to \mathbb{D}$ and a subset $dom \subseteq \mathcal{X}$ of unknowns. Then, dom is called $(\sigma, \mathcal{E})$-*closed* if $(F_x\,\sigma)_2 \subseteq dom$ is satisfied for all $x \in dom$. The pair (σ, dom) is called a *partial post-solution* of $\mathcal{E}$, if dom is $(\sigma, \mathcal{E})$-closed, and for each $x \in dom$, $\sigma\ x \sqsupseteq (F_x\,\sigma)_1$ holds. The partial post-solution (σ, dom) of $\mathcal{E}$ is *total* (or just a post-solution of $\mathcal{E}$) if $dom = \mathcal{X}$. In this case, we also skip the second component in this pair.

Example 3. Consider the following system of equations with $\mathcal{X} = \mathcal{Q} = \mathbb{Z}$ where

$$
\begin{aligned}
F_1\,\sigma &= (\sigma\ 2, \{2\})\\
F_2\,\sigma &= (\{2\}, \emptyset)\\
F_3\,\sigma &= (\{3\}, \emptyset)\\
F_x\,\sigma &= (\emptyset, \emptyset) \qquad \text{otherwise}
\end{aligned}
$$

Assume we are given the set of unknowns of interest $X = \{1\}$. Then, (σ, dom) with $dom = \{1, 2\}$, $\sigma\ 1 = \{2\}$ and $\sigma\ 2 = \{2\}$ is the *least* partial post-solution with $X \subseteq dom$. For $X' = \{1, 3\}$ on the other hand, the least partial solution (σ', dom') with $X' \subseteq dom'$ is given by $dom' = \{1, 2, 3\}$ and $\sigma'\ 1 = \{2\}, \sigma'\ 2 = \{2\}$ and $\sigma'\ 3 = \{3\}$. □

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

For monotonic systems such as those used for representing the collecting semantics, and any set $X \subseteq \mathcal{X}$ of unknowns of interest, there always exists a *least* partial solution comprising X.

Proposition 1. *Assume that the system $\mathcal{E}$ of equations is monotonic. Then for each subset $X \subseteq \mathcal{X}$ of unknowns, consider the set P of all partial post-solutions $(\sigma', dom') \in ((\mathcal{X} \to \mathbb{D}) \times \mathcal{P}(\mathcal{X}))$ so that $X \subseteq dom'$. Then, P has a unique least element (σ_X, dom_X). In particular, $\sigma_X\, x' = \bot$ for all $x' \notin dom_X$.*

Proof. Consider the complete lattice $\mathbb{L} = (\mathcal{X} \to \mathbb{D}) \times \mathcal{P}(\mathcal{X})$. The system $\mathcal{E}$ defines a function $F : \mathbb{L} \to \mathbb{L}$ by $F(\sigma_1, X_1) = (\sigma_2, X_2)$ where

$$\begin{aligned} X_2 &= X \cup X_1 \cup \bigcup\{(F_x\, \sigma_1)_2 \mid x \in X_1\} \\ \sigma_2\, x &= (F_x\, \sigma_1)_1 \qquad \text{for } x \in X_1 \text{ and } \bot \text{ otherwise} \end{aligned}$$

Since each right-hand side F_x in $\mathcal{E}$ is monotonic, so is the function F. Moreover, (σ, dom) is a post-fixpoint of F iff (σ, dom) is a partial post-solution of $\mathcal{E}$ with $X \subseteq dom$. By the fixpoint theorem of Knaster-Tarski, F has a unique least post-fixpoint – which happens to be also the least fixpoint of F. □

For a given set X, there thus is a *least* partial solution of $\mathcal{E}$ with a least domain dom_X comprising X. Moreover for $X \subseteq X'$ and least partial solutions (σ_X, dom_X), $(\sigma_{X'}, dom_{X'})$ comprising X and X', respectively, we have $dom_X \subseteq dom_{X'}$ and $\sigma_{X'}\, x = \sigma_X\, x$ for all $x \in dom_X$. In particular, this means for the least total solution σ that $\sigma_X\, x = \sigma\, x$ whenever $x \in dom_X$.

Example 4. Consider the program from Example 2. Assume that $\mathcal{Q} = \{q_0, q_1, q_2\}$ where the set of initial calling contexts is given by $\{q_1\}$. Accordingly, the set of unknowns of interest is given by $X = \{\langle 2, q_1\rangle\}$ (2 being the return point of main). Assume that the functions h_1, h_2 are given by

$$\begin{aligned} h_1 &= \{q_0 \mapsto \emptyset, q_1 \mapsto \{q_2\}, q_2 \mapsto \{q_0\}\} \\ h_2 &= \{q_0 \mapsto \{q_0\}, q_1 \mapsto \emptyset, q_2 \mapsto \emptyset\} \end{aligned}$$

while the function combine always returns its second argument, that is, combine $q\, q' = q'$.

Let *dom* denote the set

$$\begin{aligned} &\{\langle 0, q_1\rangle, \langle 1, q_1\rangle, \langle 2, q_1\rangle, \\ &\langle 3, q_0\rangle, \langle 4, q_0\rangle, \langle 5, q_0\rangle, \langle 6, q_0\rangle, \langle 7, q_0\rangle, \\ &\langle 3, q_2\rangle, \langle 4, q_2\rangle, \langle 5, q_2\rangle, \langle 6, q_2\rangle, \langle 7, q_2\rangle\}. \end{aligned}$$

Together with the assignment $\sigma : dom \to \mathcal{P}(\mathcal{Q}) \times \mathcal{P}(\mathcal{X})$ as shown in Figure 2, we obtain the *least* partial solution of the given system of equations which we refer to as the *collecting semantics* of the program. We remark that *dom* only has calls of procedure p for the calling contexts q_0 and q_2. □

3. Abstract Systems of Equations

Systems of *abstract* equations are meant to provide sound information for concrete systems. In order to distinguish abstract systems from concrete ones, we usually use superscripts $\sharp$ at all corresponding entities. Abstract systems of equations differ from concrete ones in several aspects:

- Right-hand side functions need no longer be monotonic.
- Right-hand side functions should be *effectively computable* and thus may access the values only of *finitely many* other unknowns in the system (which need not be the case for concrete systems).

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

$\langle 0, q_1\rangle$	$\{q_1\}$	$\emptyset$
$\langle 1, q_1\rangle$	$\{q_2\}$	$\{\langle 0, q_1\rangle\}$
$\langle 2, q_1\rangle$	$\{q_0\}$	$\{\langle 1, q_1\rangle, \langle 7, q_2\rangle\}$
$\langle 3, q_0\rangle$	$\{q_0\}$	$\emptyset$
$\langle 4, q_0\rangle$	$\{q_0\}$	$\{\langle 3, q_0\rangle\}$
$\langle 5, q_0\rangle$	$\emptyset$	$\{\langle 4, q_0\rangle\}$
$\langle 6, q_0\rangle$	$\{q_0\}$	$\{\langle 3, q_0\rangle\}$
$\langle 7, q_0\rangle$	$\{q_0\}$	$\{\langle 5, q_0\rangle, \langle 6, q_0\rangle\}$
$\langle 3, q_2\rangle$	$\{q_2\}$	$\emptyset$
$\langle 4, q_2\rangle$	$\{q_2\}$	$\{\langle 3, q_2\rangle\}$
$\langle 5, q_2\rangle$	$\{q_0\}$	$\{\langle 4, q_2\rangle\}$
$\langle 6, q_2\rangle$	$\emptyset$	$\{\langle 3, q_2\rangle\}$
$\langle 7, q_2\rangle$	$\{q_0\}$	$\{\langle 5, q_2\rangle, \langle 7, q_0\rangle, \langle 6, q_2\rangle\}$

Figure 2. The collecting semantics.

Example 5. Consider again the program from Figure 1 consisting of the procedures main and p. Assume that the abstract domain is given by some complete lattice $\mathbb{D}$ where the functions $h_1^\sharp, h_2^\sharp : \mathbb{D} \to \mathbb{D}$ represent the semantics of basic computation steps. The abstract semantics for the program provides an abstract state in $\mathbb{D}$ for each pair $\langle u, a\rangle$ (u program point from $\{0, \ldots, 7\}$, a possible abstract calling context from $\mathbb{D}$). The right-hand sides $F^\sharp_{\langle u,a\rangle}$ then are given by:

$$
\begin{aligned}
F^\sharp_{\langle 0,a\rangle}\,\sigma &= (a, \emptyset)\\
F^\sharp_{\langle 1,a\rangle}\,\sigma &= (h^\sharp{}_1\,(\sigma\,\langle 0, a\rangle),\ \{\ \langle 0, a\rangle\ \})\\
F^\sharp_{\langle 2,a\rangle}\,\sigma &= (\mathsf{combine}^\sharp\,(\sigma\,\langle 1, a\rangle)\,(\sigma\,\langle 7, \sigma\,\langle 1, a\rangle\rangle),\\
&\qquad \{\ \langle 1, a\rangle, \langle 7, \sigma\,\langle 1, a\rangle\rangle\ \})\\
F^\sharp_{\langle 3,a\rangle}\,\sigma &= (a, \emptyset)\\
F^\sharp_{\langle 4,a\rangle}\,\sigma &= (\sigma\,\langle 3, a\rangle,\ \{\ \langle 3, a\rangle\ \})\\
F^\sharp_{\langle 5,a\rangle}\,\sigma &= (h^\sharp{}_1\,(\sigma\,\langle 4, a\rangle), \{\ \langle 4, a\rangle\ \})\\
F^\sharp_{\langle 6,a\rangle}\,\sigma &= (\sigma\ \langle 3, a\rangle,\ \{\ \langle 3, a\rangle\ \})\\
F^\sharp_{\langle 7,a\rangle}\,\sigma &= (\mathsf{combine}^\sharp\,(\sigma\,\langle 5, a\rangle)\,(\sigma\,\langle 7, \sigma\,\langle 5, a\rangle\rangle) \sqcup\ h^\sharp{}_2\,(\sigma\,\langle 6, a\rangle),\\
&\qquad \{\ \langle 5, a\rangle, \langle 7, \sigma\,\langle 5, a\rangle\rangle, \langle 6, a\rangle\ \})
\end{aligned}
$$

Corresponding to the function combine required by the collecting semantics, the auxiliary function $\mathsf{combine}^\sharp : \mathbb{D} \to \mathbb{D} \to \mathbb{D}$ determines the abstract program state after a call from the abstract state before the call and the abstract state at the end of the procedure body. □

Right-hand functions in practically given abstract systems of equations, however, usually do not *explicitly* provide a set of unknowns onto which the evaluation depends. Instead, the latter set is given *implicitly* via the implementation of the function computing the value for the left-hand side unknown. If necessary, this set must be determined by the solver, for example, as in case of **TD**, by keeping track of the unknowns *accessed* during the evaluation of the function.

Evaluation of the right-hand side function during solving may thus affect the *internal* state of the solver. Such operational behavior can conveniently be made explicit by means of *state transformer monads*. For a set S of (solver) states, the state transformer monad $\mathcal{M}_S(A)$ for values

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

of type A consists of all functions $S \to S \times A$. As a special case of a *monad*, the state transformer monad $\mathcal{M}_S(A)$ provides functions $\mathsf{return} : A \to \mathcal{M}_S(A)$ and $\mathsf{bind} : \mathcal{M}_S(A) \to (A \to \mathcal{M}_S(B)) \to \mathcal{M}_S(B)$. These are defined by

$$\begin{aligned}\mathsf{return}\ a &= \mathbf{fun}\ s \to (s, a)\\ \mathsf{bind}\ m\ f &= \mathbf{fun}\ s \to\ \mathbf{let}\ (s', a) = m\ s\\ &\qquad\qquad\ \mathbf{in}\ f\ a\ s'\end{aligned}$$

The solvers we consider only take actions when the current values of unknowns are accessed during the evaluation of right-hand sides. In the monadic formulation, the right-hand side functions $f^\sharp_x, x \in \mathcal{X}$ of the abstract system of equations $\mathcal{E}^\sharp$ therefore are of type $(\mathcal{X} \to \mathcal{M}(\mathbb{D})) \to \mathcal{M}(\mathbb{D})$ for *any* monad $\mathcal{M}$, that is, are *parametric* in $\mathcal{M}$ (the system of equations should be ignorant of the internals of the solver!). Such functions $f^\sharp{}_x$ have been called *pure* in Karbyshev (2013).

Example 6. For $\sigma^\sharp : \mathcal{X} \to \mathcal{M}_S(\mathbb{D})$, the right-hand side functions in the monadic formulation of the abstract system of equations for the program from Figure 1 now are given by

$$\begin{aligned}f^\sharp_{\langle 0,a\rangle}\ \sigma^\sharp &= \mathsf{return}\ a\\ f^\sharp_{\langle 1,a\rangle}\ \sigma &= \mathsf{bind}\ (\sigma\ \langle 0, a\rangle)\ (\mathbf{fun}\ b \to\\ &\qquad \mathsf{return}\ (h^\sharp_1\ b))\\ f^\sharp_{\langle 2,a\rangle}\ \sigma &= \mathsf{bind}\ (\sigma\ \langle 1, a\rangle)\ (\mathbf{fun}\ b_1 \to\\ &\qquad \mathsf{bind}\ (\sigma\ \langle 7, b_1\rangle)\ (\mathbf{fun} b_2 \to\\ &\qquad \mathsf{return}\ (\mathsf{combine}^\sharp\ b_1\ b_2)))\\ f^\sharp_{\langle 3,a\rangle}\ \sigma &= \mathsf{return}\ a\\ f^\sharp_{\langle 4,a\rangle}\ \sigma &= \sigma\ \langle 3, a\rangle\\ f^\sharp_{\langle 5,a\rangle}\ \sigma &= \mathsf{bind}\ (\sigma\ \langle 4, a\rangle)\ (\mathbf{fun}\ b \to\\ &\qquad \mathsf{return}\ (h^\sharp_1\ b))\\ f^\sharp_{\langle 6,a\rangle}\ \sigma &= \sigma\ \langle 3, a\rangle\\ f^\sharp_{\langle 7,a\rangle}\ \sigma &= \mathsf{bind}\ (\sigma\ \langle 5, a\rangle)\ (\mathbf{fun}\ b_1 \to\\ &\qquad \mathsf{bind}\ (\sigma\ \langle 7, b_1\rangle)\ (\mathbf{fun}\ b_2 \to\\ &\qquad \mathsf{bind}\ (\sigma\ \langle 6, a\rangle)\ (\mathbf{fun}\ b_3 \to\\ &\qquad \mathsf{return}\ (\mathsf{combine}^\sharp\ b_1\ b2\ \sqcup\ h^\sharp_2\ b_3))))\end{aligned}$$

□

According to the considerations in Karbyshev (2013), each pure function f of type $(\mathcal{X} \to \mathcal{M}(\mathbb{D})) \to \mathcal{M}(\mathbb{D})$ equals the semantics $[\![t]\!]$ of some *computation tree* t. Computation trees make explicit in which order the values of unknowns are queried when computing the result values of a function. The set of all computation trees (over the unknowns $\mathcal{X}^\sharp$ and the set of values $\mathbb{D}$) is the least set $\mathcal{T}$ with

$$\mathcal{T} \quad ::= \quad \mathsf{A}\,\mathbb{D} \mid \mathsf{Q}\,(\mathcal{X}^\sharp, \mathbb{D} \to \mathcal{T})$$

The computation tree $\mathsf{A}\,d$ immediately returns the answer d, while the computation tree $\mathsf{Q}\,(x, f)$ queries the value of the unknown x in order to apply the continuation f to the obtained value.

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

Example 7. Consider again the program from Figure 1 consisting of the procedures main and p and the abstract system of equations as provided in Example 6. The computation trees $t_{\langle u,a\rangle}$ for the right-hand side functions $f^\sharp_{\langle u,a\rangle}$ then are given by:

$$\begin{aligned}
t_{\langle 0,a\rangle} &= \mathsf{A}\, a\\
t_{\langle 1,a\rangle} &= \mathsf{Q}\,(\langle 0,a\rangle, \mathbf{fun}\, b \to \mathsf{A}\,(h^\sharp{}_1\, b))\\
t_{\langle 2,a\rangle} &= \mathsf{Q}\,(\langle 1,a\rangle, \mathbf{fun}\, b \to\\
&\qquad \mathsf{Q}\,(\langle 7,b\rangle, \mathbf{fun}\, b' \to\\
&\qquad \mathsf{A}\,(\mathsf{combine}^\sharp\, b\, b')))\\
t_{\langle 3,a\rangle} &= \mathsf{A}\, a\\
t_{\langle 4,a\rangle} &= \mathsf{Q}\,(\langle 3,a\rangle, \mathbf{fun}\, b \to \mathsf{A}\, b)\\
t_{\langle 5,a\rangle} &= \mathsf{Q}\,(\langle 4,a\rangle, \mathbf{fun}\, b \to \mathsf{A}\,(h^\sharp{}_1\, b))\\
t_{\langle 6,a\rangle} &= \mathsf{Q}\,(\langle 3,a\rangle, \mathbf{fun}\, b \to \mathsf{A}\, b)\\
t_{\langle 7,a\rangle} &= \mathsf{Q}\,(\langle 5,a\rangle, \mathbf{fun}\, b \to\\
&\qquad \mathsf{Q}\,(\langle 7,b\rangle, \mathbf{fun}\, b' \to\\
&\qquad \mathsf{Q}\,(\langle 6,a\rangle, \mathbf{fun}\, b'' \to\\
&\qquad \mathsf{A}\,(\mathsf{combine}^\sharp\, b\, b' \sqcup h^\sharp{}_2\, b'))))
\end{aligned}$$

□

The *semantics* of a computation tree t is a function $[\![t]\!] : (\mathcal{X}^\sharp \to \mathcal{M}(\mathbb{D})) \to \mathcal{M}(\mathbb{D})$ where for $\mathsf{get} : \mathcal{X} \to \mathcal{M}(\mathbb{D})$,

$$\begin{aligned}
[\![\mathsf{A}\, d]\!]\, \mathsf{get} &= \mathsf{return}\, d\\
[\![\mathsf{Q}\,(x,f)]\!]\, \mathsf{get} &= \mathsf{bind}\,(\mathsf{get}\, x)\,(\mathsf{fun}\, d \to [\![f\, d]\!]\, \mathsf{get})
\end{aligned}$$

In the particular case that $\mathcal{M}$ is the state transformer monad for a set of states S, we have:

$$\begin{aligned}
[\![\mathsf{A}\, d]\!]\, \mathsf{get}\, s &= (s, d)\\
[\![\mathsf{Q}\,(x,f)]\!]\, \mathsf{get}\, s &= \mathbf{let}\,(s', d) = \mathsf{get}\, x\, s\\
&\qquad \mathbf{in}\, [\![f\, d]\!]\, \mathsf{get}\, s'
\end{aligned}$$

When reasoning about (partial post-)*solutions* of abstract systems of equations, we prefer to have right-hand side functions where (a superset of) the set of accessed unknowns is explicit, as we used for *concrete* systems of equations. These functions, however, can be recovered from right-hand side functions in monadic form, as we indicate now.

One instance of state transformer monads is a monad which tracks the variables accessed during the evaluation. Consider the set of states $S = (\mathcal{X}^\sharp \to \mathbb{D}) \times \mathcal{P}(\mathcal{X}^\sharp)$ together with the function

$$\begin{aligned}
\mathsf{get}\, x\, (\sigma, X) &= \mathbf{let}\, d = \sigma\, x\\
&\qquad \mathbf{in}\, ((\sigma, X \cup \{x\}), d)
\end{aligned}$$

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

Proposition 2. *For a mapping* $\sigma : \mathcal{X}^\sharp \to \mathbb{D}$ *and* $s = (\sigma, \emptyset)$, *assume that* $[\![t]\!]$ *get* $s = (s_1, d)$. *Then for* $s_1 = (\sigma_1, X)$ *the following holds:*

1. $\sigma = \sigma_1$;
2. *Assume that* $\sigma' : \mathcal{X}^\sharp \to \mathbb{D}$ *is another mapping and* $s' = (\sigma', \emptyset)$. *Let* $[\![t]\!]$ *get* $s' = ((\sigma', X'), d')$. *If* σ' *agrees with* σ *on* X, *that is,* $\sigma|_X = \sigma'|_X$, *then* $X' = X$ *and* $d' = d$ *holds.*

We strengthen the statement by claiming that the conclusions also hold when s and s' are given by (σ, X_0) and (σ', X_0), respectively, for the same set X_0. Then, the proof is by induction on the structure of t.

Now assume that for each abstract unknown $x \in \mathcal{X}^\sharp$, the system $\mathcal{E}^\sharp$ provides us with a right-hand side function $f_x^\sharp : (\mathcal{X} \to \mathcal{M}(\mathbb{D})) \to \mathcal{M}(\mathbb{D})$. Then, the *elaborated* abstract right-hand side function $F_x^\sharp : (\mathcal{X}^\sharp \to \mathbb{D}) \to \mathbb{D} \times \mathcal{P}(\mathcal{X}^\sharp)$ of x is given by:

$$F_x^\sharp\, \sigma = \textbf{let}\ ((\_, X), d) = f_x^\sharp\ \text{get}\ (\sigma, \emptyset)\ \textbf{in}\ (d, X)$$

In fact, the explicit right-hand side functions $F^\sharp_{\langle u,a\rangle}$ of Example 5 are obtained in this way from the functions $f^\sharp_{\langle u,a\rangle}$ of Example 6.

In order to relate the abstract with a corresponding concrete system of equations, we assume that there is a *Galois connection* (Cousot and Cousot 1977) between the complete lattices $\mathcal{P}(\mathcal{Q})$ and $\mathbb{D}$, that is, monotonic mappings $\alpha : \mathcal{P}(\mathcal{Q}) \to \mathbb{D}$ (the *abstraction*) and $\gamma : \mathbb{D} \to \mathcal{P}(\mathcal{Q})$ (the *concretization*) so that

$$\alpha\, Q \sqsubseteq a \qquad \text{iff} \qquad Q \subseteq \gamma\, a$$

holds for all $Q \in \mathcal{P}(\mathcal{Q})$ and $a \in \mathbb{D}$.

In general, the sets of unknowns of the concrete system to be analyzed and the corresponding abstract system need not coincide. For interprocedural context-sensitive analysis, for example, the set of concrete unknowns is given by the set of all pairs $\langle u, q\rangle$ where u is a program point and $q \in \mathcal{Q}$ is a program state. The set of abstract unknowns are of the same form. The second components of pairs, however, now represent *abstract* calling contexts. Therefore, we assume that we are given a *description* relation $\mathcal{R} \subseteq \mathcal{X} \times \mathcal{X}^\sharp$ between the concrete and abstract unknowns. In case of interprocedural analysis, for example, we define $\mathcal{R}$ by

$$\langle u, q\rangle\ \mathcal{R}\ \langle u, a\rangle \quad \text{iff} \quad q \in \gamma(a)$$

Using the concretization γ, the description relation $\mathcal{R}$ on unknowns is extended as follows.

- For sets of unknowns $Y \subseteq \mathcal{X}$ and $Y^\sharp \subseteq \mathcal{X}^\sharp$, $Y\, \mathcal{R}\, Y^\sharp$ iff for each $y \in Y$, $y\, \mathcal{R}\, y^\sharp$ for some $y^\sharp \in Y^\sharp$.
- For sets of states $Q \in \mathcal{P}(\mathcal{Q})$ and $d \in \mathbb{D}$, $Q\, \mathcal{R}\, d$ iff $Q \subseteq \gamma\ d$;
- For partial assignments (σ, dom) and $(\sigma^\sharp, dom^\sharp)$ with $\sigma : \mathcal{X} \to \mathbb{D}$, $\sigma^\sharp : \mathcal{X}^\sharp \to \mathbb{D}^\sharp$ and $dom \subseteq \mathcal{X}$, $dom^\sharp \subseteq \mathcal{X}^\sharp$, $(\sigma, dom)\ \mathcal{R}\ (\sigma^\sharp, dom^\sharp)$ holds if $dom\ \mathcal{R}\ dom^\sharp$, and for all $y \in dom$, $y \in dom^\sharp$ with $y\, \mathcal{R}\, y^\sharp$, $\sigma\ y \subseteq \gamma(\sigma^\sharp\ y^\sharp)$.
- For (elaborated) right-hand sides $F : (\mathcal{X} \to \mathcal{P}(\mathcal{Q})) \to (\mathcal{P}(\mathcal{Q}) \times \mathcal{P}(\mathcal{X}))$ and $F^\sharp : (\mathcal{X}^\sharp \to \mathbb{D}) \to (\mathbb{D} \times \mathcal{P}(\mathcal{X}^\sharp))$, $F\, \mathcal{R}\, F^\sharp$ iff $(F\, \sigma)_1 \subseteq \gamma\ (F^\sharp\, \sigma^\sharp)_1$, and $(F\, \sigma)_2\, \mathcal{R}\, (F^\sharp\, \sigma^\sharp)_2$ whenever $(\sigma, dom)\ \mathcal{R}\ (\sigma^\sharp, dom^\sharp)$ holds for domains $dom, dom^\sharp$ which are $(\sigma, \mathcal{E})$-closed and $(\sigma^\sharp, \mathcal{E}^\sharp)$-closed, respectively.
- For equation systems $\mathcal{E} : \mathcal{X} \to ((\mathcal{X} \to \mathcal{P}(\mathcal{Q})) \to (\mathcal{P}(\mathcal{Q}) \times \mathcal{P}(\mathcal{X})))$ and $\mathcal{E}^\sharp : \mathcal{X}^\sharp \to ((\mathcal{X}^\sharp \to \mathcal{M}_S(\mathbb{D})) \to \mathcal{M}_S(\mathbb{D}))$, $\mathcal{E}\ \mathcal{R}\ \mathcal{E}^\sharp$ iff $(\mathcal{E}\, x)\ \mathcal{R}\ F^\sharp_{x^\sharp}$ for each $x \in \mathcal{X}$, $x^\sharp \in \mathcal{X}^\sharp$, where $F^\sharp_{x^\sharp}$ is the elaboration of $\mathcal{E}^\sharp\, x^\sharp$.

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

Let (σ, dom) be the least solution of the concrete system $\mathcal{E}$ for some set X of interesting unknowns. Let $\mathcal{E}^\sharp$ denote an abstract system corresponding to $\mathcal{E}$ and $X^\sharp$ a set of abstract unknowns and $\mathcal{R}$ a description relation between the unknowns of $\mathcal{E}$ and $\mathcal{E}^\sharp$ such that $\mathcal{E}\ \mathcal{R}\ \mathcal{E}^\sharp$ and $X\ \mathcal{R}\ X^\sharp$ holds. A *local solver* then determines for $\mathcal{E}^\sharp$ and $X^\sharp$ a pair $(\sigma^\sharp, dom^\sharp)$ so that $X \subseteq dom^\sharp$, $dom^\sharp$ is $\sigma^\sharp$-closed and $(\sigma, dom)\ \mathcal{R}\ (\sigma^\sharp, dom^\sharp)$ holds, that is, the result produced by the solver is a *sound* description of the least partial solution of the concrete system.

In absence of narrowing, the correctness of a solver can be proven *intrinsically*, that is, just by verifying that it terminates with a *post*-solution of the system of equations.

We first convince ourselves that the following holds:

Proposition 3. *Assume that $\mathcal{E}\ \mathcal{R}\ \mathcal{E}^\sharp$ holds and $X\ \mathcal{R}\ X^\sharp$ for subsets X and $X^\sharp$ concrete and abstract unknowns, respectively. Assume that (σ, dom) is the least partial post-solution of $\mathcal{E}$ with $X \subseteq dom$, and $(\sigma^\sharp, dom^\sharp)$ some partial post-solution of $\mathcal{E}^\sharp$ with $X^\sharp \subseteq dom^\sharp$. Then $(\sigma, dom)\ \mathcal{R}\ (\sigma^\sharp, dom^\sharp)$ holds.*

Proposition 3 is a special case of Proposition 4 where additionally side effects and dynamic description relations are taken into account. Therefore, the proof of Proposition 3 is omitted. Proposition 3 can be used to prove soundness for local solver algorithms which perform accumulating fixpoint iteration and thus return partial post-solutions. These kinds of solvers require abstract domains where strictly ascending chains are always finite. This assumption, however, is no longer met for more complicated domains such as the interval domain (Cousot and Cousot 1977) or octagons (Mine 2001). As already observed in Cousot and Cousot (1977), their applicability to these domains can still be extended by introducing *widening* operators. According to Cousot and Cousot (1992), Cousot (2015), a widening operator ∇ is a mapping $\nabla : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ with the following two properties:

(1) $a \sqcup b \sqsubseteq a \nabla b$ for all $a, b \in \mathbb{D}$;
(2) Every sequence $a_0, a_1, \ldots$ defined by $a_{i+1} = a_i \nabla b_i$, $i \geq 0$ for any $b_i \in \mathbb{D}$ is ultimately stable.

Acceleration with widening then means that the occurrences of $\sqcup$ in the solver are replaced with ∇.

Example 8. For intervals over $\mathbb{Z}_{-\infty}^{+\infty}$ we could use primitive widening:

$$[a_1, b_1] \nabla [a_2, b_2] = [\textbf{if } a_2 < a_1 \textbf{ then } -\infty \textbf{ else } a_1,\ \textbf{if } b_2 > b_1 \textbf{ then } +\infty \textbf{ else } b_1]$$

Alternatively, we could use threshold widening where several intermediate bounds are introduced that can be jumped to. Note that widening in general is not monotonic in the first argument: $[0, 1] \sqsubseteq [0, 2]$ but $[0, 1] \nabla [0, 2] = [0, +\infty] \not\sqsubseteq [0, 2] = [0, 2] \nabla [0, 2]$. □

We remark that Cousot and Cousot (1992), Cousot (2015) provide a more general notion of widening which refers not to the ordering of $\mathbb{D}$ but (via γ) to the ordering in the concrete lattice $\mathcal{P}(\mathcal{Q})$ alone. W.r.t. that definition, $a \sqcup b$ is no longer necessarily less or equal $a \nabla b$. In many applications, however, accelerated loss of precision due to widening may result in unacceptable analysis results. Therefore, Cousot and Cousot (1977) proposed to complement a terminating widening iteration with a *narrowing* iteration which subsequently tries to recover some of the precision loss. Following Cousot and Cousot (1992), Cousot (2015), a narrowing operator Δ is a mapping $\Delta : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ with the following two properties:

(1) $a \sqcap b \sqsubseteq (a \Delta b) \sqsubseteq a$ for all $a, b \in \mathbb{D}$;
(2) Every sequence $a_0, a_1, \ldots$ defined by $a_{i+1} = a_i \Delta b_i$, $i \geq 0$ for any $b_i \in \mathbb{D}$ is ultimately stable.

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

Example 9. For intervals over $\mathbb{Z}^{+\infty}_{-\infty}$ we could use primitive narrowing:

$$[a_1, b_1] \,\Delta\, [a_2, b_2] = [\textbf{if } a_1 = -\infty \textbf{ then } a_2 \textbf{ else } a_1,\\ \textbf{if } b_1 = +\infty \textbf{ then } b_2 \textbf{ else } b_1]$$

which improves infinite bounds only. More sophisticated narrowing operators may allow a bounded number of improvements of finite bounds as well. □

Again we remark that, according to the more general definition in Cousot and Cousot (1992), Cousot (2015), the first property need not necessarily be satisfied. For monotonic systems of equations, the narrowing iteration when starting with a (partial) post-solution still will return a (partial) post-solution. The correctness of the solver started with an initial query $x^\sharp$ and returning a partial assignment $\sigma^\sharp$ with domain $dom^\sharp$ thus can readily be checked by verifying that

1. $x^\sharp \in dom^\sharp$;
2. $(F^\sharp_x\, \sigma^\sharp)_2 \subseteq dom^\sharp$ for all $x \in dom^\sharp$, that is, $dom^\sharp$ is $(\sigma^\sharp, \mathcal{E}^\sharp)$-closed; and
3. $\sigma^\sharp\, x \sqsupseteq (F^\sharp_x\, \sigma^\sharp)_1$ for all $x \in dom^\sharp$.

When the system of equations is non-monotonic, though, the computed assignment still is a sound description. It is, however, no longer guaranteed to be a post-solution of $\mathcal{E}^\sharp$. In Section 6, we come back to this point.

4. The Terminating Solver $\mathbf{TD}_{\mathbf{term}}$

In this section, we present our modification to the **TD** solver with widening and narrowing which improves on the variant in Apinis et al. (2016) in that termination guarantees can be proven even for *non-monotonic* abstract systems of equations. The vanilla **TD** solver from Muthukumar and Hermenegildo (1990), Charlier and Van Hentenryck (1992) (see Appendix A for a pseudo code formulation of this solver along the lines presented in Fecht and Seidl 1999) starts by querying the value of a given unknown. In order to answer the query, the solver evaluates the corresponding right-hand side. Whenever in the course of that evaluation, the value of another unknown is required, the best possible value for that unknown is computed first, before evaluation of the current right-hand side continues. Interestingly, the strategy employed by **TD** for choosing the next unknown to iterate upon, thereby resembles the iteration orders considered in Bourdoncle (1993) (see Fecht and Seidl 1999 for a detailed comparison) for systems of equations derived from control-flow graphs of programs. The most remarkable difference, however, is that **TD** determines its order on-the-fly, while the ordering in Bourdoncle (1993) is determined via preprocessing.

In Apinis et al. (2016), the vanilla **TD** solver from Muthukumar and Hermenegildo (1990), Charlier and Van Hentenryck (1992), Fecht and Seidl (1999) is enhanced with widening and narrowing. For that, the solver is equipped with a novel technique for identifying not only accesses to unknowns, but also widening and narrowing points on-the-fly. Moreover, that solver does not delegate the narrowing iteration to a separate second phase (as was done in the original papers on widening and narrowing Cousot 2015; Cousot and Cousot 1992), once a proceeding widening iteration has completed. Instead, widening and narrowing iterations may occur *intertwined* (Amato et al. 2016). This is achieved by combining the widening operator ∇ with the narrowing operator Δ into a single *warrowing* operator $\boxtriangledown$:

$$a \boxtriangledown b = \textbf{if } b \sqsubseteq a \textbf{ then } a \,\Delta\, b \\ \textbf{else } a \,\nabla\, b$$

This operator applies Δ whenever values decrease and otherwise applies ∇.

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

In Apinis et al. (2016), it is proven that solver **TD** (in the formulation of Fecht and Seidl 1999) and equipped with warrowing at dynamically detected widening/narrowing points terminates for *monotonic* systems – whenever only finitely many unknowns are encountered. Example 10, though, shows a non-monotonic system for which this solver does not terminate – while the new solver $\mathbf{TD}_{\mathbf{term}}$ does.

Example 10. Consider the single equation:

$$x = \text{if } x = 0 \text{ then } 1 \text{ else } 0$$

over the lattice of naturals (with infinity) with $a \nabla b = \infty$ whenever $a < b$ and $a \Delta b = b$ whenever $a = \infty$. The right-hand side of this equation is not monotonic. An iteration with warrowing leads to the sequence of values for x

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

and thus will not terminate. □

In order to deal with *non-monotonic* systems, we do no longer rely on *warrowing*. Instead, we equip the solver with extra logic to switch for each unknown from widening to narrowing (and never back). Our new solver is presented as OCaml pseudocode operating on abstract systems of equations. W.l.o.g., we also assume that solving starts with a *single* unknown of interest. In case that simultaneously values for unknowns from an arbitrary finite set X are of interest, we may introduce an *artificial* fresh unknown x_0 whose right-hand side successively queries the values of $x \in X$.

For better readability, the solver state is not threaded through the evaluation of right-hand sides by means of a monad, but realized by mutable data structures:

```
(* state / mutable data structures: *)
val σ      : (𝒳, 𝔻) Map.t
val infl   : (𝒳, 𝒳 Set.t) Map.t
val called : 𝒳 Set.t
val stable : 𝒳 Set.t
val point  : 𝒳 Set.t
(* creation *)
val Set.create : unit -> 'a Set.t
val Map.create : (unit -> 'b) -> ('a,'b) Map.t
(* unary operator *)
val (!) : ('a,'b) Map.t -> 'a -> 'b
(* binary operators *)
val (:=) : ('a,'b) Map.t * 'a -> 'b -> unit
val (∈ ) : 'a -> 'a Set.t -> bool
val (+=) : 'a Set.t -> 'a -> unit
val (-=) : 'a Set.t -> 'a -> unit
```

Here, the functional argument f to the function Map.create is meant to return an initial value f () for a key which has not yet been assigned in the given map data structure.

Under that proviso, the OCaml type of a right-hand side function $f_x^\sharp$ thus is just $(\mathcal{X} \to \mathbb{D}) \to \mathbb{D}$. When reasoning about the values computed by right-hand sides as well as the sets of unknowns accessed during the evaluation, we will refer to the *elaborated* right-hand sides $F_x^\sharp$ corresponding to the $f_x^\sharp$. The solver itself consists of three functions: the function solve which performs the iteration for a given unknown x; the function eval which wraps lookups of values of unknowns in the current state; and finally, the function destabilize which marks encountered unknowns possibly affected by a change to the value of some unknown, for re-evaluation.

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

```
let stable = Set.create ()
let called = Set.create ()
let point  = Set.create ()
let infl   = Map.create Set.create
let σ      = Map.create (fun () -> ⊥)

let rec destabilize x =
  let w = !infl x in
  infl, x := Set.create ();
  Set.iter (fun y ->
      stable -= y;
      if y ∉ called then
        destabilize y
    ) w

let rec eval x y =
  if y ∈ called then
    point += y
  else
    solve ∇ y;
  !infl y += x;
  !σ y

and solve p x =
  if x ∉ stable && x ∉ called then (
    stable += x;
    called += x;
    let tmp = f♯x (eval x) in
    called -= x;
    let tmp =
      if x ∈ point then p (!σ x) tmp
      else tmp
    in
    if !σ x = tmp then
      if p = ∇ && x ∈ point then (
        stable -= x;
        solve Δ x
      ) else ()
    else (
      σ, x := tmp;
      destabilize x;
      solve p x
    )
  )

let solve x = solve ∇ x; (σ, stable)
```

We briefly sketch the intuition behind this algorithm. The set *called* is the set of all unknowns where iteration currently is in progress, that is, which are contained in the call stack of the solver. The set *stable* on the other hand consists of all unknowns where the iteration (relative to the current values of unknowns on the call stack) has terminated. For unknowns in any of these two sets, solving should immediately terminate.

A key issue is to propagate the information that the value of some unknown y has changed, to all unknowns whose right-hand sides have been evaluated under wrong assumptions on y. A key

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

ingredient both of the original **TD** solver and the variant in Apinis et al. (2016) is that *dependences* between unknowns are dynamically detected and recorded in the map $infl : \mathcal{X}^\sharp \to \mathcal{P}(\mathcal{X}^\sharp)$. The set $infl\ y$ is meant to record the set of unknowns x where the evaluation of ${f^\sharp}_x$ has accessed the current value of y, that is, where y *influences* x.

As soon as the value of an unknown y is changed, therefore, the function destabilize is called. destabilize y removes all unknowns directly or indirectly influenced by y from the set *stable*. That recursive removal only stops at unknowns which are contained in the set *called*. We remark that by the call destabilize y, also the set $infl\ y$ is reset to $\emptyset$.

The current values for unknowns are maintained in the map σ. In order to track dependences between unknowns, we wrap the access to entries in σ into the call eval x where x is the unknown in whose right-hand side the values in σ are queried. Thus, a call eval $x\ y$ ultimately returns the latest value in σ for y. Before that, however, it checks whether $y \in called$ holds. If this is the case, y is turned into an unknown where widening and narrowing is applied. All these unknowns are collected into the set *point*. If this is not the case, the call solve $\nabla\ y$ is evaluated to determine the best possible value for y before-hand. Then, x is added to the set $infl\ y$ of unknowns influenced by y, and finally, the value for y in σ is returned.

The main ingredient of the algorithm, though, is the function solve. The function solve ∇ is applied to an unknown x to determine the best possible value for x. In case that $x \in point$, the computation starts with a local widening iteration on x which is followed by a call solve $\Delta\ x$ to subsequently perform a local narrowing iteration on x. Any call solve $p\ x$ immediately terminates if x is already found to be in $called \cup stable$. If this is not the case, x is added to *stable* and *called*, and the right-hand side ${f^\sharp}_x$ of x is evaluated for the argument eval x, and the first component of the result stored in the temporary *tmp*. After the evaluation, x is removed from *called*. If x has been found to be in *point*, we use p to combine the old value for x as stored in σ with the new value in *tmp*. Otherwise, we use *tmp* directly.

Assume that the new value is the same as the old value for x as provided by σ. If $p = \nabla$, then the widening phase is completed. Therefore, x is removed from *stable* and *solve* $\Delta\ x$ is called. Otherwise, we are done.

Now assume that the new value is different from the old value for x. Then, we update the value of σ for x to the new value, call destabilize x in order to propagate this information, and recursively call solve $p\ x$.

5. Termination of $\mathbf{TD_{term}}$

Each state s attained by the solver during its evaluation, consists of the tuple of mutable data structures $s = (\sigma, infl, stable, called, point)$. We call s *consistent* if

1. for all $x \in (stable \setminus called)$, and all $y \in ({F^\sharp}_x\sigma)_2$, $y \in (stable \cup called)$ and $x \in infl\ y$ holds; and
2. for each $x \notin (stable \cup called)$, $infl\ x = \emptyset$.

Each call solve $p\ x$ encountered during the evaluation of the initial call, starts in a consistent solver state s. Upon its termination, a solver state $s' = (\sigma', infl', stable', called', point')$ is attained such that $s = s'$ whenever $x \in stable \cup called$. Otherwise, it holds that

1. s' is again consistent;
2. $called' = called$ and $stable \subseteq stable'$ where for all $y \in stable \cup called$, $\sigma'\ y = \sigma\ y$ and $infl\ y \subseteq infl'\ y$;
3. $point \subseteq point'$;
4. $x \in stable'$.

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

Also, each call eval $x\ y$ encountered during the evaluation starts in a consistent state s where $x \in (\mathit{stable} \cap \mathit{called})$. Upon its termination, a solver state s' is attained such that

1. s' is again consistent;
2. $\mathit{called}' = \mathit{called}$;
3. $\mathit{stable} \subseteq \mathit{stable}'$ where for all $y' \in \mathit{stable} \cup \mathit{called}$, $\sigma'\ y' = \sigma\ y'$ and $\mathit{infl}\ y' \subseteq \mathit{infl}'\ y'$;
4. $\mathit{point} \subseteq \mathit{point}'$;
5. If $y \in \mathit{stable} \cup \mathit{called}$, then
   - $\sigma' = \sigma$, $\mathit{stable}' = \mathit{stable}$ and $\mathit{point}' = \mathit{point}$ if $y \notin \mathit{called}$ whereas $\mathit{point}' = \mathit{point} \cup \{y\}$ if $y \notin \mathit{called}$; ;
   - $\mathit{infl}'\ y' = \mathit{infl}\ y'$ for all $y' \neq y$, and
   - $\mathit{infl}'\ y = \mathit{infl}\ y \cup \{x\}$;
6. $y \in \mathit{stable}'$, $x \in \mathit{infl}'\ y$, and the value $\sigma'\ y$ is returned.

In particular, x is still contained in $\mathit{stable}' \cap \mathit{called}'$. Finally, consider the call to destabilize x in the body of solve, and let $s = (\sigma, \mathit{infl}, \mathit{stable}, \mathit{called}, \mathit{point})$ before this call. Then, s is consistent with $x \in \mathit{stable} \backslash \mathit{called}$, where upon termination of the call, a solver state s' is attained so that

1. s' is again consistent;
2. $\sigma' = \sigma$, $\mathit{called} = \mathit{called}'$, $\mathit{point} = \mathit{point}'$;
3. $\mathit{stable}' \subseteq \mathit{stable}$, and
4. for all y, $\mathit{infl}'\ y$ either equals $\emptyset$ or $\mathit{infl}\ y$ where infl' and stable' are maximal so that s' is consistent while $\mathit{infl}'\ x = \emptyset$ and $\mathit{infl}\ x \cap \mathit{stable}' = \emptyset$.

This invariant allows us to prove that the solver $\mathbf{TD}_{\mathbf{term}}$ indeed terminates for arbitrary systems of abstract equations.

Theorem 1. *Let $\mathcal{E}^\sharp$ denote an arbitrary system of abstract equations, and let x_0 be the initial unknown of interest. Assume that initially the sets called and stable are empty, and likewise, infl maps each unknown to the empty set. Then, the call* solve $\nabla\ x_0$ *will always terminate, as long as only finitely many unknowns are encountered.*

Proof. First we note that every call destabilize y terminates. This is immediate in case when $\mathit{infl}\ y$ is empty. Moreover, $\mathit{infl}\ y'$ is non-empty for only finitely many unknowns y'. Since the set $\mathit{infl}\ y$ is set to $\emptyset$, before any recursive call to unknowns, termination follows.

Assume now that during evaluation of the initial call solve $\nabla\ x_0$, only finitely many unknowns x are encountered for which solve $p\ x$ is called for some p. In order to prove termination of all calls solve $p\ x$ encountered during the evaluation of solve $\nabla\ x_0$, we perform an induction on the cardinality of the set of unknowns which are *not* in *called*. Let Y denote the set of all unknowns $y \in \mathcal{X} \setminus (\{x\} \cup \mathit{called} \cup \mathit{stable})$ for which solve $\nabla\ y$ is called during the call solve $p\ x$ and proceeds to the evaluation of the right-hand side of y.

Assume for a contradiction that the call solve $p\ x$ would not terminate. Let us first assume that $x \notin \mathit{point}$ throughout the iteration, that is, all tail-recursive calls to solve $p\ x$. In particular, this means that for none of the unknowns $y \in Y$, evaluation of the right-hand side ${f^\sharp}_y$ accessed the unknown x – as in this case, x would have been added to *point*. Therefore, when $p = \nabla$, $\mathit{infl}\ x$ does not contain any of the unknowns in Y, that is, is still empty after the first update of $\sigma\ x$. Accordingly, destabilization of x will immediately terminate and leave x in the set *stable*. If $p = \Delta$, $\mathit{infl}\ x = \emptyset$ at least for all calls after the first call to solve $\Delta\ x$. As a consequence, the call solve $p\ x$ terminates for each phase p – in contradiction to our assumption.

Therefore, necessarily, $x \in \mathit{point}$ at least after the first evaluation of ${f^\sharp}_x$. We distinguish two cases.

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

Case 1. $p = \Delta$.

By inductive hypothesis, all occurring calls solve $p\ y$ with $y \in Y$ terminate. Let $a_i, i \geq 0$ denote the sequence of values of $\sigma\ x$ before the ith tail-recursive call to solve $\Delta\ x$. Then necessarily $a_i \neq a_{i+1}$ for all $i \geq 0$. Let $b_1, \ldots, b_i, \ldots$ denote the sequence of values returned by the ith evaluation of the right-hand side ${f^\sharp}_x$ of x. Then, $a_{i+1} = a_i \,\Delta\, b_{i+1}$ for $i \geq 0$. Due to the properties of narrowing, however, the latter sequence is ultimately stable, that is, $a_i = a_{i+1}$ for some i – in which case the recursion terminates: contradiction. Therefore, each call solve $\Delta\ x$ eventually terminates.

Case 2. $p = \nabla$.

By inductive hypothesis, again all calls solve $\nabla\ y$ with $y \in Y$ terminate. Let $a_i, i \geq 0$ denote the sequence of values of $\sigma\ x$ before the ith tail-recursive call to solve $\nabla\ x$. Assume that $a_i \neq a_{i+1}$ for all $i \geq 0$. Let $b_1, \ldots$ denote the sequence of values returned by the ith evaluation of the right-hand side ${f^\sharp}_x$ of x. Then, $a_{i+1} = a_i \,\nabla\, b_{i+1}$ for $i \geq 0$. Due to the properties of widening, however, the latter sequence is ultimately stable, that is, $a_i = a_{i+1}$ for some i. Therefore eventually, solve $\Delta\ x$ is tail-recursively called. But then, termination follows due to termination of the call solve $\Delta\ x$: contradiction!

Accordingly, we conclude that all encountered calls solve $p\ x$ terminate, and thus also the call solve $\nabla\ x_0$. □

6. Correctness of $\mathsf{TD}_{\mathsf{term}}$

Our goal is to prove that the result of our algorithm is a sound description of the least partial solution of the concrete system. In case of monotonic abstract systems and total solutions, soundness is known to hold for any *post*-solution of the abstract system, whenever the concrete system is described by the abstract system. Recall that while the right-hand sides of our concrete system are monotonic, this need not necessarily be the case for the abstract system. Consider, for example, an interprocedural analysis as in Examples 5 and 6 and there the elaborated right-hand side

$$F^\sharp_{\langle 2,a\rangle} = \mathbf{fun}\,\sigma \to (\mathsf{combine}^\sharp\ (\sigma\,\langle 1,a\rangle)\ (\sigma\,\langle 7, \sigma\,\langle 1,a\rangle\rangle),\\ \{\ \langle 1,a\rangle, \langle 7,\sigma\,\langle 1,a\rangle\rangle\ \}\ )$$

for the unknown $\langle 2,a\rangle$. Since for different values in σ for $\langle 1,a\rangle$, different unknowns are queried, this function cannot be monotonic. In order to deal with such non-monotonicities, Frielinghaus et al. (2016) have introduced the concept of a *lower monotonization* of the abstract system $\mathcal{E}^\sharp$.

For the system $\mathcal{E}^\sharp$ with set of unknowns $\mathcal{X}^\sharp$, the lower monotonization $\underline{\mathcal{E}}^\sharp$ is a system of equations with the same set of unknowns where the elaborated right-hand side $\underline{F}^\sharp_x$ for $x \in \mathcal{X}^\sharp$ is given by $\underline{F}^\sharp_x\,\sigma = (d, X)$ with

$$d = \bigsqcap \{\ (F^\sharp{}_x\,\sigma')_1 \mid \sigma' \sqsupseteq \sigma\ \}\\ X = \mathcal{X}^\sharp$$

Thus, we over-approximate the sets of unknowns influencing the result of a right-hand side by the full set $\mathcal{X}^\sharp$ while d is the greatest lower bound to all first components of results of ${F^\sharp}_x$ for assignments exceeding σ.

Since all right-hand sides of $\underline{\mathcal{E}}^\sharp$ are monotonic, the system has a least total solution $\underline{\sigma}$. Let σ denote the least total solution of some concrete system $\mathcal{E}$ which is described by $\mathcal{E}^\sharp$. In Frielinghaus et al. (2016), it has been proven that then also $\sigma\ \mathcal{R}\ \underline{\sigma}$ holds. Moreover, let *dom* denote the least $(\sigma, \mathcal{E})$-closed subset containing some initial unknown x. Let $x\ \mathcal{R}\ y$ for some unknown $y \in \mathcal{X}^\sharp$. Let $\sigma' : \mathcal{X}^\sharp \to \mathbb{D}$ be some abstract assignment, and $dom^\sharp$ be $(\sigma', \mathcal{E}^\sharp)$-closed with $y \in dom^\sharp$. Assume

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

further that $\underline{\sigma}\, y' \sqsubseteq \sigma'\, y'$ for all $y' \in dom^\sharp$. Then by the results of Frielinghaus et al. (2016), also $dom\, \mathcal{R}\, dom^\sharp$ holds. Accordingly, the pair $(\sigma', dom^\sharp)$ can then be understood as a sound description of the least partial post-solution of the concrete system $\mathcal{E}$ for x.

Therefore, now assume that we are given a set X of unknowns of the concrete system of equations $\mathcal{E}$, together with a an abstract unknown x_0 so that for all $x \in X$, $x\, \mathcal{R}\, x_0$ holds. Assume further that the solver $\mathbf{TD_{term}}$, when started with the call solve $\nabla\ x_0$, returns the abstract assignment $\sigma^\sharp$ where by default, $\sigma^\sharp\, y = \bot$ for all unknowns y which have not been encountered during solving. It therefore suffices to prove:

1. There is a subset $dom^\sharp \subseteq \mathcal{X}^\sharp$ containing x_0 which is $(\sigma^\sharp, \mathcal{E}^\sharp)$-closed.
2. $\underline{\sigma}^\sharp\, y \sqsubseteq \sigma^\sharp\, y$ holds for all $y \in dom^\sharp$.

After evaluation of each call solve $p\, x$, evaluation of the right-hand side of an unknown in *stable* will access only unknowns which are either again in *stable*, or in *called*. As a consequence, the set of all stable unknowns after termination of all calls solve $\nabla\ x_0$ is $(\sigma^\sharp, \mathcal{E}^\sharp)$-closed. Therefore, it remains to verify the second property. Let $dom^\sharp$ denote the subset $stable \subseteq \mathcal{X}^\sharp$ upon termination of the call solve $\nabla\ x_0$. Let $\underline{\sigma} : \mathcal{X}^\sharp \to \mathbb{D}$ denote the least solution of the lower monotonization $\underline{\mathcal{E}^\sharp}$ of $\mathcal{E}^\sharp$. Our goal is to show that $\underline{\sigma}\, x \sqsubseteq \sigma^\sharp\, x$ for all $x \in dom^\sharp$.

Assume that we are given a subset Y of unknowns together with an assignment $\tau : Y \to \mathbb{D}$. Here, we assume Y and τ to represent the set of unknowns which are currently stable (or called) together with their current values. In the following, we use the binary operator $\oplus$ to denote an update of the left argument with the bindings provided by the right argument. Let ${\mathcal{E}^\sharp}_\tau$ denote the system of equations with unknowns from $\mathcal{X}^\sharp \setminus Y$ where the right-hand side $f^\sharp_{\tau,y}$ of y behaves like the right-hand side ${f^\sharp}_y$ of $\mathcal{E}^\sharp$, but looks up the values for encountered unknowns from Y in τ. Technically, this means that

$$ {f^\sharp}_{\tau,y}\, \sigma = {f^\sharp}_y\, (\sigma \oplus (\mathsf{return} \circ \tau)) $$

Accordingly, the elaborated right-hand side ${F^\sharp}_{\tau,y}$ for the unknown y is given by

$$ \begin{aligned} {F^\sharp}_{\tau,y}\, \sigma \quad &= \mathbf{let}\ (d, X) = {F^\sharp}_y\, (\sigma \oplus \tau) \\ &\quad\ \mathbf{in}\ (d, X \setminus Y) \end{aligned} $$

Let $s = (\sigma, infl, stable, called, point)$ denote a consistent solver state. Let $\tau : called \to \mathbb{D}$ denote the restriction of σ to the set *called*. We call s *saturated* if for all $x \in stable \setminus called$, $\underline{\sigma}\, x \sqsubseteq \sigma\, x$ where $(\underline{\sigma}, X^\sharp \setminus called)$ is the least total solution of the lower monotonization of ${\mathcal{E}^\sharp}_\tau$. We claim:

Theorem 2. *Each call* solve $\nabla\ x$ *starting in a saturated solver state, results upon termination, in a solver state* $s' = (\sigma', infl', stable', called', point')$ *which is again saturated where additionally,* $x \in stable'$.

Since before the initial call solve $\nabla\ x_0$ the set *called* is empty, this theorem implies that upon termination, $dom^\sharp$ is $(\sigma', \mathcal{E}^\sharp)$-closed with $x_0 \in dom^\sharp$, and $\underline{\sigma}\, x \sqsubseteq \sigma'\, x$ for all $x \in dom^\sharp$.

Proof. We proceed by induction on the set $\mathcal{X}'$ of unknowns *not* contained in the set *stable* of stable unknowns before the call. For $\mathcal{X}' = \emptyset$, the assertion obviously holds. For the inductive step, first assume that after the evaluation of the right-hand side ${f^\sharp}_x$, x is not included in *point*. This means that no variable in $\mathcal{X}'$ may depend on the unknown x. Assume that the values of x before and after solve are d and d', respectively. Consider the least total solutions $\underline{\sigma}$ and $\underline{\sigma}'$ of the lower monotonizations of the systems ${\mathcal{E}^\sharp}_\tau$ and ${\mathcal{E}^\sharp}_{\tau \oplus \{x \mapsto d'\}}$, respectively, where $\tau : stable \to \mathbb{D}$ records the values of all unknowns from *stable*. Let $Y = ({F^\sharp}_{\tau \oplus \{x \mapsto d\}, x})_2 \setminus stable$ be the set of unknowns accessed during the evaluation of the right-hand side for x (in particular, $x \notin Y$). Then $\mathcal{X}' \setminus \{x\}$ is

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

the disjoint union of subsets $\mathcal{X}'_y, y \in Y$, where $\mathcal{X}'_y$ is the subset of unknowns freshly solved when y is encountered. By applying the inductive hypothesis to the unknowns in y in sequence, we obtain that for all $y \in Y$, $\underline{\sigma}'\, y' \sqsubseteq \sigma\, y'$ for all $y' \in \mathcal{X}_y$. Accordingly, we have for all $y \in \mathcal{X}' \setminus \{x\}$, that

$$(\underline{F}'_y \underline{\sigma}')_1 \sqsubseteq \sigma'\, y$$

Here, $\underline{F}'_y$ denotes the elaborated right-hand side of the lower monotonization of $\mathcal{E}_{\tau \oplus \{x \mapsto d'\}}$ for y. This allows us to deduce for x that

$$\begin{aligned}(\underline{F^\sharp}_{\tau,x}(\underline{\sigma}' \oplus \{x \mapsto d'\}))_1 &\sqsubseteq (\underline{F^\sharp}_{\tau,x}(\sigma \oplus \{x \mapsto d'\}))_1 \\ &\sqsubseteq (F^\sharp_{\tau,x}(\sigma \oplus \{x \mapsto d'\}))_1 \\ &= (F^\sharp_{\tau,x}(\sigma \oplus \{x \mapsto d\}))_1 \\ &= d'\end{aligned}$$

Thereby, we have used that the value of the unknown x is not contained in $(F^\sharp_x(\sigma \oplus \{x \mapsto d'\}))_2$ – implying that it is also not contained in $(F^\sharp_x(\sigma \oplus \{x \mapsto d\}))_2$. Moreover, for $y \in \mathcal{X}'$ different from x,

$$(\underline{F^\sharp}_{\tau,y}\, (\underline{\sigma}' \oplus \{x \mapsto d'\}))_1 \sqsubseteq (\underline{F}'_y\, \underline{\sigma}')_1 \sqsubseteq \underline{\sigma}'\, y$$

Accordingly, $\underline{\sigma}' \oplus \{x \mapsto d'\}$ is a post-solution of the lower monotonization of $\mathcal{E}_\tau$, implying that $\underline{\sigma} \sqsubseteq \sigma'$.

Now assume that after the evaluation of the right-hand side $f^\sharp_x$, $x \in \mathit{point}$ holds. We concentrate on the last tail-recursive call $\mathsf{solve}\ \nabla\ x$ before the call to $\mathsf{solve}\ \Delta\ x$. Let $s_0 = (\sigma_0, \mathit{infl}_0, \mathit{stable}_0, \mathit{called}_0, \mathit{point}_0)$. Let d denote the value of x *before* the evaluation of the right-hand side, and d' the value returned by evaluating the right-hand side. Let $\mathcal{X}'$ denote the set of unknowns accessed during the call which are not stable before. Since subsequently $\mathsf{solve}\ \Delta\ x$ is called, after that the evaluation of $f^\sharp_x$, x is still stable. Therefore, one of the following two situations is encountered after the evaluation of the right-hand side:

1. $d' \sqsubseteq d$, that is, the value d' is subsumed by the current value of x; or
2. subsequent destabilization will not destabilize x.

In the second case, the unknowns from $\mathcal{X}'$ that directly or indirectly influence x cannot depend on x (w.r.t. the current map infl). Thus, a similar argument as for unknowns not in point applies.

Accordingly, it remains to consider the first case. Consider the lower monotonizations of the abstract systems ${\mathcal{E}^\sharp}_\tau$ and ${\mathcal{E}^\sharp}_{\tau \oplus \{x \mapsto d\}}$ with least solutions $\underline{\sigma}$ and $\underline{\sigma}'$, respectively. By inductive hypothesis applied to the unknowns in $\mathcal{X}' \setminus \{x\}$ and $\tau \oplus \{x \mapsto d\}$, we find that $\underline{\sigma}'\, y \sqsubseteq \sigma_0\, y$ for all unknowns $y \in \mathcal{X}' \setminus \{x\}$. This allows us to prove that $\underline{\sigma}' \oplus \{x \mapsto d\}$ is a post-solution of the lower monotonization of ${\mathcal{E}^\sharp}_\tau$. Therefore, $\underline{\sigma}\, y \sqsubseteq (\underline{\sigma}' \oplus \{x \mapsto d\})\, y \sqsubseteq (\sigma_0 \oplus \{x \mapsto d\})\, y$ for all $y \in \mathcal{X}'$ holds, and the claim follows.

Now consider the narrowing iteration performed for x in the subsequent call $\mathsf{solve}\ \Delta\ x$. Let d_0 denote the value after the last call to $\mathsf{solve}\ \nabla\ x$, and $d_1, \ldots, d_k$ denote the values returned by evaluating the right-hand side of x during this iteration and define $d'_0 = d_0$ and for $i > 0$, $d'_i = d'_{i-1}\ \Delta\ d_i$. Let σ_i denote the assignment attained after the ith narrowing step restricted to the unknowns $\mathcal{X}^\sharp \setminus (\mathcal{X}_0 \cup \{x\})$. For $i > 0$, let $\underline{\sigma}_i$ denote the least solution of the lower monotonization of ${\mathcal{E}^\sharp}_{\tau \oplus \{x \mapsto d_{i-1}\}}$. By inductive hypothesis, $\underline{\sigma}_i\, y \sqsubseteq \sigma_i$ for all $y \neq x$ which are stable after the ith iteration. By induction on i, we prove that $\underline{\sigma}\, x \sqsubseteq d_i$ and thus also $\underline{\sigma}\, x \sqsubseteq d'_i$, and $\underline{\sigma}\, y \sqsubseteq \sigma_i\, y$ for every $y \in \mathcal{X}^\sharp \setminus (\mathcal{X}_0 \cup \{x\})$ which is stable after the ith narrowing iteration. This assertion holds for

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

$i = 0$. For $i > 0$, the assertion on the $y \neq x$ follows by inductive hypothesis for a larger set of called unknowns. And for x, we have

$$\begin{aligned} d_i &= (F^\sharp{}_x(\tau \oplus \sigma_i \oplus \{x \mapsto d'_{i-1}\}))_1 \\ &\sqsupseteq (\underline{F}^{(i)}_x(\sigma_i))_1 \\ &\sqsupseteq (\underline{F}^{(i)}_x(\underline{\sigma}_i))_1 \\ &\sqsupseteq (\underline{F}^\sharp{}_x(\underline{\sigma}_i \oplus \{x \mapsto d_{i-1}\}))_1 \\ &\sqsupseteq (\underline{F}^\sharp{}_x\underline{\sigma})_1 \end{aligned}$$

Here, $\underline{F}^\sharp{}_x$ and $\underline{F}^{(i)}_x$ are the right-hand sides of the lower monotonizations of $\mathcal{E}^\sharp{}_\tau$ and $\mathcal{E}^\sharp{}_{\tau \oplus \{x \mapsto d_{i-1}\}}$ for x, respectively. This completes the proof of the theorem. □

7. The Space-efficient Solver TD$_{\text{space}}$

So far, our solver maintains an abstract value for each queried unknown. Given that the program to be analyzed is not small (e.g., more than 10,000 LOC) and program points must be analyzed for multiple contexts, the number of unknowns to be considered by the solver can be quite large. For more complicated properties to be analyzed, these abstract values to be recorded for these unknowns in themselves are space-consuming. The applicability of solvers for interprocedural analysis based on such solvers therefore is significantly increased if space consumption can be reduced. This is the objective of our second modification to the local generic solver **TD**.

```
let rec destabilize x = (* ... *)

let rec eval x y =
  if y ∈ called then
    point += y;
  if y ∈ point then (
    solve ∇ y;
    !infl y += x;
    !σ y
  ) else (
    called += y;
    let tmp = f♯_y (eval x) in
    called -= y;
    if y ∉ point then tmp
    else (
      solve ∇ y;
      !infl y += x;
      !σ y
    )
  )

and solve p x =
  if x ∉ stable && x ∉ called then (
    stable += x;
```

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

```
    called += x;
    let tmp = (f♯x (eval x)) in
    called -= x;
    let tmp = p (!σ x) tmp in
    if !σ x = tmp then
      if p = ∇ && x ∈ point then (
        stable -= x;
        solve Δ x
      ) else ()
    else (
      σ, x := tmp;
      destabilize x;
      solve p x
    )
  )

let solve x = point += x;
              solve ∇ x;
              (σ, stable)
```

In contrast to algorithm $\mathbf{TD_{term}}$, the new solver maintains abstract values only for widening and narrowing points as collected in the set *point*. The intuition is that the current values in σ for all other unknowns can be reconstructed by evaluating their right-hand sides. Thus, we only call solve for unknowns in *point*, which is why we now always perform widening or narrowing in solve.

We remark that in absence of procedure calls, the set *point* may be statically chosen as the set of *loop heads* – given that each loop is dominated by a single program point. In presence of procedure calls, however, this is no longer easily possible.

Example 11. Consider again the example program from Figure 1, and the corresponding elaborated right-hand sides from Example 5. For the assignment σ with $\sigma\langle 3,a\rangle = \sigma\langle 4,a\rangle = \sigma\langle 6,a\rangle = a$ and $\sigma\langle 5,a\rangle = h^\sharp{}_1\, a$, the right-hand side of unknown $\langle 7,a\rangle$ accesses, for example, the unknown $\langle 7, \sigma\langle 5,a\rangle\rangle = \langle 7, h^\sharp{}_1\, a\rangle$ which may put $\langle 7,a\rangle$ into *point* only if $(h^\sharp{}_1)^r a = a$ for some $r > 0$. □

The call eval x y behaves the same as before for unknowns $y \in$ *point*. For unknowns $y \notin$ *point*, the value now must be recovered. For that, y first is marked as *called*. Then, the right-hand side of y is evaluated (still passing x as first argument to eval); finally, y is again removed from *called*. If y is still not in *point*, the result for y is plainly returned. If, however, the evaluation of $f^\sharp{}_y$ has inserted y into the set *point*, the function eval proceeds as if y had been contained in *point* right from the beginning. This means that solve ∇ y is called, x is inserted into the set *infl* y, and subsequently, the value of σ for y is returned.

We remark that on some inputs, the solver $\mathbf{TD_{space}}$ may be rather inefficient, since the same unknown $y \notin$ *point* must be re-evaluated whenever the value of y is queried. At the expense of slightly more space, this deficiency can be remedied by maintaining the values for these unknowns encountered during the re-evaluation of the right-hand side of some unknown $x \in$ *point* in a separate map τ (see the algorithm in Section B of the appendix).

8. Termination and Correctness of $\mathbf{TD_{space}}$

In the following, we convince ourselves that the solver $\mathbf{TD_{space}}$ has the same termination behavior as the solver $\mathbf{TD_{term}}$.

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

For a finite subset of unknowns $Y \subseteq \mathcal{X}^\sharp$, we construct from $\mathcal{E}^\sharp$ the *residual* system $\mathcal{E}^\sharp{}_Y$ where the right-hand sides $\bar{f}^\sharp_{Y,y}$ are obtained from the right-hand sides $f^\sharp_y$ successively exploring the right-hand sides of unknowns not contained in Y. Technically, this means that for $y \in Y$ and $\sigma : Y \to \mathcal{M}(\mathbb{D})$,

$$\begin{aligned} f^\sharp_{Y,y}\,\sigma \quad &= f^\sharp_y\,(\mathsf{bar}_Y\,\sigma) \qquad \text{where} \\ \mathsf{bar}_Y\,\sigma\,y &= \textbf{if}\; y \in Y \;\textbf{then}\; \sigma\,y \;\textbf{else}\; f^\sharp_y\,(\mathsf{bar}_Y\,\sigma) \end{aligned}$$

Example 12. Consider the monadic right-hand side $f^\sharp_{\langle 7,a\rangle}$ from Example 6 for the program from Figure 1, and $Y = \{\langle 7, a\rangle \mid a \in \mathbb{D}\}$, we have

$$\begin{aligned} f^\sharp_{Y,\langle 7,a\rangle}\,\sigma &= f^\sharp_{\langle 7,a\rangle}\,(\mathsf{bar}_Y\,\sigma) \\ &= \mathsf{bind}\,(\mathsf{bar}_Y\,\sigma\,\langle 5,a\rangle)\,(\textbf{fun}\; b_1 \to \\ &\quad \mathsf{bind}\,(\mathsf{bar}\,\sigma\,\langle 7,b_1\rangle)\,(\textbf{fun}\; b_2 \to \\ &\quad \mathsf{bind}\,(\mathsf{bar}_Y\,\sigma\,\langle 6,a\rangle)\,(\textbf{fun}\; b_3 \to \\ &\quad \mathsf{return}\,(\mathsf{combine}^\sharp\; b_1\; b2 \sqcup h^\sharp_2\; b_3)))) \\ &= \mathsf{bind}\,(\mathsf{return}\,(h^\sharp_1\,a))\,(\textbf{fun}\; b_1 \to \\ &\quad \mathsf{bind}\,(\sigma\,\langle 7,b_1\rangle)\,(\textbf{fun}\; b_2 \to \\ &\quad \mathsf{bind}\,(\mathsf{return}\,a)\,(\textbf{fun}\; b_3 \to \\ &\quad \mathsf{return}\,(\mathsf{combine}^\sharp\; b_1\; b2 \sqcup h^\sharp_2\; b_3)))) \\ &= \mathsf{bind}\,(\sigma\,\langle 7,h^\sharp_1\,a\rangle)\,(\textbf{fun}\; b_2 \to \\ &\quad \mathsf{return}\,(\mathsf{combine}^\sharp\,(h^\sharp_1\,a)\; b_2 \sqcup h^\sharp_2\,a)) \end{aligned}$$

□

In general, let $F^\sharp{}_{Y,y}$ be the elaboration of $f^\sharp_{Y,y}$. For some $y \in Y$ and some σ, the evaluation of the right-hand side $f^\sharp_{Y,y}$ may not terminate – in which case, $F^\sharp{}_{Y,y}\,\sigma$ is undefined. For $\subseteq Y$, let us call σ (Y, dom)-*consistent* if $F^\sharp{}_{Y,y}\,\sigma$ is defined for all $y \in dom$. In that case, there is a set $\bar{Y} = Y \cup \{y_1, \ldots, y_h\} \subseteq \mathcal{X}$ together with a map $\bar{\sigma} : \bar{Y} \to \mathbb{D}$ such that $\bar{\sigma}\,|_Y = \sigma$, and

1. $(F^\sharp{}_{y_j}\,\bar{\sigma})_1 = \bar{\sigma}\; y_j$ and $(F^\sharp{}_{y_j}\,\bar{\sigma})_2 \subseteq \bar{Y} \cup \{y_1, \ldots, y_{j-1}\}$ for all $j = 1, \ldots, h$; and
2. $(F^\sharp{}_y\,\bar{\sigma})_2 \subseteq \bar{Y}$ for all $y \in dom$.

The values of the unknowns in $\bar{Y}$ thus are sufficient to evaluate all right-hand sides of unknowns in $dom \cup (\bar{Y} \setminus Y)$, while the values of $\bar{Y} \setminus Y$ can be recovered from the values of the unknowns in Y.

Example 13. Continuing with Example 12, we have that the elaborated right-hand side $F^\sharp{}_{Y,\langle 7,a\rangle}$ is given by:

$$F^\sharp{}_{Y,\langle 7,a\rangle}\,\sigma = (\mathsf{combine}^\sharp\,(h^\sharp_1\,a)\,(\sigma\,\langle 7,h^\sharp_1\,a\rangle) \sqcup h^\sharp{}_2\,a, \{\langle 7,h^\sharp_1\,a\rangle\})$$

where the required auxiliary unknowns y are given by

$$\langle 3,a\rangle, \langle 4,a\rangle, \langle 5,a\rangle, \langle 6,a\rangle$$

□

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

Subsequently, we adapt the notion of consistency from Section 5 to the case where values from $\mathbb{D}$ are only recorded for unknowns in *point*. Moreover, we maintain that *stable* as well as all sets *infl* y are all subsets of *point*. We now call a solver state $s = (\sigma, infl, stable, called, point)$ *consistent* if for $Y = (stable \cup called) \cap point$ and $dom = stable \backslash called$,

1. σ is (Y, dom)-consistent;
2. for all $x \in dom$ and all $y \in (F^\sharp{}_{Y,x}\, \sigma)_2$, $y \in Y$ and $x \in infl\, y$ holds;
3. for each $x \notin Y$, $infl\, x = \emptyset$.

With this new notion, the invariants for the calls solve $p\, x$, and destabilize x from Section 5, now stay literally the same – with the extra assumption that x should necessarily be contained in *point*. For eval $x\, y$, the new invariant must distinguish whether y is contained in *point* or not. Assuming that the solver state $s = (\sigma, infl, stable, called, point)$ before the call is consistent where $x \in stable \cap called \cap point$, the solver state $s' = (\sigma', infl', stable', called', point')$ after the call should now satisfy:

1. s' is again consistent;
2. $called = called'$, and
3. $stable \subseteq stable'$ where for all $y' \in (stable \cup called) \cap point$, $\sigma'\, y' = \sigma\, y'$ and $infl\, y' \subseteq infl'\, y'$;
4. $point \subseteq point'$ where $y \in point'$ whenever $y \in called$;
5. If $y \in (stable \cup called) \cap point$, then
   - $\sigma = \sigma'$, $stable = stable'$ and $point = point'$;
   - $infl'\, y' = infl'\, y'$ for all $y' \neq y$, and
   - $infl'\, y = infl\, y \cup \{x\}$.
6. In all cases when $y \in point'$, then $y \in stable'$ and $x \in infl'\, y$ and the value $\sigma'\, y$ is returned;
7. If $y \notin point'$, then $F^\sharp{}_{Y',y}\, \sigma'$ is defined and produces the return value where $Y' = (stable' \cup called') \cap point'$.

In particular, x is still contained in $stable' \cap called'$. What we additionally need is an extra argument why evaluation of the right-hand sides of unknowns $y \notin point$ will necessarily terminate. For that, we observe that, according to our assumption, evaluation of each right-hand side $f^\sharp{}_y$ will access only finitely many unknowns from $\mathcal{X}$. Also, we note that before evaluation of $f^\sharp{}_y$, the set *called* additionally receives the unknown y. An unknown z accessed during the evaluation of the right-hand side $f^\sharp{}_y$ can only be found not in *point*, if $z \notin called \cup \{y\}$, that is, each unknown into which recursive evaluation descends must be different from all unknowns added to the set *called* so far.

Assuming that altogether only finitely many unknowns are encountered during solving, recursive evaluation will eventually terminate having called finitely many times solve for unknowns in *point*. Therefore, we can adapt the proof of Theorem 1 to obtain:

Theorem 3. *Let $\mathcal{E}^\sharp$ denote an arbitrary system of abstract equations, and $x_0 \in \mathcal{X}^\sharp$ the unknown of interest. Assume that initially the sets called and stable are empty, and likewise, infl maps each unknown to the empty set. Assume further that $x \in point$. Then, the call* solve x_0 *of solver* $\mathbf{TD}_{\mathbf{space}}$ *will always terminate, as long as only finitely many unknowns are encountered.* □

The notion of saturation of solver states literally stays the same. Let $s = (\sigma, infl, stable, called, point)$ denote a consistent solver state. Let $\tau : called \to \mathbb{D}$ denote the restriction of σ to the set *called*. We call s *saturated* if for all $x \in stable \setminus called$, $\underline{\sigma}\, x \sqsubseteq \sigma\, x$ where $(\underline{\sigma}, X^\sharp \setminus called)$ is the least total solution of the lower monotonization of $\mathcal{E}^\sharp{}_\tau$. For the space-efficient solver $\mathbf{TD}_{\mathbf{space}}$, Theorem 2 must be re-phrased as follows:

Theorem 4. *Let $x \in point$, and consider a call* solve $\nabla\ x$ *encountered by the solver* $\mathbf{TD}_{\mathbf{space}}$ *during the evaluation of the initial call. Then it always starts in a saturated solver state, and upon*

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

termination, it results in a solver state $s' = (\sigma', infl', stable', called', point')$ *which is again saturated so that* $x \in stable'$.

The proof is along the same lines as the proof for Theorem 2 – only that now the assignment σ must be replaced with Y-consistent mappings $\bar{\sigma}_Y$ for suitable Y with $stable \setminus called \subseteq Y \subseteq stable' \setminus called$.

Proof. Let $\mathcal{X}'$ denote the set of all unknowns in $stable' \setminus called'$. In particular, $x \in \mathcal{X}'$. We proceed by induction on the set of unknowns *not* contained in the set $\mathcal{X}_0$ of called unknowns. Again, we concentrate on the last tail-recursive call solve $\nabla\ x$ before the call to solve $\Delta\ x$. Let d denote the value of x *before* the evaluation of the right-hand side, and d' the value returned by evaluating the right-hand side. Let $\mathcal{X}'$ denote the set of unknowns accessed during the call. Since subsequently solve $\Delta\ x$ is called, after that call x is contained in *stable*. Therefore, one of the following two situations is encountered after the evaluation of the right-hand side:

1. $d' \sqsubseteq d$, that is, the value d' is subsumed by the current value of x; or
2. subsequent destabilization will not destabilize x.

In the second case, the unknowns that directly or indirectly influence x do cannot depend on x (w.r.t. the current sets *infl*). Thus, a similar argument as for unknowns not in *point* in the proof of Theorem 2 applies. Accordingly, it remains to consider the first case. Let Y denote the set $stable \cup called$ in the current solver state, and τ the restriction of σ to $called \cap point$, extended with $\{y \mapsto \bot \mid y \in called \setminus point\}$. Consider the lower monotonizations of the abstract systems ${\mathcal{E}^\sharp}_\tau$ and ${\mathcal{E}^\sharp}_{\tau \oplus \{x \mapsto d\}}$ with least solutions $\underline{\sigma}$ and $\underline{\sigma}'$, respectively. Let σ_0 denote the assignment to the unknowns in $\mathcal{X}^\sharp \setminus (\mathcal{X}_0 \cup \{x\})$ before the call solve $\Delta\ x$. By inductive hypothesis applied to the unknowns in $\mathcal{X}' \setminus \{x\}$ and $\tau \oplus \{x \mapsto d\}$, we find that $\underline{\sigma}'\, y \sqsubseteq \sigma_0\, y$ for all unknowns $y \in \mathcal{X}' \setminus \{x\}$ where $\underline{\sigma}' \oplus \{x \mapsto d\}$ is a post-solution of the lower monotonization of ${\mathcal{E}^\sharp}_\tau$. Therefore, $\underline{\sigma}\, y \sqsubseteq \underline{\sigma}' \oplus \{x \mapsto d\} \sqsubseteq (\sigma_0 \oplus \{x \mapsto d\})\, y$ for all $y \in \mathcal{X}'$ holds, and the claim follows. Now consider the narrowing iteration performed for x in the subsequent call solve $\Delta\ x$. Let d_0 denote the value after the last call to solve $\nabla\ x$, and $d_1, \ldots, d_k$ denote the values returned by evaluating the right-hand side of x during this iteration, and define $d'_0 = d_0$ and for $i > 0$, $d'_i = d'_{i-1}\ \Delta\ d_i$. Let σ_i denote the assignment attained after the ith narrowing step restricted to the unknowns $\mathcal{X}^\sharp \setminus (\mathcal{X}_0 \cup \{x\})$. For $i > 0$, let $\underline{\sigma}_i$ denote the least solution of the lower monotonization of ${\mathcal{E}^\sharp}_{\tau \oplus \{x \mapsto d_{i-1}\}}$. By inductive hypothesis, $\underline{\sigma}_i\, y \sqsubseteq \sigma_i$ for all $y \neq x$ which are stable after the ith iteration. By induction on i, we prove that $\underline{\sigma}\, x \sqsubseteq d_i$ and thus also $\underline{\sigma}\, x \sqsubseteq d'_i$, and $\underline{\sigma}\, y \sqsubseteq \sigma_i\, y$ for every $y \in \mathcal{X}^\sharp \setminus (\mathcal{X}_0 \cup \{x\})$ which is stable after the ith narrowing iteration. This assertion holds for $i = 0$. For $i > 0$, the assertion on the $y \neq x$ follows by inductive hypothesis for a larger set of called unknowns. And for x, we have

$$\begin{aligned} d_i &= ({F^\sharp}_{Y_i,x}(\tau \oplus \sigma_i \oplus \{x \mapsto d'_{i-1}\}))_1 \\ &\sqsupseteq (\underline{F}^{(i)}_x(\sigma_i))_1 \\ &\sqsupseteq (\underline{F}^{(i)}_x(\underline{\sigma}_i))_1 \\ &\sqsupseteq (\underline{F^\sharp}_x(\underline{\sigma}_i \oplus \{d_{i-1}\}))_1 \\ &\sqsupseteq (\underline{F^\sharp}_x \underline{\sigma})_1 \end{aligned}$$

Here, ${F^\sharp}_{Y_i,x}$ is the elaborated right-hand side for $f^\sharp_x \circ \mathsf{bar}_{Y_i}$ where Y_i equals the union of the set $point \cup called$ after the ith iteration of solve $\Delta\ x$. Moreover, $\underline{F^\sharp}_x$ and $\underline{F}^{(i)}_x$ are the right-hand sides of the lower monotonizations of ${\mathcal{E}^\sharp}_\tau$ and ${\mathcal{E}^\sharp}_{\tau \oplus \{x \mapsto d_{i-1}\}}$ for x, respectively. This completes the proof of the theorem. □

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

9. Side-Effecting Systems of Equations

Concurrency libraries such as POSIX threads (Walli 1995) or OSEK (Lemieux 2001) support communication between threads by means of shared program variables and data structures, while primitives such as *mutexes* or *resources* are provided to synchronize their executions.

Example 14. Consider the program

```
1  int g, h; // global variables
2  int main(){
3     g = 0; h = 0;
4     create(f);
5     g = 1; // side-effect
6     return 0;
7   }
8  int f(){
9     if (g) h = 1; // side-effect
10    return 0;
11  }
```

with global variables g, h, where the call create(f) is meant to spawn a new thread which executes the parameterless function f. The function main thus first initializes the globals g, h. Then, a thread is spawned executing the function f. Finally, g is set to 1, followed by returning with 0. The function f, when executed, checks the global variable g. If it is different from 0, 1 is assigned to h. Finally, 0 is returned. Accordingly, the two threads communicate via the shared program variable g. □

One natural way to de-couple the analysis of multi-threaded code is to interprocedurally analyze only the thread-local states, while a flow-insensitive *global invariant* is accumulated for shared data (Seidl and Vogler 2017; Vojdani and Vene 2009). This idea is realized in the analyzer GOBLINT (Vojdani et al. 2016). To a certain extent, this approach can be accommodated also to analyses taking mutexes into account such as Mine (2012, 2014). One way of conveniently combining flow-insensitive analysis of shared data with context-sensitive analysis of local state is by extending systems of equations with *side* effects (Apinis et al. 2012). Side effects during solving should be seen in analogy to the meta-predicate assert in PROLOG clauses: while computing a contribution to the predicate of the head, a contribution to some other predicate is triggered. A right-hand side function $f^\sharp_x$ in monadic form now has type:

$$(\mathcal{X}^\sharp \to \mathcal{M}(\mathbb{D})) \to (\mathcal{X} \to \mathbb{D} \to \mathcal{M}(\bullet)) \to \mathcal{M}(\mathbb{D})$$

for *any* monad $\mathcal{M}$. The $\bullet$ here represents the one-element complete lattice $\bullet = \{()\}$. The second argument function is meant to be called for triggering side effects. These events are observed by the solver, but otherwise do not affect the computed value. Accordingly, the return values of the second argument function should be in $\mathcal{M}(\bullet)$.

Example 15. Consider the program from Example 14. Let us assume that we use intervals as values for each of these unknowns. Since the functions have no local variables, but a return value each, let us represent the local state of a function by a single interval as well which may initially be thought of having value $\top = [-\infty, \infty]$. The part of the system of abstract equations for calling contexts $\top$ is given by the right-hand sides (in monadic form) of the unknowns g, h as well as main, and f corresponding to the end points of the respective functions:

$$f^\sharp_g \text{ get set} = \text{return } \emptyset$$
$$f^\sharp_h \text{ get set} = \text{return } \emptyset$$

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

$$
\begin{aligned}
f^\sharp_{\mathsf{main}}\ \mathsf{get}\ \mathsf{set} &= \mathsf{bind}\ (\mathsf{set}\ g\ [0,0])\ (\mathbf{fun}\ () \to \\
&\quad\ \mathsf{bind}\ (\mathsf{set}\ h\ [0,0])\ (\mathbf{fun}\ () \to \\
&\quad\ \mathsf{bind}\ (\mathsf{get}\ f)\ (\mathbf{fun}\ \_ \to \\
&\quad\ \mathsf{bind}\ (\mathsf{set}\ g\ [1,1])\ (\mathbf{fun}\ () \to \\
&\quad\ \mathsf{return}\ [0,0])))) \\
f^\sharp_f\ \mathsf{get}\ \mathsf{set} &= \mathsf{bind}\ (\mathsf{get}\ g)\ (\mathbf{fun}\ d \to \\
&\qquad \mathbf{if}\ \neg(d \subseteq [0,0])\ \mathbf{then} \\
&\qquad\quad \mathsf{bind}\ (\mathsf{set}\ h\ [1,1])\ (\mathbf{fun}\ () \to \\
&\qquad\quad \mathsf{return}\ [0,0]) \\
&\qquad \mathbf{else}\ \mathsf{return}\ [0,0])
\end{aligned}
$$

The unknowns g and h corresponding to the global variables of the program thus have *trivial* right-hand sides. Both unknowns are meant to receive their values solely via side effects. □

Due to parametricity in the monad, each right-hand side function of a side-effecting abstract system of equations again can be represented by a *generalized* execution tree t_x such that $f^\sharp{}_x = [\![t_x]\!]$. Generalized execution trees now make not only explicit which unknowns are accessed, but also which side effects should be triggered during evaluation. Therefore, they additionally may contain a constructor

$$\mathsf{S}\ (\mathcal{X}^\sharp, \mathbb{D}, \mathcal{T})$$

The intention is that evaluation of $\mathsf{S}\ (x, d, t)$ first adds $d \in \mathbb{D}$ to the value of x and then continues with the evaluation of t. Accordingly, the semantics of such a generalized tree t is the function $[\![t]\!] : (\mathcal{X}^\sharp \to \mathcal{M}(\mathbb{D})) \to (\mathcal{X}^\sharp \to \mathbb{D} \to \mathcal{M}(\bullet)) \to \mathcal{M}(\mathbb{D})$ which for functions $\mathsf{get} : \mathcal{X} \to \mathcal{M}(\mathbb{D})$ and $\mathsf{set} : \mathcal{X} \to \mathbb{D} \to \mathcal{M}(\bullet)$ is defined by

$$
\begin{aligned}
[\![\mathsf{A}\ d]\!]\ \mathsf{get}\ \mathsf{set} &= \mathsf{return}\ d \\
[\![\mathsf{Q}\ (x,f)]\!]\ \mathsf{get}\ \mathsf{set} &= \mathsf{bind}\ (\mathsf{get}\ x)\ (\mathbf{fun}\ d \to [\![f\ d]\!]\ \mathsf{get}\ \mathsf{set}) \\
[\![\mathsf{S}\ (x,d,t)]\!]\ \mathsf{get}\ \mathsf{set} &= \mathsf{bind}\ (\mathsf{set}\ x\ d)\ (\mathbf{fun}\ () \to [\![t]\!]\ \mathsf{get}\ \mathsf{set})
\end{aligned}
$$

In case of a state transformer monad with set of states S, this amounts to

$$
\begin{aligned}
[\![\mathsf{A}\ d]\!]\ \mathsf{get}\ \mathsf{set}\ s &= (s, d) \\
[\![\mathsf{Q}\ (x,f)]\!]\ \mathsf{get}\ \mathsf{set}\ s &= \mathbf{let}\ (s', d) = \mathsf{get}\ x\ s \\
&\qquad\quad \mathbf{in}\ [\![f\ d]\!]\ \mathsf{get}\ \mathsf{set}\ s' \\
[\![\mathsf{S}\ (x,d,t)]\!]\ \mathsf{get}\ \mathsf{set}\ s &= \mathbf{let}\ (s', \_) = \mathsf{set}\ x\ d\ s \\
&\qquad\quad \mathbf{in}\ [\![t]\!]\ \mathsf{get}\ \mathsf{set}\ s'
\end{aligned}
$$

Consider the set of extended states $S = (\mathcal{X}^\sharp \to \mathbb{D}) \times \mathcal{P}(\mathcal{X}^\sharp) \times (\mathcal{X}^\sharp \to \mathbb{D})$ where the third component accumulates side effects. Consider the functions

$$
\begin{aligned}
\mathsf{get}\ x\ (\sigma, X, \rho) &= \mathbf{let}\ d = \sigma\ x \\
&\qquad\quad \mathbf{in}\ ((\sigma, X \cup \{x\}, \rho), d) \\
\mathsf{set}\ x\ d\ (\sigma, X, \rho) &= \mathbf{let}\ \rho = \rho \oplus \{x \mapsto \rho\ x \sqcup d\} \\
&\qquad\quad \mathbf{in}\ ((\sigma, X, \rho), \bullet)
\end{aligned}
$$

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

Then, the *elaborated* right-hand side function ${F^\sharp}_x$ has type

$$ {F^\sharp}_x : (\mathcal{X}^\sharp \to \mathbb{D}) \to (\mathbb{D} \times \mathcal{P}(\mathcal{X}^\sharp) \times (\mathcal{X}^\sharp \to \mathbb{D})) $$

where the third component of the result represents the side effects to other unknowns encountered during evaluation of ${f^\sharp}_x$. This function is given by:

$$ \begin{aligned} F^\sharp_x\, \sigma \;=\; & \mathbf{let}\; ((\_, X, \rho), d) = {f^\sharp}_x\; \mathsf{get}\; \mathsf{set}\; (\sigma, \emptyset, \underline{\bot}) \\ & \mathbf{in}\; (d, X, \rho) \end{aligned} $$

Example 16. Let us consider the right-hand side functions from Example 15 for the interval analysis of the program in Example 14 at the beginning of this section. These are now are given by

$$ \begin{aligned} t_g \quad &= \mathsf{A}\, \emptyset \\ t_h \quad &= \mathsf{A}\, \emptyset \\ t_{\mathsf{main}} &= \mathsf{S}\, (g, [0,0], \mathsf{S}\, (h, [0,0], \\ & \qquad \mathsf{Q}\, (f, \mathbf{fun}\, \_ \to\; \mathsf{S}\, (g, [1,1], \mathsf{A}\, [0,0])))) \\ t_f \quad &= \mathsf{Q}\, (g, \mathbf{fun}\, d \to\; \mathbf{if}\, \neg (d \subseteq [0,0])\; \mathbf{then}\, \mathsf{S}\, (h, [1,1], \mathsf{A}\, [0,0]) \\ & \qquad\qquad\qquad\qquad \mathbf{else}\, \mathsf{A}\, [0,0]) \end{aligned} $$

The computation tree t_{main} for main first produces side effects $[0,0]$ onto the unknowns g and h, respectively. Then, it queries the return value of f, but ignores that value. This query enables the local solver to start the evaluation of the unknown f and thus the exploration of the function f. Subsequently, another side effect $[1,1]$ is produced onto the unknown g, before the value $[0,0]$ is returned. The computation tree t_f for f first queries g. Depending on the obtained value, a side effect of $[1,1]$ is produced onto the variable h or omitted. Eventually then the value $[0,0]$ is returned. Elaboration of the right-hand side functions results in the functions

$$ \begin{aligned} {F^\sharp}_g\, \sigma \quad &= (\emptyset, \emptyset, \underline{\bot}) \\ {F^\sharp}_h\, \sigma \quad &= (\emptyset, \emptyset, \underline{\bot}) \\ {F^\sharp}_{\mathsf{main}}\, \sigma &= ([0,0], \{ f \}, \underline{\bot} \oplus \{g \mapsto [0,1]\}) \\ {F^\sharp}_f\, \sigma \quad &= ([0,0], \{ g \}, \underline{\bot} \oplus \{h \mapsto \mathbf{if}\, \neg (\sigma\, g \subseteq [0,0])\; \mathbf{then}\, [1,1]\; \mathbf{else}\, \emptyset\}) \end{aligned} $$

Here, $\underline{\bot}$ denotes the assignment mapping each unknown to $\bot$ – which in the example equals $\emptyset$. As can be seen for main, elaborated right-hand side functions may combine multiple side effects to the same unknown (in this case, g) into one and also no longer differentiate when during the evaluation, each side effect is produced. □

Side effects also come in handy when only parts of the context are used to discriminate procedure calls (Apinis et al. 2012). Assume that we are given a mapping $\pi : \mathbb{D} \to A$ for some set A of distinguishing abstract properties of contexts. Consider the abstract effect of a call to a procedure p. Let u, v denote the program points before and after the call, respectively. Then for every abstract context $a \in A$, the right-hand side for the unknown $\langle v, a \rangle$ is given by

$$ \begin{aligned} {f^\sharp}_{v,a}\; \mathsf{get}\; \mathsf{set} = {} & \mathsf{bind}\; (\mathsf{get}\; \langle u, a \rangle)\; (\mathbf{fun}\, d \to\; \mathbf{let}\; a' = \pi\; d\; \mathbf{in} \\ & \mathsf{bind}\; (\mathsf{set}\; \langle \mathsf{st}_p, a' \rangle\; d)\; (\mathbf{fun}\, () \to \\ & \mathsf{bind}\; (\mathsf{get}\; \langle \mathsf{ret}_p, a' \rangle)\; (\mathbf{fun}\, d' \to \\ & \mathsf{return}\; (\mathsf{combine}^\sharp\; d\; d')))) \end{aligned} $$

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

or, alternatively, by the computation tree

$$\begin{aligned}&\mathsf{Q}\,(\langle u, a\rangle, \mathbf{fun}\, d \to \mathbf{let}\ a' = \pi\ d\ \mathbf{in}\\ &\mathsf{S}\,(\langle \mathsf{st}_p, a'\rangle, d,\\ &\mathsf{Q}\,(\langle \mathsf{ret}_p, a'\rangle, \mathbf{fun}\, d' \ \to\\ &\mathsf{A}\,(\mathsf{combine}^\sharp\ d\ d'))))\end{aligned}$$

where $\mathsf{combine}^\sharp : \mathbb{D} \to \mathbb{D} \to \mathbb{D}$ again combines the abstract program state attained at the end-point ret_p of p with the state before the call to the state of the caller after the call.

In this formalization, the side effect to the start point st_p of the called procedure p is used to accumulate all values d leading to the same abstract context a' for which p is analyzed. In order to prove the resulting analysis sound, it is convenient to reformulate also the concrete collecting semantics by introducing side effects. For a concrete program state q, the right-hand side $\mathcal{E}\ \langle v, q\rangle\ \sigma$ for the unknown representing the procedure call in concrete calling context q, is given by

$$\begin{aligned}&\mathbf{let}\ S &&= \sigma\ \langle u, q\rangle\ \mathbf{in}\\ &\mathbf{let}\ S' &&= \{\mathsf{combine}\, q'\ q'' \mid q' \in S, q'' \in \sigma\,\langle \mathsf{ret}_f, q'\rangle\}\ \mathbf{in}\\ &\mathbf{let}\ X' &&= \{\langle u, q\rangle\} \cup \{\langle \mathsf{ret}_f, q'\rangle \mid q' \in S\}\ \mathbf{in}\\ &\mathbf{let}\ E' &&= \{\langle \mathsf{st}_f, q'\rangle \mapsto \{q'\} \mid q' \in S\}\ \mathbf{in}\\ &&&(S', X', E')\end{aligned}$$

For convenience we here introduce the convention that those unknowns not explicitly listed as arguments in E' are implicitly all mapped to $\bot$ ($\emptyset$ in case of the concrete collecting semantics).

Interestingly, the *description relation* $\mathcal{R}$ between unknowns of the concrete and abstract systems can now no longer be specified as is, but must take some over-approximation ρ of all abstract side effects into account. In order to see this, assume for a moment that the set A contains a single element $\bullet$, that is, procedures are analyzed *without* context. Consider the unknown $\langle \mathsf{st}_p, \bullet\rangle$ for the procedure p. This unknown should describe concrete unknowns $\langle \mathsf{st}_p, q\rangle$ – however, not for *all* program states q. Instead, only those q must be taken into account which actually occur as calling contexts of p in the collecting semantics of the program. A (hopefully not too large) superset of these is given by $\gamma(\rho\,\langle \mathsf{st}_p, \bullet\rangle)$ if ρ is an over-approximation of the side effects encountered during the abstract fixpoint iteration. More generally, we define for interprocedural analysis with a set A of abstract distinguishing contexts,

$$\langle u', q\rangle\ \mathcal{R}_\rho\ \langle u', a\rangle \quad \text{iff} \quad q \in \gamma(\rho\ \langle \mathsf{st}_p, a\rangle)$$

if u' is a program point of the procedure p and $a \in A$. Subsequently, we therefore assume that we are given a *description* relation $\mathcal{R}_\rho \subseteq \mathcal{X} \times \mathcal{X}^\sharp$ between the concrete and abstract unknowns depending on some over-approximation of the side effects ρ. As in Section 3, the description relation $\mathcal{R}_\rho$ on unknowns is extended to sets of unknowns and assignments to unknowns, right-hand sides and systems of equations. For elaborated right-hand side functions $F, F^\sharp$, $F\ \mathcal{R}_\rho\ F^\sharp$ is meant to hold whenever for assignments $\sigma, \sigma^\sharp$ with $\sigma\ \mathcal{R}_\rho\ \sigma^\sharp$, the following holds:

- $(F\,\sigma)_1 \subseteq \gamma((F^\sharp\ \sigma^\sharp)_1)$;
- $(F\,\sigma)_2\ \mathcal{R}_\rho\ (F^\sharp\ \sigma^\sharp)_2$ where
- $(F\,\sigma)_3\ \mathcal{R}_\rho\ \rho$ as well as $(F^\sharp\ \sigma^\sharp)_3 \sqsubseteq \rho$.

while then for equation systems $\mathcal{E}, \mathcal{E}^\sharp$, $\mathcal{E}\ \mathcal{R}_\rho\ \mathcal{E}^\sharp$ holds if $F_x\ \mathcal{R}_\rho\ F^\sharp{}_{x^\sharp}$ holds for all unknowns $x, x^\sharp$ with $x\ \mathcal{R}_\rho\ x^\sharp$.

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

Accordingly, the mapping ρ is used to *de-couple* individual occurrences of side effects in right-hand sides of the concrete and the abstract semantics, respectively. Now assume that we are given a system $\mathcal{E}^\sharp$ of abstract equations with side effects where the elaborated right-hand side for an unknown $x \in \mathcal{X}^\sharp$ is given by $F^\sharp{}_x : (\mathcal{X}^\sharp \to \mathbb{D}) \to (\mathbb{D} \times \mathcal{P}(\mathcal{X}^\sharp) \times (\mathcal{X}^\sharp \to \mathbb{D}))$. Assume that *leaf* is a subset of abstract unknowns y where the right-hand side is described by the computation tree $A\,\bot$, that is, $F^\sharp{}_y\,\sigma = (\bot, \emptyset, \underline{\bot})$ where $\underline{\bot}$ is the mapping which assigns $\bot$ to each unknown. For convenience, we make the extra assumption that side effects only occur to unknowns in *leaf*.

A *partial post-solution* of $\mathcal{E}^\sharp$ has to take side effects incurred by the evaluation of the right-hand sides into account. Let $\sigma : \mathcal{X}^\sharp \to \mathbb{D}$ and $dom^\sharp \subseteq \mathcal{X}^\sharp$. Then we call $dom^\sharp$ $(\sigma, \mathcal{E}^\sharp)$-closed if for all $x \in dom^\sharp$ and $F^\sharp{}_x\,\sigma = (d, X, E)$, $X \subseteq dom^\sharp$ and also $E\,y \neq \bot$ only for unknowns $y \in dom^\sharp$. The pair $(\sigma, dom^\sharp)$ is a *partial post-solution* of $\mathcal{E}^\sharp$ if

1. $dom^\sharp$ is $(\sigma, \mathcal{E}^\sharp)$-closed, and
2. for all $x \in dom^\sharp$ with $F^\sharp{}_x\,\sigma = (d, X, E)$,
   (a) $\sigma \sqsupseteq E$, and
   (b) $\sigma\,x \sqsupseteq d$.

Our goal is to design an extension of the generic local solver $\mathbf{TD_{term}}$ which is able to deal with side-effecting systems of abstract equations. Due to the intertwined narrowing iterations, property (2.b) may be violated. Therefore, consider a pair $(\sigma, dom^\sharp)$ satisfying properties (1) and (2.a). Let $\rho : \mathcal{X}^\sharp \to \mathbb{D}$ denote the *accumulated* side effects of the pair, that is,

$$\rho\,y = \bigsqcup \{(F^\sharp{}_x\,\sigma)_3\,y \mid x \in dom^\sharp\}$$

We remark that $\rho\,y \neq \bot$ only for unknowns $y \in dom^\sharp$. Let $\mathcal{E}^\rho$ denote the system of abstract equations (without side effects) where the right-hand side f^ρ_x of $x \in \mathcal{X}^\sharp$ is given by

$$f^\rho_x\,\sigma'' = \mathsf{bind}\,(f^\sharp{}_x\,\sigma''\,(\mathbf{fun}\ \_\ \_ \to \mathsf{return}\,()))\,(\mathbf{fun}\ b \to \mathsf{return}\,(\rho\,x \sqcup b))$$

for $\sigma'' : \mathcal{X} \to \mathcal{M}(\mathbb{D})$. This means that all side effects are now ignored, while instead the returned values take the contributions of ρ into account. For the elaborated right-hand side for x and $\sigma : \mathcal{X} \to \mathbb{D}$, this means that

$$\begin{aligned} F^\rho_x\,\sigma' = \mathbf{let}\ d &= \rho\,x \sqcup (F^\sharp{}_x\,\sigma')_1\ \mathbf{in} \\ \mathbf{let}\ X &= (F^\sharp{}_x\,\sigma')_2\ \mathbf{in} \\ &(d, X) \end{aligned}$$

Let $\underline{\sigma}^\rho$ denote the least solution of the lower monotonization of $\mathcal{E}^\rho$. Then, we replace condition (2.b) for $(\sigma, dom^\sharp)$ with the condition

(2.b′) $\underline{\sigma}^\rho\,x \sqsubseteq \sigma\,x$ for all $x \in dom^\sharp$.

In this case, we call $(\sigma, dom^\sharp)$ an *improved* partial post-condition. The significance of that notion is provided by the next proposition.

In the following, we assume that the concrete system $\mathcal{E}$ of equations may have side effects, but is monotonic. In that case, it also has for each subset X of unknowns, a unique least partial solution (σ, dom) with $X \subseteq dom$. For the following proposition, we make the reasonable assumption that the abstract value $\bot$ only describes an *empty* set of concrete states, that is, $\gamma\,\bot = \emptyset$. In an earlier version of this paper, we additionally introduced the restriction that each concrete unknown x is described by at most one unknown $x^\sharp$ in the abstract. This property, for example, is met for unknowns representing global variables such as g and h in Example 14 – but is violated when partial contexts are used. Removal of the given restriction is now possible due to the parametrization

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

of the description relation $\mathcal{R}$, that is, the (perhaps) surprising observation that the description relation may not be given before-hand – but is calculated by the analysis itself. We have:

Proposition 4. *Assume that* $(\sigma^\sharp, dom^\sharp)$ *is an improved partial post-solution of* $\mathcal{E}^\sharp$ *with* $X^\sharp \subseteq dom^\sharp$, *and that* $\mathcal{E}\ \mathcal{R}_\rho\ \mathcal{E}^\sharp$ *holds and* $X\ \mathcal{R}_\rho\ X^\sharp$ *for some set* X *of concrete unknowns. Assume further that* (σ, dom) *is the least partial solution of the concrete system with* $X \subseteq dom$, *Then* $(\sigma, dom)\ \mathcal{R}_\rho\ (\sigma^\sharp, dom^\sharp)$ *holds.*

Proof. Let $\rho = \bigsqcup\{(F^\sharp_y\, \sigma^\sharp)_3 \mid y \in dom^\sharp\}$ denote the accumulated side effect corresponding to $\sigma^\sharp$. Let $\underline{\sigma}^\rho$ denote the least solution of the lower monotonization $\underline{\mathcal{E}}^\rho$. As in the case without side effects, we proceed by ordinal induction. For each ordinal ι, we define the corresponding approximation $(\sigma_\iota, dom_\iota)$ of (σ, dom) by:

- $\sigma_0\, x = \emptyset$ for all $x \in \mathcal{X}$ and $dom_0 = X$;
- For each successor ordinal $\iota' = \iota + 1$,

$$\begin{aligned} \sigma_{\iota'}\, y &= (F_y\, \sigma_\iota)_1 \cup \{(F_z\, \sigma_\iota)_3\, y \mid z \in dom_\iota\} \\ dom_{\iota'} &= dom_\iota \cup \bigcup\{(F_y\, \sigma_\iota)_2 \mid y \in dom_\iota\} \cup \\ &\qquad \{y \mid z \in dom_\iota, (F_z\, \sigma_\iota)_3\, y \neq \emptyset\} \end{aligned}$$

- For each limit ordinal ι', $\sigma_{\iota'}\, y = \bigcup\{\sigma_\iota\, y \mid \iota < \iota'\}$, and $dom_{\iota'} = \bigcup\{dom_\iota \mid \iota < \iota'\}$.

where generally, $\sigma_{\iota'}\, y = \emptyset$ for all $y \notin dom_{\iota'}$. Our goal is to prove for each ordinal ι', that $(\sigma_{\iota'}, dom_{\iota'})\ \mathcal{R}_\rho\ (\sigma^\sharp, dom^\sharp)$ holds. This assertion clearly holds for $\iota' = 0$, and also for each limit ordinal ι', if it holds for each ordinal $\iota < \iota'$. Therefore, it remains to consider the case of a successor ordinal $\iota' = \iota + 1$. Then, we have by inductive hypothesis, for each $y \in dom_\iota$, there exists some $y^\sharp \in dom^\sharp$ with $y\ \mathcal{R}_\rho\ y^\sharp$, and also $\sigma_\iota\ \mathcal{R}_\rho\ \sigma^\sharp$. Since $\mathcal{E}\ \mathcal{R}_\rho\ \mathcal{E}^\sharp$, we have that for each such $y^\sharp$,

$$(F_y\, \sigma_\iota)_1 \subseteq \gamma((F^\sharp_{y^\sharp}\, \sigma^\sharp)_1)$$

and thus likewise,

$$\begin{aligned} (F_y\, \sigma_\iota)_1 &\subseteq \gamma((\underline{F}^\sharp_{y^\sharp}\, \sigma^\sharp)_1) \\ &\subseteq \gamma(\underline{\sigma}^\rho\, y^\sharp) \\ &\subseteq \gamma(\underline{\sigma}^\sharp\, y^\sharp) \end{aligned}$$

furthermore for unknowns $z \in dom_\iota$ and $z^\sharp \in dom^\sharp$ with $z\ \mathcal{R}_\rho\ z^\sharp$,

$$\begin{aligned} (F_z\, \sigma_\iota)_3\, y &\subseteq \gamma(\rho\, y^\sharp) \\ &\subseteq \gamma(\underline{\sigma}^\rho\, y^\sharp) \\ &\subseteq \gamma(\sigma^\sharp\, y^\sharp) \end{aligned}$$

Altogether therefore,

$$\sigma_{\iota'}\, y \subseteq \gamma(\sigma^\sharp\, y^\sharp)$$

which we wanted to prove. It remains to prove that also $dom_{\iota'}\ \mathcal{R}_\rho\ dom^\sharp$ holds. By induction hypothesis for ι, $dom_\iota\ \mathcal{R}_\rho\ dom^\sharp$ holds. Since $\mathcal{E}\ \mathcal{R}_\rho\ \mathcal{E}^\sharp$, also $(F_y\, \sigma_\iota)_2\ \mathcal{R}_\rho\ (F^\sharp_{y^\sharp}\, \sigma^\sharp)_2$ holds. Moreover, $(F_z\, \sigma_\iota)_3\, y \neq \emptyset$ implies that $\rho\, y^\sharp \neq \bot$ whenever $y\ \mathcal{R}_\rho\ y^\sharp$ holds. But then there must be some $z^\sharp$ with $(F^\sharp_{z^\sharp}\, \sigma^\sharp)_3\, y^\sharp \neq \bot$. Accordingly, $y^\sharp \in dom^\sharp$. This completes the proof. □

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

In light of Proposition 4, it therefore suffices to design algorithms for computing improved partial post-solutions of the abstract system starting from a given set of abstract unknowns. For that, we assume that the abstract system $\mathcal{E}^\sharp$ provides us for each abstract unknown $x^\sharp \in \mathcal{X}^\sharp$, with a right-hand side function $f^\sharp_{x^\sharp}$ of type $(\mathcal{X} \to \mathcal{M}(\mathbb{D})) \to (\mathcal{X} \to \mathbb{D} \to \mathcal{M}(\bullet)) \to \mathcal{M}(\mathbb{D})$ for every monad $\mathcal{M}$. As before, the solver state is maintained in mutable data structures. Therefore, the algorithm assumes right-hand side functions to have the OCaml type $(\mathcal{X} \to \mathbb{D}) \to (\mathcal{X} \to \mathbb{D} \to \bullet) \to \mathbb{D}$. As an extension of $\mathbf{TD}_{\mathbf{term}}$, we introduce the solver $\mathbf{TD}_{\mathbf{side}}$. In order to decide whether or not the different side effects to a leaf unknown y should be combined by the join operator $\sqcup$ or widening, the solver now maintains an additional data structure to record for each unknown y the set of unknowns from which it has received a side effect.

```
let sides = Map.create Set.create

let rec destabilize x = (* ... *)

let rec eval x y = (* ... *)

and side x y d =
  let op = if y ∈ point then ∇ else ⊔ in
  let tmp = op (!σ y) d in
  stable += y;
  if !σ y ≠ tmp then (
    σ, y := tmp;
    destabilize y;
    if x ∈ !sides y then point += y
    else !sides y += x
  )

and solve p x =
  if x ∉ stable && x ∉ called then (
    stable += x;
    called += x;
    let tmp = f♯x (eval x) (side x) in
    called -= x;
    let tmp =
      if x ∈ point then p (!σ x) tmp
      else tmp
    in
    if x ∉ stable then solve ∇ x
    else if !σ x = tmp then
      if p = ∇ && x ∈ point then (
        stable -= x;
        solve Δ x
      ) else ()
    else (
      σ, x := tmp;
      let _ = destabilize x in
      solve p x
    )
  )

let solve x = solve ∇ x; (σ, stable)
```

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

The functions destabilize and eval have not been changed. New, however, is the function side. A call side $x\ y\ d$ realizes the side effect from the unknown x onto the unknown y by combining the old value for y in σ with the new contribution d. If y is marked as a widening point, the value is combined using the widening operator, otherwise using the join operator. Generally, y is marked as stable. If the result is different from the old value for y, then the value of y in σ is updated, and all influenced unknowns are destabilized by means of the call destabilize y.

In order to decide whether y should be included into the set *point*, the algorithm maintains for each (leaf) unknown y' the set *sides* y' of all unknowns whose evaluations so far have led to an *increase* of the value of y'. Thus, if the current contribution d is not subsumed by $\sigma\ y$ and x is already contained in *sides* y, then y is added to the set *point*.

We remark that the call side $x\ y\ d$ may update the value of the leaf unknown y, but does not itself call the procedure solve. Accordingly, it removes those unknowns from *stable* whose latest values have been computed based on the out-dated assumption on y.

Finally, the main function solve is adapted to take side effects into account. This means that the evaluation of the right-hand side for an unknown x must now take the partial application side x as second argument. In presence of side effects, we may no longer assume that, after evaluation of $f^\sharp{}_x$ (eval x) (side x), the left-hand side x is necessarily contained in *stable*. Destabilization of x could only have been caused due to some side effect during solving of subsequent unknowns. In this case, we call solve $\nabla\ x$ – no matter whether we had already reached the narrowing iteration for x or not.

Example 17. Consider, for example, the abstract equation system from Example 16 for the program from Example 14.

Starting $\mathbf{TD}_{\mathbf{side}}$ for the unknown main in an initial solver state s_0 where all data structures are empty, will first produce side effects to g, h and record that increasing side effects have occurred from main. Then the evaluation of the unknown f is triggered.

Evaluating the computation tree t_f will query g (which is already found stable) and result in another side effect to h. None of these unknowns so far is put into the set *point*.

Continuing with the evaluation of the computation tree t_{main} will produce another side effect onto g. Since the value of g in σ is again modified, g is now put into the set *point*. Furthermore, destabilize g is called – which will remove f as well as main from the set *stable*. At that point, main is still contained in *called*. This implies that solve ∇ main is called again.

The second round of solving with ∇, though, will only update the values of h (recording at h an increasing contribution from f) and main in σ to $[0, 1]$ and $[0, 0]$, respectively, while leaving all unknowns in *stable*. Also, the subsequent call destabilize main will not remove main from *stable*, and the iteration terminates with

$$\sigma = \{g \mapsto [0, 1], h \mapsto [0, 1], f \mapsto [0, 0], \mathsf{main} \mapsto [0, 0]\}$$

□

We remark that in the preliminary version of the solver $\mathbf{TD}_{\mathbf{side}}$ in Seidl and Vogler (2018), the fresh values for globals are always combined with the corresponding old values by means of widening. This has been improved in the present version of the solver where widening for globals is restricted only to those globals which have been put into *point*.

In order to reason about termination and soundness of $\mathbf{TD}_{\mathbf{side}}$, we extend the notion of *consistency* as introduced in Section 5 appropriately. We now call a solver state $s = (\sigma, \mathit{infl}, \mathit{sides}, \mathit{stable}, \mathit{called}, \mathit{point})$ *consistent* if

1. for all $x \in (\mathit{stable} \setminus \mathit{called})$,
   (a) for all $y \in (F^\sharp{}_x\sigma)_2$, $y \in (\mathit{stable} \cup \mathit{called})$ and $x \in \mathit{infl}\ y$ holds;
   (b) for all y with $(F^\sharp{}_x\sigma)_3\ y \neq \bot$, $(F^\sharp{}_x\sigma)_3\ y \sqsubseteq \sigma\ y$;
2. for each $x \notin (\mathit{stable} \cup \mathit{called})$, $\mathit{infl}\ x = \emptyset$.

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

With this modified definition, the invariant for destabilize from Section 5 essentially remains the same, while the invariants for solve and eval must be refined. Each call solve $p\,x$ encountered during the evaluation of the initial call, starts in a consistent solver state $s = (\sigma, \mathit{infl}, \mathit{sides}, \mathit{stable}, \mathit{called}, \mathit{point})$. Upon its termination, a solver state $s' = (\sigma', \mathit{infl}', \mathit{sides}', \mathit{stable}', \mathit{called}', \mathit{point}')$ is attained such that $s = s'$ when $x \in \mathit{stable} \cup \mathit{called}$. If $x \notin \mathit{called} \cup \mathit{stable}$, the following holds:

1. s' is again consistent;
2. $\mathit{called}' = \mathit{called}$ where $\sigma'\ y = \sigma\ y$ for all $y \in \mathit{called}$;
3. $\mathit{sides}\ y \subseteq \mathit{sides}'\ y$ for all y, and $\mathit{point} \subseteq \mathit{point}'$;
4. $x \in \mathit{stable}'$.

In particular, the set of stable unknowns is not necessarily monotonically increasing.

Each call eval $x\,y$ encountered during the evaluation, starts in a consistent state $s = (\sigma, \mathit{infl}, \mathit{sides}, \mathit{stable}, \mathit{called}, \mathit{point})$ where $x \in \mathit{called}$ (no longer necessarily also in stable). Upon its termination, a solver state $s' = (\sigma', \mathit{infl}', \mathit{sides}', \mathit{stable}', \mathit{called}', \mathit{point}')$ is attained such that

1. s' is again consistent;
2. $\mathit{called}' = \mathit{called}$ where $\sigma'\ y' = \sigma\ y'$ for all $y' \in \mathit{called}$;
3. $\mathit{sides}\ y' \subseteq \mathit{sides}'\ y'$ for all y', and $\mathit{point} \subseteq \mathit{point}'$;
4. If $y \in \mathit{stable} \cup \mathit{called}$, then
   - $\sigma' = \sigma$, $\mathit{sides}' = \mathit{sides}$, $\mathit{stable}' = \mathit{stable}$, and $\mathit{point}' = \mathit{point}$ if $y \notin \mathit{called}$ whereas $\mathit{point}' = \mathit{point} \cup \{y\}$ if $y \in \mathit{called}$;
   - $\mathit{infl}'\ y' = \mathit{infl}\ y'$ for all $y' \neq y$, and
   - $\mathit{infl}'\ y = \mathit{infl}\ y \cup \{x\}$;
5. $y \in \mathit{stable}'$, $x \in \mathit{infl}'\ y$, and the value $\sigma'\ y$ is returned.

In particular upon termination of eval $x\,y$, we are also no longer guaranteed that $x \in \mathit{stable}'$.

Each call side $x\,y\,d$ encountered during the evaluation, starts in a consistent state $s = (\sigma, \mathit{infl}, \mathit{sides}, \mathit{stable}, \mathit{called}, \mathit{point})$ where $x \in \mathit{called}$. Upon its termination, a solver state $s' = (\sigma', \mathit{infl}', \mathit{sides}', \mathit{stable}', \mathit{called}', \mathit{point}')$ is attained such that $s = s'$ whenever $d \sqsubseteq \sigma\ y$. Otherwise,

1. s' is again consistent where $\mathit{called}' = \mathit{called}$;
2. $\mathit{sides}'\ y = \mathit{sides}\ y \cup \{x\}$, and $\mathit{sides}\ y' = \mathit{sides}'\ y'$ for all $y' \neq y$;
3. If $x \in \mathit{sides}\ y$ then $\mathit{point}' = \mathit{point} \cup \{y\}$, and $\mathit{point}' = \mathit{point}$ otherwise;
4. $\sigma'\ y \sqsupseteq \sigma\ y \sqcup d$, and $\sigma\ y' = \sigma'\ y'$ for all $y' \neq y$;
5. $y \in \mathit{stable}'$ where infl' and stable' are obtained from infl and stable by destabilizing y, that is, for all y', $\mathit{infl}'\ y'$ either equals $\emptyset$ or $\mathit{infl}\ y'$ where infl' and stable' are maximal so that s' is consistent while $\mathit{infl}'\ y = \emptyset$ and $\mathit{infl}\ y \cap \mathit{stable}' = \emptyset$.

Given that the number of unknowns encountered during the evaluation of the right-hand side is finite, let us first assume that only finitely many calls side $x\,y\,d$ ever will result in a modification of σ. Once within the iteration on some x, σ is no longer modified due to side effects; however, the same invariants as for the non-side-effecting solver $\mathbf{TD_{term}}$ apply – allowing us thus to deduce that each encountered call to solve will necessarily terminate. Now assume for a contradiction, that for some consistent solver state, the evaluation of solve $p\,x$ results in an infinite number of updates to leaf unknowns, say from set G. From some point on then all leaf unknowns y which have been added to point will not change anymore. In particular, none of the unknowns from G has been added to point. Then, there must exist some unknown x' and some unknown $g \in G$ so that an infinite sequence of calls solve $p_i\,x'$ is encountered for consistent solver states s_i immediately

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

before these calls where during the evaluation of the right-hand side of x' in each of these calls, a side effect to g occurs which results in a change to the value of g. After the first call, however, x' necessarily will be contained in the set sides y, that is, is contained in sides y at states s_i for all $i \geq 2$. This means that g is contained in the set *point* after the second call – in contradiction to our assumption. We therefore obtain:

Theorem 5. *Let $\mathcal{E}^\sharp$ denote an arbitrary system of abstract equations, and $x_0 \in \mathcal{X}^\sharp$ is the unknown of interest. Assume that initially the sets called and stable are empty, and likewise, infl maps each unknown to the empty set. Then, the call* solve x_0 *of solver* $\mathbf{TD_{side}}$ *will always terminate, as long as only finitely many unknowns are encountered.* □

Likewise, we can adapt the proof of Theorem 2 for the case of no side effects to obtain:

Theorem 6. *Each call* solve ∇ x *encountered during the evaluation of the call* solve x_0 *starting in a saturated solver state $s = (\sigma, infl, stable, called, point)$, results upon termination, in a solver state $s' = (\sigma', infl', stable', called', point')$ which is again saturated where additionally $x \in stable'$.*

The proof is based on a variant of the proof of Theorem 2. Taking into account that from some point in evaluation on, only calls solve p x occur where σ y does no longer change for any encountered unknown $y \in leaf$ via side effects, we obtain as a corollary:

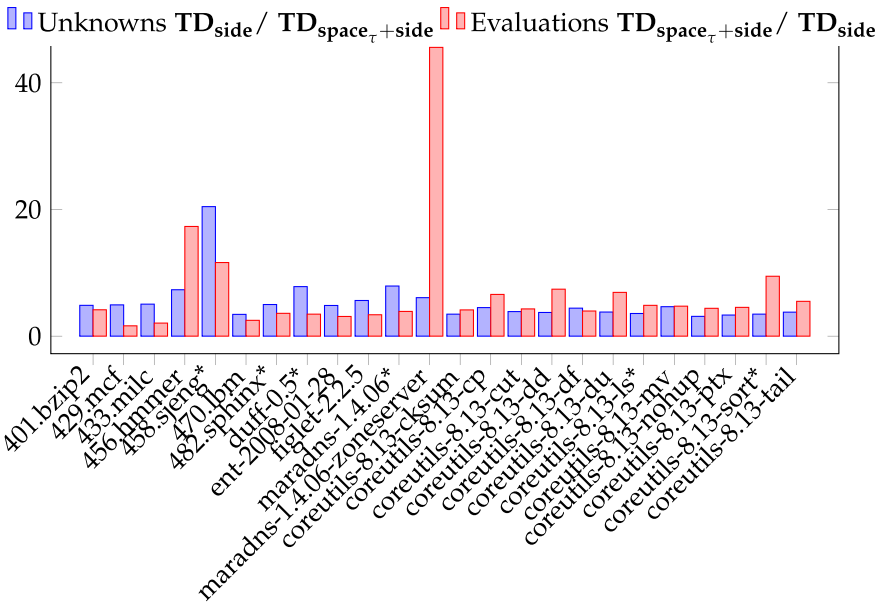
Corollary 5. *Assume that $\mathcal{E}^\sharp$ is a system of abstract equations with side effects and x_0 is an unknown of $\mathcal{E}^\sharp$. Assume that the sets stable and called are empty, and infl maps each unknown to Ø. Assume that the top-level call* solve x_0 *is evaluated by* $\mathbf{TD_{side}}$ *for $\mathcal{E}^\sharp$. Let σ and stable denote the values of these data structures after termination. Then $x_0 \in stable$, and $(\sigma, stable)$ is an improved partial post-solution of $\mathcal{E}^\sharp$.*

10. Experimental Evaluation

We implemented the presented solvers within the analysis framework GOBLINT.[1] In particular, we also realized a solver $\mathbf{TD_{space_\tau+side}}$ which combines the optimization for space with side effects (see Appendix C for this solver). The solvers were evaluated on the SPECint benchmark suite[2] consisting of not too small real-world C programs (1600–34,000 LOC after preprocessing). The programs 433.milc, 470.lbm, and 482.sphinx are part of the CFP2006 benchmark suite.[3] Furthermore, the following C programs were analyzed: duff-0.5,[4] ent-2008-01-28,[5] figlet-2.2.5,[6] maradns-1.4.06,[7] wget-1.12,[8] and some programs from the coreutils[9] package. The analyzed program wget-1.12 is the largest one with around 77,000 LOC. However, lines of code is not a good metric for complexity since there might be a lot of unused definitions from header files, and on the other hand many revisited lines due to loops and function calls. The most complex program by number of unknowns is 458.sjeng with 322,321 unknowns but only 17,336 LOC.

On top of a basic analysis of pointers and strings, we put an interval analysis of integer variables. By this, we chose the simplest meaningful setup where widening and narrowing is required. Clearly, at the expense of worse scalability, more complicated abstract domains could be tried within the same analysis framework. The benchmark programs were analyzed with full context-sensitivity of local data while globals were treated flow-insensitively. For programs with recursive function calls, the functions will be analyzed for more and more contexts which may lead to excessively long analysis times, stack overflows, or exhaustion of memory. We added an extra option that keeps the contexts for currently called functions for each unknown and widens the context for recursive calls. This can be seen as an abstraction of the call stack with a partial map from function names to abstract calling contexts. This partial map itself is not included into the context, but propagated via side effect to the entry points of the called functions. The analysis of the following programs only terminated with this widening on contexts enabled (these are suffixed

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press



The blue bars (resp. red bars) depict the ratio of required unknowns (resp. evaluations of right-hand sides) of the solver $\mathbf{TD_{side}}$ compared to the solver $\mathbf{TD_{space_\tau+side}}$.

Figure 3. Performance of $\mathbf{TD_{space_\tau+side}}$ vs. $\mathbf{TD_{side}}$.

with * below in figures and text): 458.sjeng, 482.sphinx, duff-0.5, maradns-1.4.06, coreutils-8.13-ls, coreutils-8.13-sort. The analysis of the following programs did not terminate within the given timeout of eight hours: 400.perlbench, 445.gobmk, wget-1.12. Our hypothesis is that widening on contexts for these programs still results in too many intermediate contexts. All benchmarks ran with an increased stack space of 48 MB (`ulimit -Ss 49152`).

Figure 3 compares the solvers $\mathbf{TD_{space_\tau+side}}$ and $\mathbf{TD_{side}}$ in terms of space and time. As a metric for space, we choose the total number of unknowns (i.e., occurring pairs of program points and contexts), and for time the total number of evaluations of right-hand sides of corresponding unknowns. Solver $\mathbf{TD_{side}}$ requires more than four times as many unknowns as the solver $\mathbf{TD_{space_\tau+side}}$. As expected, the price to be paid by solver $\mathbf{TD_{space_\tau+side}}$ for the fewer unknowns is an increase in the number of evaluations of right-hand sides. In practice, the CPU times and memory usage are of interest. For our experiments, we used an Intel Xeon E3–1270 v3 (3.50 GHz) with 32 GB of RAM.

Table 1 shows the number of unknowns, evaluations, the CPU time, and peak memory usage for $\mathbf{TD_{side}}$ (left columns) and $\mathbf{TD_{space_\tau+side}}$ (right columns). The maximum values for each column are bold. In the benchmark column, coreutils has been shortened to cu, as well as zoneserver to zs. The run times roughly correlate with the number of required evaluations of right-hand sides. Concerning memory usage, $\mathbf{TD_{space_\tau+side}}$ was 69% of $\mathbf{TD_{side}}$ on average with widening of contexts enabled, and 92% for benchmarks that ran without widening of contexts. The minimum memory usage was around 29 MB which can be seen as the overhead of the rest of the analyzer and the garbage collector. The memory saving gets more noticeable with an increasing number of unknowns. The biggest benchmarks 458.sjeng* and maradns-1.4.06* only consumed 43% and 69% of memory compared to $\mathbf{TD_{side}}$. We have not profiled how efficiently intermediate values

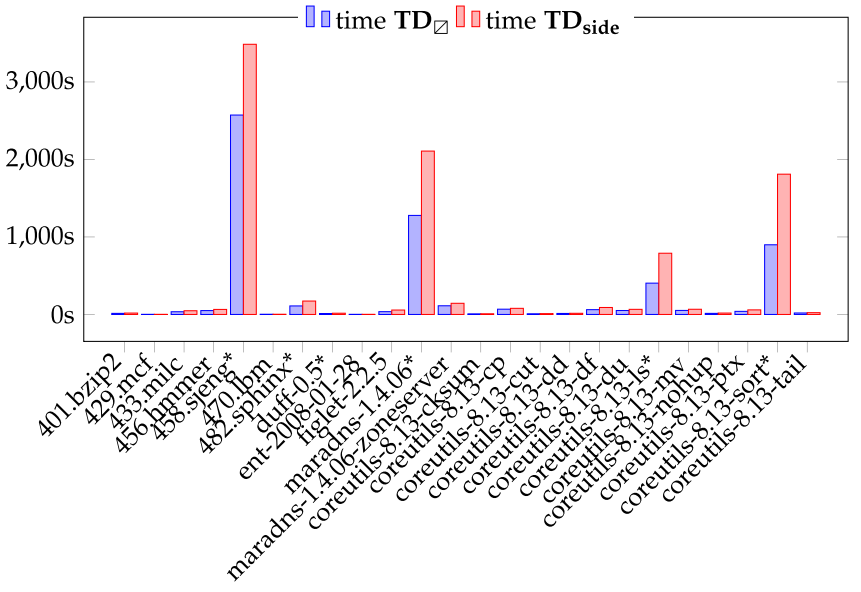
https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

Table 1. Performance of $\mathbf{TD}_{\mathbf{side}}$ and $\mathbf{TD}_{\mathbf{space}_\tau+\mathbf{side}}$

Benchmark	Unknowns		Evaluations		CPU time		Max. memory (kB)	
401.bzip2	7862	1616	38,688	161,058	18s	54s	112,212	80,472
429.mcf	1170	237	4806	7832	1s	1s	35,076	31,880
433.milc	27,701	5482	168,806	347,940	48s	1m23s	250,736	219,508
456.hmmer	59,244	8085	169,152	2,929,793	1m05s	12m41s	304,312	286,580
458.sjeng*	**322,321**	15,760	**3,875,316**	**45,040,785**	**1h04m50s**	**5h52m07s**	**4,187,152**	**1,807,292**
470.lbm	760	221	4702	11,745	2s	4s	35,340	31,812
482.sphinx*	34,508	6910	285,221	1,027,653	2m49s	5m52s	389,660	174,352
duff-0.5*	5459	698	36,617	127,181	14s	42s	82,584	56,244
ent-2008-01-28	402	83	1581	4916	0s	1s	30,808	30,024
figlet-2.2.5	9826	1745	120,985	408,616	55s	2m53s	181,564	161,036
maradns-1.4.06*	112,183	14,176	2,903,539	11,338,976	38m56s	2h14m34s	1,820,228	1,257,744
maradns-1.4.06-zs	77,952	12,828	237,864	10,845,779	2m09s	52m04s	337,788	550,736
cu-8.13-cksum	4166	1200	15,215	63,015	9s	23s	124,292	122,900
cu-8.13-cp	43,701	9694	131,799	868,455	1m23s	5m25s	338,472	252,744
cu-8.13-cut	4786	1234	19,170	82,416	10s	29s	139,472	122,944
cu-8.13-dd	10,310	2757	30,999	229,873	16s	1m10s	154,840	154,320
cu-8.13-df	20,065	4540	129,465	513,894	1m32s	4m59s	272,676	196,580

Table 1. Continued

Benchmark	Unknowns		Evaluations		CPU time		Max. memory (kB)	
cu-8.13-du	30,526	8229	133,769	1,037,280	1m04s	5m56s	293,188	267,896
cu-8.13-ls*	60,075	16,792	824,927	4,013,155	13m31s	38m25s	927,208	958,180
cu-8.13-mv	32,992	7094	117,967	558,431	1m10s	3m29s	304,276	251,536
cu-8.13-nohup	10,316	3301	34,889	153,365	18s	50s	139,700	137,732
cu-8.13-ptx	22,214	6667	115,046	522,881	58s	3m10s	250,500	196,868
cu-8.13-sort*	90,273	**25,972**	2,037,968	19,274,196	30m19s	2h46m02s	1,237,384	1,079,776
cu-8.13-tail	15,957	4211	49,279	270,897	23s	1m28s	173,916	155,124



The blue bars depict the run times for the solver $\mathbf{TD}_{\boxslash}$, the red bars for $\mathbf{TD}_{\mathbf{side}}$.

Figure 4. Absolute run times of $\mathbf{TD}_{\boxslash}$ vs. $\mathbf{TD}_{\mathbf{side}}$.

can be freed by the garbage collector. The results without widening of contexts give an indication how $\mathbf{TD}_{\mathbf{space}_\tau+\mathbf{side}}$ could be useful for bigger benchmarks: for maradns-1.4.06* it calculated 25,755 widening points, whereas $\mathbf{TD}_{\mathbf{side}}$ only managed to calculate 4797 widening points before both failed with a stack overflow with 3.15 GB and 6.23 GB peak memory usage, respectively.

Another important question is how the $\mathbf{TD}_{\boxslash}$ solver, that is the version of the **TD** solver equipped with the warrowing operator as in Apinis et al. (2016) and enhanced with our novel treatment of side effects, compares with the solver $\mathbf{TD}_{\mathbf{side}}$ in terms of run time and precision. Interestingly, we found that $\mathbf{TD}_{\mathbf{side}}$ on average runs 25% longer and gives the same results, that is, no differences in precision. The run times of the two variants are shown in Figure 4 in absolute numbers as well as in Figure 5 in relation.

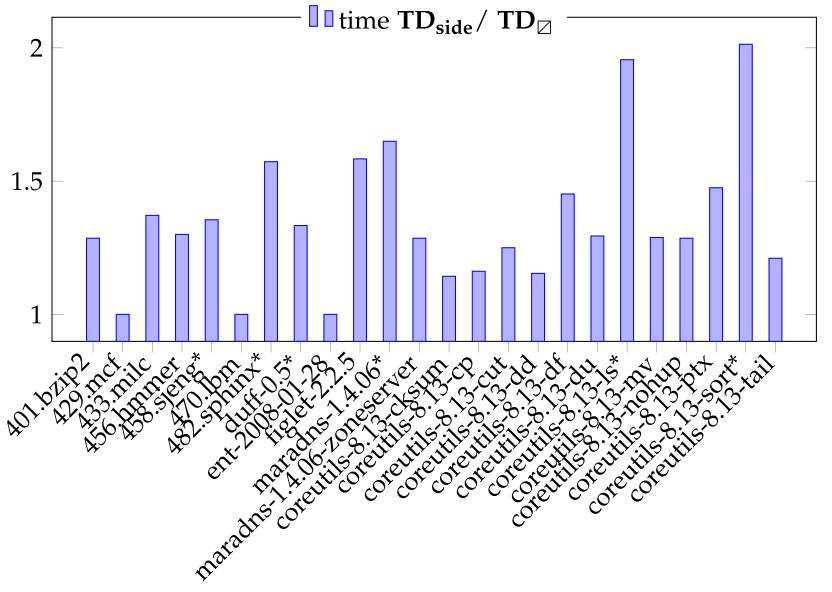
Finally, we also compared the impact of reluctantly widening for globals (as in the algorithm from the present paper) with default widening (as considered in Seidl and Vogler 2018). The run times of the two variants are the same on average. The detection of widening points for side effects leads to higher precision for globals compared to always widening (which was never more precise). The advantage in precision depends on the configuration of the analyzer as well as the analyzed program. 21 out of the 24 benchmarks had higher precision. Of those, the average increase of precision was 2.94%. The highest precision benefit was 25% for the benchmark 429.mcf.

All in all, we find that the optimization for space consumption had a significant beneficial impact on the practical behavior of the resulting solver – a factor of five in the number of unknowns may be a game changer for the overall space consumption, even at the expense of longer running times. On the given benchmarks, though, it did not result in different (practical) termination behavior.

11. Conclusion

The generic local solver **TD** from Hermenegildo and Muthukumar (1992), Charlier and Van Hentenryck (1992) is not suited to deal with complicated abstract domains with infinite

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press



The blue bars depict the run times of the solver $\mathbf{TD_{side}}$ relative to $\mathbf{TD_{\boxslash}}$.

Figure 5. Relative run times of $\mathbf{TD_{side}}$ over $\mathbf{TD_{\boxslash}}$.

strictly ascending and/or descending chains. Equipping the original **TD** with *warrowing* as in Apinis et al. (2016) on the other hand results in a solver which is only guaranteed to terminate for monotonic systems of abstract equations, and as is, provides no support for side-effecting. Therefore, we have provided three enhancements or additions. First, we considered extra program logic to conceptually guarantee termination for arbitrary systems of abstract equations – whenever only finitely many unknowns are encountered. We then showed how the self-monitoring capability of the solver can be used to reduce space consumption by only storing values at unknowns where also widening and narrowing should be applied. We finally indicated how these solvers can be extended to local generic solvers that operate on *side-effecting* systems of abstract equations. All three solvers could be proven to conceptually terminate (whenever only finitely many unknowns are encountered). The terminating variant could be proven sound by referring to the *lower monotonization* of the system of abstract equations. In the space-efficient version, that concept had to be complemented with an argument about consistent closures of mappings. In presence of side effects, we additionally had to collect all occurring side effects before-hand and apply the lower monotonization only then.

Compared to the preceding conference version of this paper (Seidl and Vogler 2018), we have elaborated the proofs considerably. In particular, we have provided detailed invariants for the solvers to hold, which can be formally verified by local reasoning over the code. We have also generalized the concept of description relations between concrete and abstract unknowns and illustrated the approach by meaningful examples. Finally, we have re-done the experimental evaluation in order to take the latest evolution of the analyzer GOBLINT into account, which has introduced a variety of improvements, for example, at the treatment of integer domains and conditions, and removed a series of subtle soundness bugs. The goal thereby was to pin-point the impact of design decisions such as using $\mathbf{TD_{\boxslash}}$ vs. the new one, the $\mathbf{TD_{side}}$ vs. $\mathbf{TD_{space_\tau+side}}$ or to what extent the novel technique of auto-detection of widening points at side-effected unknowns is preferable to default widening. Practically, the terminating solver with side effects as well as its space-efficient version turned out to be promising fixpoint engines for static analyzers based

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

on side-effecting systems of equations. Further experimentation is required to evaluate how well these solvers behave for advanced static analyses, for example, for complicated relational domains or more sophisticated analyses of dynamic data structures. It also remains for future research to explore in how far information gathered during the fixpoint iteration itself can systematically be used for further increasing either precision or efficiency of **TD** solvers.

Conflicts of interest

The authors declare none.

Notes

1 http://goblint.in.tum.de/
2 https://www.spec.org/cpu2006/CINT2006/
3 https://www.spec.org/cpu2006/CFP2006/
4 http://duff.dreda.org/
5 http://www.fourmilab.ch/random/ (version 28.01.2008).
6 http://www.figlet.org/
7 http://www.maradns.org/
8 https://www.gnu.org/s/wget/
9 https://www.gnu.org/s/coreutils/

References

Amato, G., Scozzari, F., Seidl, H., Apinis, K. and Vojdani, V. (2016). Efficiently intertwining widening and narrowing. *Science of Computer Programming* **120** 1–24.

Apinis, K., Seidl, H. and Vojdani, V. (2012). Side-effecting constraint systems: A swiss army knife for program analysis. In: Jhala, R. and Igarashi, A. (eds.) *Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11–13, 2012. Proceedings*, Lecture Notes in Computer Science, vol. 7705, Springer, 157–172.

Apinis, K., Seidl, H. and Vojdani, V. (2016). Enhancing top-down solving with widening and narrowing. In: Probst, C. W., Hankin, C. and Hansen, R. R. (eds.) *Semantics, Logics, and Calculi - Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays*, Lecture Notes in Computer Science, vol. 9560, Springer, 272–288.

Bourdoncle, F. (1993). Efficient chaotic iteration strategies with widenings. In: Bjørner, D., Broy, M. and Pottosin, I. V. (eds.) *Formal Methods in Programming and Their Applications, International Conference, Akademgorodok, Novosibirsk, Russia, June 28–July 2, 1993, Proceedings*, Lecture Notes in Computer Science, vol. 735, Springer, 128–141.

Bruynooghe, M., Janssens, G., Callebaut, A. and Demoen, B. (1987). Abstract interpretation: Towards the global optimization of prolog programs. In: *Proceedings of the 1987 Symposium on Logic Programming, San Francisco, California, USA, August 31–September 4, 1987*, IEEE-CS, 192–204.

Charlier, B. L. and Van Hentenryck, P. (1992). A universal top-down fixpoint algorithm. Technical report, Providence, RI, USA.

Cousot, P. (2015). Abstracting induction by extrapolation and interpolation. In: D'Souza, D., Lal, A. and Larsen, K. G. (eds.) *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12–14, 2015. Proceedings*, Lecture Notes in Computer Science, vol. 8931, Springer, 19–42.

Cousot, P. and Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R. M., Harrison, M. A. and Sethi, R. (eds.) *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, ACM, 238–252.

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

Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A. and Rival, X. (2009). Why does Astrée scale up? *Formal Methods in System Design* **35** (3) 229–264.

Fecht, C. and Seidl, H. (1999). A faster solver for general systems of equations. *Science of Computer Programming* **35** (2) 137–161.

Frielinghaus, S. S., Seidl, H. and Vogler, R. (2016). Enforcing termination of interprocedural analysis. In: Rival, X. (ed.) *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8–10, 2016, Proceedings*, Lecture Notes in Computer Science, vol. 9837, Springer, 447–468.

Gallagher, J. P. and Henriksen, K. S. (2006). Abstract interpretation of PIC programs through logic programming. In: *SCAM*, IEEE Computer Society, 184–196.
Hermenegildo, M. (2000). Parallelizing irregular and pointer-based computations automatically: Perspectives from logic and constraint programming. *Parallel Computing* **26** (13–14) 1685–1708.
Hermenegildo, M. V., Bueno, F., Carro, M., López-García, P., Mera, E., Morales, J. F. and Puebla, G. (2012). An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming* **12** (1–2) 219–252.
Hermenegildo, M. V., Puebla, G., Bueno, F. and López-García, P. (2005). Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). *Science of Computer Programming* **58** (1–2) 115–140.
Hermenegildo, M., Mendez-Lojo, M. and Navas, J. (2007). A flexible (C)LP-based approach to the analysis of object-oriented programs. In: *LOPSTR*, LNCS, vol. 4915, Springer, 154–168.
Hermenegildo, M. and Muthukumar, K. (1989). Determination of variable dependence information at compile-time through abstract interpretation. In: *North American Conference on Logic Programming*, MIT Press, 166–189.
Hermenegildo, M. and Muthukumar, K. (1992). Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming* **13** (2/3) 315–347.
Karbyshev, A. (2013). *Monadic Parametricity of Second-Order Functionals*. Phd thesis, Technical University Munich.
Lemieux, J. (2001). *Programming in the OSEK/VDX Environment*, CMP Media, Inc., USA.
Miné, A. (2001). The octagon abstract domain. In: Burd, E., Aiken, P. and Koschke, R. (eds.) *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01, Stuttgart, Germany, October 2–5, 2001*, IEEE Computer Society, 310.
Miné, A. (2012). Static analysis of run-time errors in embedded real-time parallel C programs. *Logical Methods in Computer Science* **8** (1) 1–63.
Miné, A. (2014). Relational thread-modular static value analysis by abstract interpretation. In: *VMCAI'14*, LNCS, vol. 8318, Springer, 39–58.
Muthukumar, K. and Hermenegildo, M. (1990). 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.
Seidl, H. and Vogler, R. (2017). Proving absence of starvation by means of abstract interpretation and model checking. In: D'Souza, D. and Narayan Kumar, K. (eds.) *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3–6, 2017, Proceedings*, Lecture Notes in Computer Science, vol. 10482, Springer, 3–22.
Seidl, H. and Vogler, R. (2018). Three improvements to the top-down solver. In: Sabel, D. and Thiemann, P. (eds.) *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03–05, 2018*, ACM, 21:1–21:14.
Vojdani, V., Apinis, K., Rõtov, V., Seidl, H., Vene, V. and Vogler, R. (2016). Static race detection for device drivers: The GOBLINT approach. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, ACM, 391–402.
Vojdani, V. and Vene, V. (2009). GOBLINT: Path-sensitive data race analysis. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eotvos Nominatae, Sectio Geologica* **30** 141–155.
Walli, S. R. (1995). The posix family of standards. *StandardView* **3** (1) 11–17.

Appendix A. The Original Solver TD

For a better comparison, we recall the original solver **TD** in the formulation of Fecht and Seidl (1999). This version is slightly more generic than the versions presented in Hermenegildo and Muthukumar (1992), Charlier and Van Hentenryck (1992) as it does not refer explicitly to partial tabulation of procedure summaries. In order to deal with non-monotonicity, the presented version performs an *accumulating* iteration, that is, updates maintained values of unknowns with the least upper bound of the respective old value and the new contribution. Side-effecting is not supported.

```
(* state / mutable data structures: *)
val σ      : (𝒳, 𝔻) Map.t
val infl   : (𝒳, 𝒳 Set.t) Map.t
```

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

```
val called : 𝒳 Set.t
val stable : 𝒳 Set.t
(* creation *)
val Set.create : unit -> 'a Set.t
val Map.create : (unit -> 'b) -> ('a,'b) Map.t
(* unary operator *)
val (!) : ('a,'b) Map.t -> 'a -> 'b
(* binary operators *)
val (:=) : ('a,'b) Map.t * 'a -> 'b -> unit
val (∈ ) : 'a -> 'a Set.t -> bool
val (+=) : 'a Set.t -> 'a -> unit
val (-=) : 'a Set.t -> 'a -> unit

let stable = Set.create ()
let called = Set.create ()
let infl   = Map.create Set.create
let σ      = Map.create (fun () -> ⊥)

let rec destabilize x =
  let w = !infl x in
  infl, x := Set.create ();
  Set.iter (fun y ->
      stable -= y;
      if y ∉ called then
        destabilize y
    ) w

let rec eval x y =
  solve y;
  !infl y += x;
  !σ y

and solve x =
  if x ∉ stable && x ∉ called then (
    stable += x;
    called += x;
    let tmp = !σ x ⊔ f♯x (eval x) in
    called -= x;
    in
    if !σ x = tmp then ()
    else (
      σ, x := tmp;
      destabilize x;
      solve x
    )
  )

let solve x = solve x; (σ,stable)
```

Similar to the solvers **TD**$_{\textbf{term}}$, **TD**$_{\textbf{space}}$ or **TD**$_{\textbf{space}_\tau+\textbf{side}}$, the algorithm starts with calling solve for a particular unknown x_0. By means of the two functions solve and eval, it descends into unknowns accessed during the evaluation of right-hand sides for recursive evaluation. Similar to what we still do, dependencies between unknowns are detected and recorded on-the-fly. Thereby, the function destabilize for removing unknowns, possibly affected by an update to the value of σ for an unknown, is identical in our algorithms.

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

It is mentioned in Charlier and Van Hentenryck (1992) that an extension of the algorithm with *widening* is possible. No mechanism, though, is provided for detecting widening points on-the-fly. Also, no intertwined *narrowing* iteration is introduced. Since no widening/narrowing points are at hand, it seems inevitable to use accumulating updates for all unknowns. Instead, our solvers apply such merging at widening/narrowing points only (and there then together with ∇ or Δ) – while at all other points, the old value in σ is *replaced* (or just recomputed).

Appendix B. The Solver $\mathbf{TD}_{\mathbf{space}}$ with Caching

The generic local solver $\mathbf{TD}_{\mathbf{space}_\tau}$ enhances solver $\mathbf{TD}_{\mathbf{space}}$ by additionally caching the values for all unknowns $y \notin \mathit{point}$ which intermediately have been computed during the evaluation of a right-hand side $f^\sharp_x$ for some unknown $x \in \mathit{point}$.

```
let rec destabilize x = (* ... *)

let rec eval x τ y =
  if y ∈ called then
    point += y;
  if y ∈ point then (
    solve ∇ y;
    !infl y += x;
    !σ y
  ) else if !τ y ≠ ⊥ then
    !τ y
  else (
    called += y;
    let tmp = f♯_y (eval x τ) in
    called -= y;
    if y ∉ point then (
      τ, y := tmp;
      tmp
    ) else (
      solve ∇ y;
      !infl y += x;
      !σ y
    )
  )

and solve p x =
  if x ∉ stable && x ∉ called then (
    stable += x;
    called += x;
    let τ = Map.create (fun () -> ⊥) in
    let tmp = f♯_x (eval x τ) in
    called -= x;
    let tmp = p (!σ x) tmp in
    if !σ x = tmp then
      if p = ∇ then (
        stable -= x;
        solve Δ x
      ) else ()
    else (
      σ, x := tmp;
      destabilize x;
      solve p x
    )
  )

let solve x = point += x;
              solve ∇ x;
              (σ, stable)
```

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

The central modification of solver $\mathbf{TD_{space}}$ is that a fresh mutable map τ is created before evaluation of the right-hand side $f^\sharp{}_x$ inside a call solve $p\,x$. That map then is passed to the function eval as an additional argument. In a call eval $x\,\tau\,y$, x is an unknown in *point* whose right-hand side is currently under evaluation; τ is the mutable map created in the surrounding call solve $p\,x$, and y is the unknown whose value is currently queried. The value of $\tau\,y$ is only queried when $y \notin called \cup point$. When $\tau\,y$ is found to be different from $\bot$, this value will be returned. Otherwise, evaluation of $f^\sharp{}_y$ (eval $x\,\tau$) proceeds as in $\mathbf{TD_{space}}$. If after evaluation, y still is not contained in *point*, the new value for y is recorded in τ and then returned.

Appendix C. The Combined Solver $\mathbf{TD_{space_\tau+side}}$

In this appendix, we finally put all ingredients into one generic solver.

```
(* state / mutable data structures: *)
val σ      : (𝒳, 𝔻) Map.t
val infl   : (𝒳, 𝒳 Set.t) Map.t
val sides  : (𝒳, 𝒳 Set.t) Map.t
val called : 𝒳 Set.t
val stable : 𝒳 Set.t
val point  : 𝒳 Set.t
(* creation *)
val Set.create : unit -> 'a Set.t
val Map.create : (unit -> 'b) -> ('a,'b) Map.t
(* unary operator *)
val (!) : ('a,'b) Map.t -> 'a -> 'b
(* binary operators *)
val (:=) : ('a,'b) Map.t * 'a -> 'b -> unit
val (∈ ) : 'a -> 'a Set.t -> bool
val (+=) : 'a Set.t -> 'a -> unit
val (-=) : 'a Set.t -> 'a -> unit

let stable = Set.create ()
let called = Set.create ()
let point  = Set.create ()
let infl   = Map.create Set.create
let sides  = Map.create Set.create
let σ      = Map.create (fun () -> ⊥)

let rec destabilize x =
  w = !infl x;
  infl, x := Set.create ();
  Set.iter (fun y ->
      stable -= y;
      if y ∉ called then destabilize y
    ) w

let rec eval x τ y =
  if y ∈ called then
    point += y;
  if y ∈ (point ∪ leaf) then (
    solve ∇ y;
    !infl y += x;
    !σ y
  ) else if !τ y ≠ ⊥ then
    !τ y
  else (
    called += y;
    let tmp = f♯ y (eval x τ) side in
    called -= y;
```

https://doi.org/10.1017/S0960129521000499 Published online by Cambridge University Press

```
    if y ∉ (point ∪ leaf) then (
      τ, y := tmp;
      tmp
    ) else (
      solve ∇ y;
      !infl y += x;
      !σ y
    )
  )

and side x y d =
  let op = if y ∈ point then ∇ else ⊔ in
  let tmp = op (!σ y) d in
  stable += y;
  if !σ y ≠ tmp then (
    σ, y := tmp;
    destabilize y;
    if x ∈ !sides y then point += y
    else !sides y += x
  )

and solve p x =
  if x ∉ stable && x ∉ called then (
    stable += x;
    called += x;
    let τ = Map.create (fun () -> ⊥) in
    let tmp = f♯x (eval x τ) (side x) in
    called -= x;
    let tmp = p (!σ x) tmp in
    if x ∉ stable then solve ∇ x
    else if !σ x = tmp then
      if p = ∇ && x ∈ point then (
        stable -= x;
        solve Δ x
      ) else ()
    else (
      σ, x := tmp;
      let _ = destabilize x in
      solve p x
    )
  )

let solve x = point += x;
              solve ∇ x;
              (σ, stable)
```

This solver now proceeds essentially as the solver $\mathbf{TD}_{\mathbf{space}_\tau}$ – but additionally takes side effects into account. This means in particular that unknowns from the set *leaf*, that is, those which may receive contributions via side effects, must now be treated by the function eval like unknowns from *point*.

Cite this article: Seidl H and Vogler R (2021). Three improvements to the top-down solver. *Mathematical Structures in Computer Science* **31**, 1090–1134. https://doi.org/10.1017/S0960129521000499