TECHNISCHE UNIVERSITÄT MÜNCHEN
Lehrstuhl für Informatik II

Frameworks for analyzing multi-threaded C

Kalmer Apinis

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Matthias Althoff
Prüfer der Dissertation:	1. Univ.-Prof. Dr. Helmut Seidl,
	2. Univ.-Prof. Dr. Dr. h.c. Javier Esparza

Die Dissertation wurde am 21.01.2014 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 23.05.2014 angenommen.

Contents

List of Original Publications

This dissertation includes the content of the following three publications:

1) Kalmer Apinis, Helmut Seidl, and Vesal Vojdani. Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis. In *APLAS*, pages 157–172. LNCS 7705, Springer, 2012.

2) Kalmer Apinis, Helmut Seidl, and Vesal Vojdani. How to combine widening and narrowing for non-monotonic systems of equations. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 377–386. ACM, 2013.

3) Gianluca Amato, Francesca Scozzari, Kalmer Apinis, Helmut Seidl, and Vesal Vojdani. Efficiently intertwining widening and narrowing. *(submitted for publication)*

1 Abstract

Suitable abstraction layers are required in order to build a generic analysis framework that is flexible enough to handle multi-threaded C. Practical analysis often requires the verification of a number of different safety properties [Vojdani, 2010] for which useful analyses might already be described in the literature. To accommodate reuse, a good abstraction, therefore, should allow for a convenient composition of individual analyses in a way that enables each analysis to benefit from information provided by the others. Additionally, configurable flow-, context-, and path-sensitivity is preferred as it extends the applicability of the analyzer to larger range of use cases. One such useful abstraction is describing the analysis as a *side-effecting constraint system* over complete lattices. We show that this abstraction is flexible and, also, useful in practice.

Side-effecting constraint systems were originally introduced in a restricted form for the analysis of multi-threaded code by Seidl, Vene, and Müller-Olm [2003]. In this thesis, I will explain how this formalism in its extended form provides a unified framework for realizing efficient interprocedural analyses of programs, possibly with dynamic function calls, where the amount of context-sensitivity can be adjusted and where the context-sensitive analyses of local properties can be combined with flow-insensitive analyses of global properties, e.g., about the heap.

One major issue for a constraint- or equation systems based approach is that many non-trivial analysis problems require complete lattices with infinite ascending and descending chains. In order to compute reasonably precise post-fixpoints of the resulting systems of equations, Cousot and Cousot have suggested to accelerate fixpoint iteration by means of widening and narrowing [Cousot and Cousot, 1977a, 1992]. The strict separation into a widening phase followed by a narrowing phase, however, may sacrifice precision which cannot be recovered later. Worse, a multi-phased approach does not lend itself to local solving. As a remedy, I present a novel operator $\boxminus$ that combines a given widening operator $\sqcup\!\!\!\!\!-$ with a given narrowing operator $\sqcap\!\!\!\!\!-$ which allows to merge the two phases.

Applying the new combined approach to off-the-shelf equation solvers, however, does not preserve the termination guarantees of the solvers as the combined operator is inherently non-monotonic. Adapted versions of round-robin as well as worklist iteration, local, and side-effecting solving algorithms are, therefore, presented for use with the combined operator. The key idea of each of the alteration is to assign priories to equation system variables and then solve variables according to these priories. The resulting solvers are proven to always return sound results and terminate for monotonic systems whenever only finitely many constraint variables are encountered and priorities are respected by side-effects. In addition, several optimizations, inspired by [Amato and Scozzari, 2013], are presented which limit the use of the widening operator or restart parts of the computation—therefore increasing precision, and, in some cases, performance.

Finally, I provide implementation details and report on the various software design decisions made while improving the Goblint tool [Vojdani and Vene, 2009; Seidl and Vojdani, 2009; Vojdani, 2010] with the ideas presented in this thesis. Most importantly, I elaborate on the software architecture of the Goblint and describe how the analysis specifications can be combined in such a way that each analysis can take advantage of the information computed by other analyses.

2 Introduction

Computers are used for a wide variety of tasks: searching in vast quantities of data, storing and providing multi-media content, steering partially autonomous robotic rovers on Mars, helping pilots to fly modern airplanes, etc. This lists will grow in the future as new technologies mature—allowing implementation of ideas, such as self-driving cars, that were previously in the realm of science-fiction.

For computers, as for all complicated systems, there exists a possibility of failure. The effects of the failure ranges from slight annoyance to serious bodily harm. It might be irritating to not being able to watch your favorite TV show on your smartphone; however, it might be disastrous if there is a disruption of service in a subsystem of a high speed train.

Sadly, disasters relating to computer failures do not only present a theoretical possibility, but rather a stark reality. In 2003, a widespread power outage occurred in parts of Northeastern and Midwestern United States and Ontario, Canada, leaving an estimated 50 million people without electricity. Power was fully restored after two days. A software bug was partially responsible for the incident by allowing certain alarms messages to go unnoticed [Poulsen, 2004]. The blackout happened approximately an hour after the first alarms were supposed to be triggered—leaving the power plant staff unaware of the real situation.

Another, classic, example of disastrous computer failure is related to the radiation therapy machine Therac-25. In six incidents between June 1985 and January 1987, massive overdoses of radiation were administered [Leveson and Turner, 1993]. Many flaws in the Therac-25 system were uncovered, including a software bug in the data entry subsystem of the radiation therapy machine. Namely, in the case when a user entered the configuration of the radiation procedure but then quickly changed the data, the final change was applied partially, leaving the system in an inconsistent state.

The most common method for improving the quality of computer programs and reducing the possibility of failure is *testing*. For a limited set of input parameters, the result produced by the program is compared against the expected result—either manually by a person, or automatically, using some testing framework.

Testing alone, however, cannot prove the absence of software bugs—at least in the case where only a subset of all possible input values are considered. For all other input values, inherently, testing cannot conclude anything. It might be feasible to test all parameters for a small deterministic program, but it is much more difficult for larger programs and programs that behave non-deterministically. For that reason, concurrent programs—programs that contain parts that may execute in parallel or allow for unknown scheduling—pose a difficult challenge for testing as concurrency can introduce non-determinism into the system. Both of the disastrous computer bugs, presented earlier, were in concurrent systems where the exact input parameters that were used in the incidents alone would not necessarily suffice to reproduce the problem.

We conclude that in many cases testing might be enough to achieve an acceptable level of quality, e.g., for an entertainment application on a smartphone. Nevertheless, for critical software, more is required. Several approaches to tackle this problem exist; in this thesis, however, we will look at techniques that fall under the general term of *static program analysis*, which, contrary to testing, computes safety invariants without executing the program itself. More specifically, we will focus on frameworks that are able to support the analysis of concurrent C programs.

To get some initial intuition about static program analysis, first we shall look at the example C program in Figure 2.1. The example program consists of two procedures: `action_A` and `action_B`, which may be triggered repeatedly and in an unknown combination—possibly in parallel by two or more *threads*. The procedure `action_A` performs three consecutive operations: first, it acquires a lock `m`, then increments a global variable `x`, and finally releases the lock `m`. The procedure `action_B` also acquires the same lock `m`, but instead decreases the variable `x`, though only in the case

```
void action_A(){                    void action_B(){
  lock(&m);                           lock(&m);
  x=x+1;                              if (x>0) x=x−1;
  unlock(&m);                         unlock(&m);
}                                   }
```

Figure 2.1: Fragment of a concurrent program.

when the variable is bigger than zero. As its last operation, action_B also releases the lock m. The locking procedures guarantee that each lock is held by a single thread at a time—in case several threads want to acquire the same lock, only one will succeed at first, while the others must wait until the lock is again released.

The following properties, among others, can be automatically concluded by using static program analysis:

a) accesses to the variable x occur only when the lock m is taken,

b) multiple action_B-s running in parallel will never interfere with each other, and

c) the variable x, when initialized with 0, will always stay non-negative.

Note that by removing the locking procedure calls from the example in Figure 2.1, all of the presented properties would not necessarily hold. It would be possible to decrease x below zero, if several action_B-s would execute in parallel in such a way that first all the checks "x<0" are performed, and only then all of the decrements.

To find such interesting properties of programs, however, it is sometimes enough to combine a number of "classical" analyses with each other, possibly in addition to some specialized analyses. The properties of the example program can be automatically derived with a combination of an interval-based value analysis and a must-lockset analysis. There is a critical interdependency between these analyses. Values of integer and pointer variables influence the control flow, and ultimately the lock-sets, while values of shared variables, as the example shows, are dependent on the lock-sets. Therefore, it is practically useful to design an analysis tool that allows flexible configuration and composition of analyses. Ideally, all analyses should be implemented in a single framework, where adding new (sub-)analyses should be straight-forward and combining of analyses should be automatic and without analysis-time overhead.

As a step towards the ideal analysis tool, we present an improved version of Goblint [Vojdani and Vene, 2009; Seidl and Vojdani, 2009; Vojdani, 2010]—a static analyzer tool for (concurrent) C programs, implemented in OCaml. This thesis includes the theoretical analysis framework that is realized in Goblint as well as a selection of interesting implementation details. In addition, the thesis provides an extensive overview of constraint system based static program analysis.

Main Contributions

The thesis consists of three major parts. In the first part, I show that constraint system based, and especially side-effecting constraint system based frameworks are flexible enough to support many interesting features—fully configurable flow- and context-sensitivity, generic combination of analyses, and forward propagation similar to the algorithms of Sharir and Pnueli [1981] instead of demand-driven iteration, as is usual for constraint systems.

In the second part, I propose a way to apply the techniques from the widening/narrowing approach, proposed by Cousot and Cousot [1976], in a setting where local solving of constraint systems is required—both for side-effecting and non-side-effecting constraint systems. This has the additional benefit of possibly increasing the precision of the analysis in comparison to the "classical" two-phased approach. Additionally, I adapt the optimizations from [Amato and Scozzari, 2013] to further improve the technique, significantly increasing the precision and, in some cases, improving performance, while still preserving termination guarantees.

In the third part, I report about implementation details and improvements to the Goblint framework that I have realized while researching the subject of this thesis. The most important implementation contribution is the generic mechanism of combining analyses in a way that allows analyses to reciprocally benefit from each other. All topics discussed in the third part of the thesis contain new contributions—with the exception of the general structure of Goblint, the individual analyses and their domains. Only minor improvements were made to path-sensitivity, dead-code lifting, and the module GlobConstrSystem that represents constraint systems—they are presented for the purpose of overview.

Side-effecting framework

The first part of the thesis starts with an introduction into static program analysis which is based on constraint system over complete lattices. Classical techniques for supporting the analysis of function calls are presented in the constraint system notation for overview. All necessary definitions are presented to allow reasoning about the correctness of the analysis. Partial tabulation is formulated using constraint systems that could then be solved using local solving. At this point we see that previous techniques have practical difficulties when reachability is directly encoded in the constraints, instead of in the solving algorithm. This is the case, for example, when only part of the starting state of a function is used as the context. We draw parallels to flow-insensitive analysis and show that extending constraints with side-effects [Seidl et al., 2003] can also be used to perform partially context sensitive analysis. We show that the more general formulation of side-effecting constraint system can be solved, by providing a local solving algorithm.

We show that side-effecting constraint systems are powerful enough to allow the encoding of dynamic procedure calls and forward propagation. Forward propagation is a technique to propagate abstract analysis values along control-flow graph edges, whereas local solving works on a constraint system and follows the dependencies of the constraints.

We provided experimental evaluation to compare forward propagation to local solving in three configurations: no calling context, partial calling contexts, and full calling context. The chapter ends with a conclusion that summarizes the consequences of using side-effecting constraint systems.

Adding Widening/Narrowing

The second part of the thesis deals with bridging the gap between constraint system based static program analysis and widening/narrowing based static program analysis. A widening (over-approximation) is used in cases where the height of the lattice is too large—a case where precise generic solvers may encounter non-termination. Later, narrowing may be used to regain some precision lost by over-approximation.

It is shown how constraint systems can be transformed into equation systems so they can be used in the context of widening/narrowing. A more general notion of a solution of an equation system is provided. Also, the classical two-phased approach to widening/narrowing is presented using equation systems.

A novel update operator $\boxminus$ is presented, which combines given widening and narrowing operators in a generic manner. Using the new operator, the two phases can be combined into one equation system. It is shown that the solution of the combined equation system is also a post-solution to the original system—allowing to apply any generic equation solver to attempt to find a solution.

Using examples, it is shown that termination of the solver depends on the order in which constraints are (re-)evaluated. Novel equation system solvers which are based on round-robin iteration and worklist iteration are presented. The new solvers choose constraints in a way that guarantees termination for finite and monotonic constraint systems.

A more generic definition of local solvers is introduced. A new solver is constructed from the solver **RLD** [Hofmann et al., 2010b] which terminates in the monotonic case where only finitely many constraint variables are encountered. As for the new round-robin and worklist solver, the new local solver can provide termination guarantees, intuitively, because the iteration adheres to a fixed linear ordering of the variables.

Next, optimization inspired by [Amato and Scozzari, 2013] are presented as variants of the new local solver. The first optimization identifies so called back-edges to detect potential dependency cycles in the equation system and applies the new update operator $\boxminus$ only for variables possessing outgoing back-edges. While the first optimization only allowed the set of back-edges to increase, the second optimization also allows the set to decrease. The third optimization, however, allows to restart the computation of a certain subset of variables in the case that the value of a variable decreased. The intuition for that is that the previously over-approximated value may have already been propagate to other variables where the effect cannot be undone using successive iteration alone. Restarting, however, may be able to improve the values of those variables.

Additionally, side-effecting versions of the new local solver are introduced with analogous guarantees for termination. For the termination guarantees to hold, however, the side-effecting constraints are expected to respect the ordering on variables, i.e., produce side-effects from a smaller priority variable to larger variables, only.

Experimental evaluation is done to assess the precision of the presented local solver. The precision of naive two-phased widening/narrowing is compared to the presented local solver. Additionally, the precision impact of the different optimizations to the solvers are examined.

The performance of the new local solvers are investigated by comparing the number of right-hand side evaluations needed in different solver configurations for the analysis of a selected set of benchmark programs. All variants of the presented local solver are compared against each other and, also, against the the configuration where only widening is preformed. The chapter is concluded with a summary.

Implementation details

The final major part of the thesis consists of an overview of the practical implementation details and improvements of the static analyzer Goblint. Starting with the basic structure of Goblint, each subsequent section focuses on a different component of the implementation.

The first practical improvement is the new configuration sub-system, which allows better handling of different analysis parameters. Technically, the configuration sub-system consists of a JSON data structure that can be manipulated in different ways.

Next, the structure of Goblint is presented in more detail—showing the most important module signatures and describing their meaning and relation to the previous chapters. In addition to textual description, the various important modules are described using code-snippets and diagrams.

A transformation is presented which adds central handling of unreachable code to any analysis specifications. Following that, another analysis transformation is presented which tries to minimize memory usage of the analysis by applying the software pattern of hash consing.

The intuition behind the query system is discussed: each analysis provides a function that answers queries based on their "local" knowledge and requests of recursive queries, and the analysis combining system binds the queries together in a way that allows a limited form of recursive queries.

The intuition behind the combination of analyses is presented along with the needed technical features from the OCaml 4.00 compiler. A small overview of path-sensitivity [Vojdani and Vene, 2009] is presented, together with some details of how it is handled by the new Goblint implementation. Finally, details are shown on how dynamic solver selection is implemented within a static solver module.

3 A Unified Framework for Program Analysis

Due to the complicated semantics of modern programming languages, analyzers inferring non-trivial program invariants require auxiliary analyses for many different properties. When checking multi-threaded C for absence of data-races, for example, one needs auxiliary analyses for disambiguating function pointers, may- and must-alias analysis for ordinary pointers, and if control-flow is to be tracked with higher precision, some form of value analysis is additionally required [Vojdani, 2010].

One choice, when combining various kinds of analyses, is to proceed in stages where later stages have access to the invariants previously computed. The advantage of the staged approach is that each stage has to deal with a small set of different concepts only and thus can be kept rather simple. The disadvantage, though, is that an unnecessary loss of precision may be incurred, since information only flows in one direction across stages. For example, call-graph construction for languages with function values must rely on information about those function values; however, a sound data flow analysis computing these values may already need to access the call graph [Grove and Chambers, 2001]. A common approach to resolve this circular dependency is to start with an over-approximation of the call-graph, and then rely on the pre-computed graph in subsequent stages for interprocedural analysis.

Thus, when precision is crucial, an integrated approach is preferred. This is the case in sound static analyzers, such as Astrée [Cousot et al., 2005] or Goblint. In these frameworks, the different analyses are joined into one global analysis which determines all required invariants in one go, so that the distinct analyses reciprocally benefit from one another. Additionally, Goblint allows the user to configure, for each analysis, whether it should run context-, path-sensitively, or not at all: the different analyses communicate through a query-system such that multiple analyses can complement each other in answering, e.g., aliasing queries. Such flexible integration is only possible, however, when the *algorithmics* of the different analyses harmonize.

The goal, therefore, is to abandon dedicated analysis algorithms and instead provide one specification formalism together with a single solver engine to compute the invariants. Proofs of soundness are then vastly simplified as the verification task is separated into proving the constraint system correct and independently proving the correctness of a generic fixpoint engine, along the lines of [Hofmann et al., 2010b].

We suggest that *side-effecting* constraint systems, introduced in [Seidl et al., 2003] for the analysis of multi-threaded code, is the ideal tool to achieve the desired harmonization. Intuitively, in each constraint of a side-effecting constraint system, the right-hand side does not only specify a sequence of reading accesses to some constraint variables, whose values are queried and used to provide a contribution to the variable on the left-hand side, but may additionally disperse write contributions to further constraint variables in-between.

The key contribution of this part of the thesis is to show that many analysis problems, whose solving seems to require different algorithms, can all be expressed using this single formalism. In particular, we show that this idea provides a uniform solution to the following interprocedural analysis problems:

a) tabulation of procedure summaries for parts of the calling context only, also in the presence of dynamic procedure calls;

b) integrated analysis which accumulates some of the data flow information flow-insensitively, while at the same time tracking other data, such as (an abstraction of) the local state, flow- as well as context-sensitively.

These problems can be expressed by ordinary constraint systems which thus may serve as a formal specification of the analysis problem. For non-trivial analyses, including constant propagation, these constraint systems are infinite. *Local* fix-point solvers, which only solve those variables that are required for the analysis, can be used to solve infinite systems. However, these constraints are not only infinite, but some variables of the constraint system may formally depend on infinitely many other variables. Therefore, they do not lend themselves to implementations by means of generic local solvers.

We show instead that these constraint systems can be reformulated by introducing side-effecting constraints. The side-effects are triggered during constraint solving and may depend on the values of other variables. Side-effecting constraints thus cannot generally be replaced by an equivalent constraint system with finite variable dependencies by factoring out side-effects as individual constraints. The reformulated constraint systems, however, can be solved efficiently by means of generic local solvers adapted to side-effecting constraints. A local solver will try to solve only variables that are required for the analysis. These adapted generic local solvers together with side-effecting constraint systems may thus serve as a Swiss army knife for efficient integrated whole-program analysis.

Related Work. The seminal paper by Kildall [1973] can already be interpreted as an attempt to provide a unifying framework for various program analysis techniques at his time. It did not incorporate, however, direct support for more advanced language features such as procedures or threads. The approach of abstract interpretation by Cousot and Cousot [1977a] not only provides the foundations for reasoning about the correctness, but also serves as the basis for a variety of program analyses which algorithmically translate into solving appropriate constraint systems [Cousot and Halbwachs, 1978] or directly interpreting the input program abstractly [Cousot et al., 2005]. As one instance, also a framework for analyzing programs with procedures has been provided [Cousot and Cousot, 1977b].

Various further approaches to interprocedural analysis are reviewed by Sharir and Pnueli [1981]—one based on call-strings while the other, similar to [Cousot and Cousot, 1977b], relies on (partially) computing abstract procedure summaries. Following Sharir and Pnueli, later, restricted frameworks for interprocedural analyses have been provided [Reps et al., 1995] which, however, only work for specific simple domains and therefore may not serve as general program analysis frameworks.

The feature of partial contexts is important for scalability since it makes compositional approaches more practically feasable. It is particularly useful for heap analysis, though the analysis designer must provide a way to isolate the procedure-relevant portion of the heap and retrofit the partial effect of a procedure into the wider context at a call site [Rinetzky et al., 2005; Calcagno et al., 2009]. For object-oriented languages, *object-sensitivity*, which distinguishes call-contexts using only the receiver object at a method invocation site, rather than the entire points-to information, is sufficiently precise [Milanova et al., 2005].

Generic frameworks for program analysis and code optimization [Necula et al., 2002; Lattner and Adve, 2004; Vallée-Rai et al., 1999; Wilson et al., 1994] follow the multi-stage analysis paradigm and do not provide a unified solving algorithm that allows one to combine mutually dependent flow-sensitive and flow-independent analyses. However, specific pointer-analyses have been proposed which flow-sensitively track a subset of relevant pointers. The *client-driven* pointer analysis by Guyer and Lin [2003] monitors the performance of an initial flow-insensitive analysis to decide which pointers to track flow-sensitively. Lhoták and Chung [2011], wishing to perform strong updates, track pointers with singleton points-to sets flow-sensitively, while relying on fast flow-insensitive approaches for points-to sets where precision is already lost.

3.1 Programs

Before we can move on to the specific details of our approach of side-effecting constraint systems, we, first, need to have a better understanding of the structure and meaning of the programs that we are going to analyze. Programs consist of a finite set of procedures Proc, including a procedure main. Each procedure p is given by its *control flow graph* (N_p, E_p, s_p, r_p), where

- N_p is the (finite) set of nodes,
- $E_p \subseteq N_p \times L \times N_p$ is the (finite) set of labeled edges, and where
- $s_p \in N_p$ and $r_p \in N_p$ are the *start* and *return* nodes of p, respectively.

The set of labels L consists of

- statements $s \in \mathsf{Stmt}$, excluding procedure calls,
- procedure calls of the form $p()$ where $p \in \mathsf{Exp}$ is an expression,
- thread spawning of the form $\mathsf{spawn}(p)$ where p is an expression,
- positive guards $\mathsf{Pos}(e) \in \mathsf{Guard}$ and negative guards $\mathsf{Neg}(e) \in \mathsf{Guard}$, given an expression $e \in \mathsf{Exp}$.

For every procedure p, the start node is a source, i.e, it has in-degree of zero, $\deg^-(s_p) = 0$. Dually, the return node r_p is a sink, i.e., it has out-degree of zero, $\deg^+(r_p) = 0$.

We assume, for simplicity, that every node $u \in N_p$ is reachable from the start node of the procedure p, i.e., there exists a path from s_p to u. Similarly, we require that the return node r_p is reachable from all nodes $u \in N_p$.

We require that the out-degree of all nodes must be less than three. Moreover, for nodes with out-degree of exactly two, one outgoing edge must be a positive and the other a negative guard over the same expression. Although the syntax of the call edge does not allow function arguments explicitly, passing of arguments or returning results may be simulated, e.g., by means of global variables. Additionally, using N we denote the disjoint union $\biguplus_{p \in \mathsf{Proc}} N_p$ for all $N_p, p \in \mathsf{Proc}$.

Example 1. Figure 3.1 depicts the functions f and g, first, as written in C, and second, by their control flow graphs. ∎

```
int i, r;
void g(){
  r = r + i;
}
void f(){
  i = 0;
  while (i<100){
    g(); i++;
  }
}
```

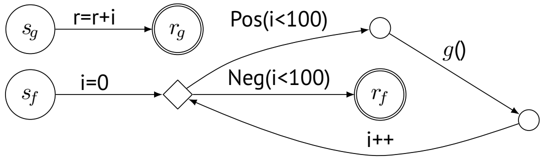


Figure 3.1: Two functions in C and their representations as CFGs.

3.2 Semantics

The execution of a program consists of interleaved executions of *threads* where each thread may advance in its execution on the control flow graph by performing a step. Initially, a single thread will begin to execute the start of the main procedure, however, other threads may be spawned by the program.

Each thread has a *call stack* that contains CFG nodes from possibly different procedures. At the top of the stack lies the current CFG node (of the thread); the following elements are return points—CFG nodes where to jump to after reaching a return node. In the following we are going to use ML-style list syntax for stacks.

The binary relation $(\twoheadrightarrow) \subseteq (\mathsf{Stack} \times S) \times (\mathsf{Stack} \times S \times \mathsf{Proc}^*)$ formalizes the execution step of a thread, where Stack represents the set of call stacks, S the set of program states, and Proc^* the possibly empty sequence of procedures. The thread step relation specifies how the program state is transformed during the execution of the step, and, similarly, which new threads are spawned.

The thread step relation is defined in Figure 3.2 using a denotational style semantics function $[\![s]\!]_{\mathsf{Stmt}} \in S \to S$ for each statement $s \in \mathsf{Stmt}$, and an evaluation function $[\![e]\!]_{\mathsf{Exp}} \in S \to \mathsf{Val}$ for each expression $e \in \mathsf{Exp}$. The evaluation function is used to evaluate the expressions on guard edges

STMT
$$\frac{(u, l, v) \in E_p \qquad [\![l]\!]_{\mathsf{Stmt}}\ s = s'}{(u :: c, s) \twoheadrightarrow (v :: c, s', [])}$$

POS
$$\frac{(u, \mathsf{Pos}(e), v) \in E_p \qquad [\![e]\!]_{\mathsf{Val}}\ s = \ \mathsf{true}}{(u :: c, s) \twoheadrightarrow (v :: c, s', [])}$$

NEG
$$\frac{(u, \mathsf{Neg}(e), v) \in E_p \qquad [\![e]\!]_{\mathsf{Val}}\ s \neq \ \mathsf{true}}{(u :: c, s) \twoheadrightarrow (v :: c, s', [])}$$

CALL
$$\frac{(u, f(), v) \in E_p \qquad [\![f]\!]_{\mathsf{Val}}\ s = p \qquad p \in \mathsf{Proc}}{(u :: c, s) \twoheadrightarrow (s_p :: v :: c, \mathsf{enter}_p\ s, [])}$$

RET
$$\frac{p \in \mathsf{Proc}}{(r_p :: c, s) \twoheadrightarrow (c, \mathsf{return}_p\ s, [])}$$

SPAWN
$$\frac{(u, \mathsf{spawn}(f), v) \in E_p \qquad [\![f]\!]_{\mathsf{Val}}\ s = p \qquad p \in \mathsf{Proc}}{(u :: c, s) \twoheadrightarrow (v :: c, s, [p])}$$

Figure 3.2: The Intra-thread Step Relation "$\twoheadrightarrow$"

$$\frac{0 \leq i \leq n \qquad (t_i, s) \twoheadrightarrow (t'_i, s', p)}{([t_0, \cdots, t_i, \cdots, t_n], s) \Rrightarrow ([t_0, \cdots, t'_i, \cdots, t_n] @ [\, [q] \mid q \in p\,], s')}$$

Figure 3.3: The Inter-thread Step Relation "$\Rrightarrow$"

to decide whether to take the edge with the positive or negative guard. Furthermore, the step relation makes use of state transformers for entering and returning from calls: $\mathsf{enter}, \mathsf{return} \in \mathsf{Proc} \to S \to S$.

As already mentioned, the relation $(c, s) \twoheadrightarrow (c', s', w)$ describes a possible step of a thread, i.e., in case a thread has a call stack c and is in the state s, then, the thread may move into a state s', with the new call stack c', and spawn new threads w. Note that a single step is always performed *atomically*—it is important to only allow statements and expressions where this premise of atomicity is fulfilled.

To figure out the next step of a thread, one must look at the outgoing edges of the current node. In case the outgoing edge is a statement, the step according to rule STMT may be used. Similarly, there are rules for procedure calls (CALL), returns (RET), and thread spawning (SPAWN).

The step related to guards is a bit more complicated, as one has to evaluate the guarding expression. In the case where the expression evaluates to true, one must step through the Pos edge, otherwise the Neg edge.

Notice that the step relation defines a partial function—for each call stack and state, there is at most one result stack, state, and list of new threads. Furthermore, the only case where the transition gets stuck is the case where the call stack is of the form $[r_p]$, for some $p \in \mathsf{Proc}$, i.e., the thread has terminated.

Using the step relation for a thread "$\twoheadrightarrow$", we can now define a similar step relation for the complete program $(\Rrightarrow) \subseteq (\mathsf{Stack}^* \times S) \times (\mathsf{Stack}^* \times S)$. Making a step in the whole program (as defined in Figure 3.3) means to pick a thread that is able to make a step, and then perform that step.

The relation "$\Rrightarrow$" is deterministic only in the case when there is only one thread that can perform steps—the order in which threads are picked is out of our control.

Given a pre-defined starting state $s_0 \in S$, the semantics of the program is the set of tuples of the form $\mathcal{S} \subseteq \mathsf{Stack}^* \times S$ that can be reached using repeated application of the "$\Rrightarrow$" relation, starting from $([[s_{\mathsf{main}}]], s_0) \in \mathsf{Stack}^* \times S$. More formally, using the Kleene star operator, it can be conveniently written as

$$\mathcal{S} = \{\, y \mid ([[s_{\mathsf{main}}]], s_0) \Rrightarrow^* y \,\} \,.$$

In some cases it is interesting to consider the possible states (elements of S), that may occur while visiting a concrete node u of some procedure p. For those cases, we define the sets $\mathcal{S}[u] \in 2^S$ as follows:

$$\mathcal{S}[u] = \{\, s \mid ([t_0, \cdots, u :: t'_i, \cdots, t_n], s) \in \mathcal{S} \,\} \,.$$

3.3 Abstraction

The semantics of a program as defined in the last section describes exactly what may happen at any point during the execution. In most cases, however, the set $\mathcal{S}$ is not reasonably computable. Even if the set of states S is finite and no new threads are created, the set $\mathcal{S}$ may grow too large to be found in tolerable time.

A useful idea that we can apply in this case is *over-approximation*. Over-approximation, or abstraction, however, must be used wisely—not only to ensure the termination of the computation, but also to have a chance of finding reasonable, non-trivial properties.

Definition 3.3.1 (Partial order).
A set D together with an operator $(\sqsubseteq) \subseteq D \times D$ is a partial order if the relation $\sqsubseteq$ is reflexive, transitive, and antisymmetric. ■

Definition 3.3.2 (Least upper bound).
For a partial order $(D, \sqsubseteq)$, d is the least upper bound of $X \subseteq D$ in the case when

a) d is an upper bound of X, i.e., $\forall x \in X,\ x \sqsubseteq d$, and

b) all upper bounds are at least as large as d, i.e.,

$$\forall d',\ (\forall x \in X.\, x \sqsubseteq d') \implies d \sqsubseteq d'\ .$$ ■

Definition 3.3.3 (Complete lattice).
The tuple $(\mathbb{D}, \sqsubseteq)$ is a complete lattice in the case when it is a partial order and each subset $X \subseteq \mathbb{D}$ has a least upper bound $\bigsqcup X \in \mathbb{D}$. Using the least upper bound operator we additionally define a binary least upper bound operator $x \sqcup y = \bigsqcup\{x, y\}$, the least element $\bot = \bigsqcup \emptyset$, and the greatest element $\top = \bigsqcup \mathbb{D}$. ■

Example 2. Four examples of complete lattices are depicted on Figure 3.4: *a*) a boolean lattice, *b*) an integer lattice with natural ordering, *c*) an extended integer lattice with "flat" ordering, and *d*) a subset lattice with subset inclusion ordering. ■

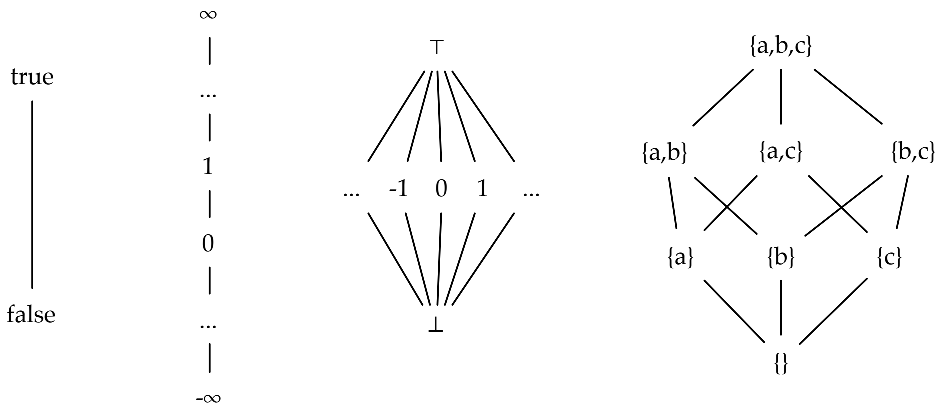


Figure 3.4: Examples of lattices, shown using Hasse diagrams.

Abstraction, in this case, means that we are looking for a mapping X so that for all v, $\mathcal{S}[v] \subseteq X[v]$. This means that we allow to *weaken* the statement about program points. To compute such an over-approximation of (concrete) semantics, we start with picking a *complete lattice* $(\mathbb{D}, \sqsubseteq)$ so that each element $d \in \mathbb{D}$ is a finite representation of a possibly infinite set of states. This step is necessary as we obviously cannot compute using infinite sets directly.

The relation between lattice elements and corresponding sets of states can be formalized using a *description relation* $\Delta \subseteq S \times \mathbb{D}$ as follows. The least element should not describe any state, i.e., there should be no $s \in S$ so that $s \ \Delta \ \bot$, and the greatest element in the lattice must describe the set of all states, i.e., $\forall s \in S.\ s \ \Delta \ \top$. Additionally, the description relation must reflect the ordering of the lattice:

$$s \ \Delta \ d_1 \wedge d_1 \sqsubseteq d_2 \implies s \ \Delta \ d_2 \ .$$

Sometimes, instead of a description relation, a *concretization* function $\gamma \in \mathbb{D} \to 2^S$ is used instead. For each element of the lattice $d \in \mathbb{D}$, the set $\gamma(d)$ is the set of (concrete) states that are described by d, i.e.,

$$\gamma(d) = \{ s \in S \mid s \ \Delta \ d \} \ .$$

Definition 3.3.4 (Galois connection).
Given partially ordered sets $(A, \leq_A)$ and $(B, \leq_B)$, the two functions $F : A \to B$ and $G : B \to A$ form a Galois connection if for all $a \in A$ and $b \in B$

$$F(a) \leq_B b \iff a \leq_A G(b)$$

holds. ■

A useful property to strive for, when designing a lattice, is that for each subset of states $X \subseteq S$, there exists a lattice element $\alpha(X) \in \mathbb{D}$ so that it is the smallest lattice element to describe all states in X. In such a case, the functions γ and α form a *Galois connection*.

To re-iterate and refine the statement stated earlier in the section, we are looking to compute a mapping $\mathcal{S}^\sharp$ so that for all $v \in N$, $s \in \mathcal{S}[v] \implies s \ \Delta \ \mathcal{S}^\sharp[v]$. I.e., the logical statement computed using the lattice must describe the set of states that are defined by the concrete semantics.

In the case where the lattice $\mathbb{D}$ forms a Galois connection with the concrete set of states, we can state the optimistic expectation of computing the lattice value that describes $\mathcal{S}[v]$ best—namely $\mathcal{S}^\sharp[v] = \alpha(\mathcal{S}[v])$. However, the value $\alpha(\mathcal{S}[v])$ is not necessarily computable in finite time.

Our strategy to compute such sets $\mathcal{S}^\sharp[v]$ lies, first, in constructing a *constraint system* so that its solution describes the concrete semantics as requested. To solve the constraint system, we can then use off-the-shelf constraint system solvers. This approach allows us to elegantly separate the concerns of describing over-approximation of the semantics from computing it.

Note that, intuitively, the description relation does not only assign meaning to lattice elements, but also states that the ordering on the lattice is the *implication* order, where larger elements are weaker statements about the program point. So it should not be surprising that we are interested in the smallest solution of the constraint system that we can find with a reasonable effort.

3.4 Constraint Systems

Definition 3.4.1 (Constraint system).
Given a complete lattice $(\mathbb{D}, \sqsubseteq)$, a (pure) constraint system $\mathcal{C}$ is a set of pairs (x, f) where $x \in V$ is a (constraint) variable and $f \in (V \to \mathbb{D}) \to \mathbb{D}$ is a function that computes a contribution to the variable x, given a variable assignment as the first argument. A variable assignment $\sigma \in V \to \mathbb{D}$ is called the (post-)solution of a constraint system $\mathcal{C}$ in the case when it subsumes all contributions, i.e., for all $(x, f) \in \mathcal{C}$ we have $\sigma\ x \sqsupseteq f\ \sigma$. Note that all constraint systems possess a trivial solution—the variable assignment $\sigma = (\lambda\, x.\, \top)$. ■

An element of a constraint system is called a constraint. For a constraint $(x,\ f) \in \mathcal{C}$, the variable x is sometimes called the *left-hand side*, and the function f the *right-hand side*. For better readability, constraints $(x,\ \lambda\,\mathsf{get}.\ e) \in \mathcal{C}$ are sometimes denoted as "$x \sqsupseteq e$" and in those cases get will always be the name of the first parameter of functions representing right-hand sides.

Definition 3.4.2 (Monotonicity).
For partially ordered sets $(X, \leq_X)$ and $(Y, \leq_Y)$, the function $f \in X \to Y$ is called monotonic if for all $x_1, x_2 \in X$ the fact that $x_1 \leq_X x_2$ implies $f\ x_1 \leq_Y f\ x_2$. ■

Definition 3.4.3 (Lattice height).
The height of a lattice $(\mathbb{D}, \sqsubseteq)$ is the maximal length of strictly ascending chains in the lattice, i.e., $\mathbb{D}$ has height h if it does not contain longer strictly ascending chains than the following

$$\bot \sqsubset d_1 \sqsubset d_2 \sqsubset \cdots \sqsubset d_h\ .$$

■

In the case when all the right-hand sides of a constraint system $\mathcal{C}$ are monotonic, then the constraint system has a unique least solution. Moreover, if the set of variables V is finite, and the complete lattice $\mathbb{D}$ has a finite height, then the least solution is computable.

Definition 3.4.4 (Kleene's fixed-point iteration).
For a lattice $(\mathbb{D}, \sqsubseteq)$ and monotonic function $f \in \mathbb{D} \to \mathbb{D}$, the chain

$$\bot \sqsubseteq f \bot \sqsubseteq f(f \bot) \sqsubseteq f^3 \bot \sqsubseteq \ldots \sqsubseteq f^n \bot \sqsubseteq \ldots$$

will, after stabilization, reach the least fixed-point of f. In the special case that the lattice has a finite height, stabilization is guaranteed. ■

Theoretically, a constraint system $\mathcal{C}$ with variables $V = \{x_1, x_2, \ldots, x_n\}$ can also be represented as a single inequation

$$\vec{x} \sqsupseteq' F \vec{x} ,$$

where the lattice relation for vectors of lattice elements $(x_1, x_2, \ldots, x_n) \in \mathbb{D}^n$ is defined point-wise, i.e.,

$$(x_1, x_2, \ldots, x_n) \sqsupseteq' (y_1, y_2, \ldots, y_n) \iff x_1 \sqsupseteq y_1 \wedge x_2 \sqsupseteq y_2 \wedge \ldots \wedge x_n \sqsupseteq y_n ,$$

and the function $F \in \mathbb{D}^n \to \mathbb{D}^n$ is defined as

$$F \vec{x} = (f_1 \vec{x}, f_2 \vec{x}, \ldots, f_n \vec{x})$$

and

$$f_i (x_1, x_2, \ldots, x_n) = \bigsqcup_{(x_i, h) \in \mathcal{C}} h (\lambda j. x_j) .$$

This allows to find the least solution of $\mathcal{C}$ by computing the least fixed-point of F, for example, by using Kleene's fixed-point iteration—at least in the case where the right-hand sides are monotonic and the lattice $\mathbb{D}$ is of a finite height. The use of dedicated constraint systems solvers, however, may allow for more efficient solving in case right-hand sides may only depend on few variables.

One of the simplest solvers is the round-robin solver **RR** (Figure 3.5). At start, **RR** initializes the vector ρ by setting all constraint variables X to $\bot$. The algorithm contains two nested loops. The outer loop iterates until the inner loop does no longer change the vector ρ. The inner loop recomputes each contribution of each constraint $(x, f) \in \mathcal{C}$ by evaluating f with the current ρ. In the case that the new contribution is not subsumed by the current value $\rho[x]$, it is added to the vector by means of the least upper bound operator.

```
foreach $x \in$
$X$ do $\rho[x] \leftarrow \bot$;
do
  dirty $\leftarrow$ false;
  foreach $(x,f) \in \mathcal{C}$ do
    new $\leftarrow \rho[x] \sqcup f\ \rho$;
    if $\rho[x] \neq$ new then
      $\rho[x] \leftarrow$ new;
      dirty $\leftarrow$ true;
  done
while (dirty)
```

Figure 3.5: The solver **RR**.

```
foreach $x \in$
$X$ do $\rho[x] \leftarrow \bot$;
W $\leftarrow \mathcal{C}$;
while W $\neq \emptyset$ do
  $(x,f) \leftarrow$ extract(W);
  new $\leftarrow \rho[x] \sqcup f\ \rho$;
  if $\rho[x] \neq$ new then
    $\rho[x] \leftarrow$ new;
    W $\leftarrow$ W $\cup$ infl$_x$
done
```

Figure 3.6: The solver **W**.

The outer loop of the round-robin solver performs at most $h \cdot n + 1$ rounds, where h denotes the height of the lattice and n the number of (solver) variables. The upper bound comes from the fact that for each round at least one variable has to increase, and moreover, each variable can only increase up to h times. The last round confirms the stabilization of values. Therefore, if each variable has at most one right-hand side, then each inner round performs no more than n evaluations of a right-hand side—totaling at most $h \cdot n^2 + n$ right-hand side evaluations.

Another—much more practical—solver is the worklist solver **W** [Kildall, 1973; Vergauwen et al., 1994; Jørgensen, 1994]. The algorithm **W** (Figure 3.6), however, can only be used given that all dependences are provided beforehand. This means that for each right-hand side f of a variable x, a (super-)set dep_x of variables is given such that for all mappings ρ, ρ', $f\ \rho = f\ \rho'$ whenever ρ and ρ' agree on all variables in dep_x. From these sets, we define the sets infl_y of constraints possibly influenced by (a change of the value of) variable y, i.e.,

$$\mathsf{infl}_y = \{\, (x, f) \in \mathcal{C} \mid y \in \mathsf{dep}_x \,\} .$$

In the case that the value of some variable y changes, all right-hand sides in the set infl_y must be re-computed.

The work set W limits the set of constraints to be recomputed to the constraints that actually may be unstable. Complexity-wise, the worklist algorithm has an advantage over round-robin in the case where the sets dep_x are smaller than the set of all constraints $\mathcal{C}$. The worst case run time is at most $h \cdot N$ where

$$N = \sum_{x \in X} (1 + |\mathsf{dep}_x|) \,.$$

3.5 Intra-Procedural Constraint Systems

Definition 3.5.1 (Flow-sensitive analysis).
If a static analysis technique is designed to take into account static control-flow of the program, then it is considered to be *flow-sensitive*. If, however, an analysis disregards control-flow information, e.g., by viewing a procedure as set of statements instead of as a graph, then it is called *flow-insensitive*. ■

Let us for the moment consider a program with just one procedure main and without procedure calls nor thread creation. Analyzing such programs is referred to as *intra-procedural* analysis. Assume that we are interested in inferring one separate invariant for each program point of the analyzed program, making it a *flow-sensitive* analysis. Flow-sensitive intra-procedural invariants can conveniently be expressed as solutions of a constraint system.

Definition 3.5.2 (Strictness).
A function $f \in \mathbb{D} \to \mathbb{D}$ over a lattice $(\mathbb{D}, \sqsubseteq)$ is called strict in the case that $f\ \bot = \bot$, i.e., if it preserves the value $\bot$. ■

For intra-procedural analysis, the set of variables are simply program points $V = N_{\mathsf{main}}$. Assume that $d_0 \in \mathbb{D}$ describes the program state before starting procedure main, i.e., $s_0\ \Delta\ d_0$. Additionally, for each statement or guard s occurring at an edge, we are given the abstract semantics $[\![s]\!]^\sharp \in \mathbb{D} \to \mathbb{D}$, which describes how the abstract state after the execution of s is obtained from the abstract state before the execution. This means that for all $x_1 \in S$ and $d \in \mathbb{D}$, if $x_1\ \Delta\ d$ then also $[\![s]\!]_{\mathsf{Stmt}}\ x_1\ \Delta\ [\![s]\!]^\sharp\ d$. As usual in program analysis, these functions are assumed to be monotonic.

Monotonicity of a right-hand side is guaranteed, for example, if the lattice forms a Galois connection with the set of concrete states, and the abstract semantics function for statements returns the least lattice element that over-approximates the concrete semantics, i.e.,

$$[\![s]\!]^\sharp\ d = (\alpha \circ [\![s]\!]_{\mathsf{Stmt}} \circ \gamma)\ d = \alpha(\{\ [\![s]\!]_{\mathsf{Stmt}}\ s \mid s \in \gamma(d)\ \})\ .$$

Because $\bot \in \mathbb{D}$ represents the empty set of concrete program states, i.e., can only be assumed at a program point which is *unreachable*, we assume the functions $[\![s]\!]^\sharp$ to be *strict*. We know that s_{main} is reachable and $d_0 \neq \bot$.

An initial abstract state $d_0 \in \mathbb{D}$ for the start point s_{main} together with the edges of the control flow graph give rise to the following system of constraints:

$$\begin{aligned} [s_{\mathsf{main}}] &\sqsupseteq d_0 \\ [v] &\sqsupseteq [\![s]\!]^\sharp\ (\mathsf{get}\ [u]) \qquad \forall (u,s,v) \in E_{\mathsf{main}} \end{aligned} \tag{0}$$

The whole constraint system uses finitely many variables, where the evaluation of the right-hand side of each constraint may also access finitely many variables. In case that the domain $\mathbb{D}$ does not contain infinite strictly ascending chains, a solution of (0) can be computed, e.g., with round-robin iteration or a worklist solver. Moreover, since all abstract functions $[\![s]\!]^\sharp$ are assumed to be monotonic, the computed result will be the least solution—at least when using solvers **RR** or **W**.

As a side-note: in the case where the right-hand sides are not monotonic, the solvers given are still guaranteed to terminate and return a post-solution. This is achieved in the presented solvers by *accumulating* contributions—taking the least upper bound of the old value and the new contribution.

3.6 Analyzing Procedures

In this section we look at one of the classical conceptual bases of analyzing procedures—*the functional approach* by Sharir and Pnueli [1981]. Next, we show that the functional approach can already be represented in the very simple call-stack-0 framework. After that, the more practical tactic, presented by Sharir and Pnueli [1981] is represented as a constraint system, following Cousot and Cousot [1977b]. The constraint system can then be solved by standard means, instead of a dedicated algorithm as in the original version.

3.6.1 The Functional Approach

Let $\mathbb{F}$ be a carrier set for a complete lattice that represents functions $\mathbb{D} \to \mathbb{D}$. Let $\mathrm{id}_{\mathbb{D}} \in \mathbb{F}$ represent the identity function and $\circ \in \mathbb{F} \to \mathbb{F} \to \mathbb{F}$ function composition. In the following, we assume that there are local and global variables, but that procedures have neither parameters nor return values. The latter simplifies the presentation but is no real restriction since passing of value parameters as well as returning of results can be simulated by means of global variables.

In presence of local variables and global variables, we require operators $\mathcal{H}_e \in \mathbb{F} \to \mathbb{F}$ which transforms the effect of a procedure f into the effect of a call to the procedure f at edge e. Then procedure summaries can be characterized as the least solution of the following constraint system over the complete lattice $\mathbb{F}$:

$$\begin{aligned} \{\!\{s_f\}\!\} &\sqsupseteq \mathrm{id}_{\mathbb{D}} && f \in \mathsf{Proc} \\ \{\!\{v\}\!\} &\sqsupseteq (\mathcal{H}_e\,(\mathsf{get}\;\{\!\{r_f\}\!\})) \circ (\mathsf{get}\;\{\!\{u\}\!\}) && e \equiv (u, f(), v) \in E \\ \{\!\{v\}\!\} &\sqsupseteq [\![s]\!]^\sharp \circ (\mathsf{get}\;\{\!\{u\}\!\}) && (u, s, v) \in E \end{aligned} \tag{s}$$

Here, the variable v represents the summary of the effects of program executions which start at the entry point of the current procedure and reach program point v inside the same invocation of the procedure. In particular, the value for $\{\!\{r_f\}\!\}$, r_f being the return point of procedure f, represents the summary for f.

In a second phase, the summaries for procedures can be applied to determine an abstract description of the program states *reaching* a program point v. Let $\mathsf{enter}_e^\sharp \in \mathbb{F}$ denote the abstract description of the transformation of the program state before the call at edge e into the program state at the start point of the called procedure. Additionally, we require an abstract notion of function application of type $\mathbb{F} \to \mathbb{D} \to \mathbb{D}$ which we denote by juxtaposition. Then this second phase can be characterized by the least solution of the following constraint system, now over the complete lattice $\mathbb{D}$:

$$\begin{aligned}
[s_{\text{main}}] &\sqsupseteq d_0 \\
[s_f] &\sqsupseteq \mathsf{enter}^\sharp{}_e\ (\mathsf{get}\ [u]) && e \equiv (u, f(), v) \in E \\
[v] &\sqsupseteq \mathcal{H}_e\ \{\!\!\{r_f\}\!\!\}\ (\mathsf{get}\ [u]) && e \equiv (u, f(), v) \in E \\
[v] &\sqsupseteq [\![s]\!]^\sharp\ (\mathsf{get}\ [u]) && (u, s, v) \in E
\end{aligned} \tag{r}$$

Safety of the constraint system (s) can be established by checking the following property for all procedures f. Namely, for all states $x \in S$ and $d \in \mathbb{D}$, whenever $x \mathrel{\Delta} d$ and $([s_f], x) \twoheadrightarrow^* ([r_f], x', \_)$, then also $x' \mathrel{\Delta} (\{\!\!\{r_f\}\!\!\}\ d)$. Then, using the safety of (s), the safety of (r) can be then established against the concrete semantics, i.e., for all variables $v \in N$, if $x \in \mathcal{S}[v]$ then $x \mathrel{\Delta} (\rho\ [v])$, where ρ is a solution for (r).

Similar constraint systems have been proposed in [Knoop and Steffen, 1992]. Constraint systems like (s) and (r) can be solved, e.g., by the worklist solver—provided that a succinct representation of the elements of the complete lattice $\mathbb{F}$ is given, that the required operations like $\circ$ and $\mathcal{H}_e$ are effectively computable for these representations and that the complete lattices $\mathbb{D}$ as well as $\mathbb{F}$ do not have infinite strictly ascending chains.

3.6.2 The Call-stack-0 Approach

Another classical way to support the analysis of code with procedures is based in recording call-stacks of function calls. However, since the number of potential call-stacks may be infinite, usually just finite abstractions of call-stacks are considered. Such a finite abstraction could consist in identifying all call-stacks with their suffix of bounded length and considering a finite abstraction of each stack frame inside the call-stack.

For the moment, we only consider the extreme case where different calls to the same procedure are *not* distinguished. This is the case of call-stacks taken into account up to depth 0. Here the statements are handled in the same manner as in (0), extended with constraints for procedure call edges.

Let again $\mathsf{enter}^\sharp_e$ denote the abstract description of the transformation of the program state before the call at edge e into the program state at the start point of the called procedure. We use a known technique where the entry state of a procedure is generalized from the states at a call sites. Assume further, that we are given functions $\mathsf{comb}^\sharp_e : \mathbb{D} \to \mathbb{D} \to \mathbb{D}$ which combines the state before the call with the state returned by the procedure called at edge e into the state after the call. Thus in particular, the operator $\mathcal{H}_e$ from subsection 3.6.1 is an abstraction of the function $\lambda\ f\ d.\, \mathsf{comb}^\sharp_e\ d\ (f\ (\mathsf{enter}^\sharp_e\ d))$. Then the constraint system (0) is extended with the following constraints:

$$\begin{aligned}[s_f] &\sqsupseteq \mathsf{enter}^\sharp_e\ (\mathsf{get}\ [u]) \\ [v] &\sqsupseteq \mathsf{comb}^\sharp_e\ (\mathsf{get}\ [u])\ (\mathsf{get}\ [r_f])\end{aligned} \qquad e \equiv (u, f(), v) \in E \qquad \texttt{(cs)}$$

The least solution of this constraint system over the complete lattice $\mathbb{D}$ can again be computed by the worklist fixpoint algorithm, if $\mathbb{D}$ has no infinite strictly ascending chains. The safety of (cs) against the concrete semantics can be shown analogously to (r), i.e., for all variables $v \in N$, if $x \in \mathcal{S}[v]$ then $x\ \Delta\ (\rho\ [v])$, where ρ is a solution for (cs).

We note that the call-stack-0 approach is strong enough to implement the functional approach based on procedure summaries. To see this, we combine the constraint systems (s) and (r) into one constraint system of type (cs). Let $\mathbb{D}_1 = \mathbb{D} \times \mathbb{F}$ where $\mathbb{F}$ again represents the summary functions over the complete lattice $\mathbb{D}$. For this new lattice, we define an initial value $d_{1,0}$, transfer functions $[\![s]\!]^\sharp_1$, as well as functions $\mathsf{enter}^\sharp_{1,e}$ and $\mathsf{comb}^\sharp_{1,e}$ by:

$$\begin{aligned} d_{1,0} &= (d_0, \mathrm{id}_{\mathbb{F}}) \\ [\![s]\!]^\sharp_1\ (d, g) &= ([\![s]\!]^\sharp\ d, [\![s]\!]^\sharp_{\mathbb{F}} \circ g) \\ \mathsf{enter}^\sharp_{1,e}\ (d, \_) &= (\mathsf{enter}^\sharp_e\ d, \mathrm{id}_{\mathbb{F}}) \\ \mathsf{comb}^\sharp_{1,e}\ (d, g_1)\ (\_, g_2) &= (\mathcal{H}_e\ g_2\ d, \mathcal{H}_e\ g_2 \circ g_1) \end{aligned} \qquad \texttt{(rs)}$$

The second components, thus, are computed along the lines of the constraint system (s). Accordingly, the first component behaves as the corresponding functions of the constraint system (r), where at call edges the corresponding summaries are taken from the second component. We have:

Theorem 1. Let σ, ρ denote the least solutions of the constraint systems (s) and (r), respectively. Let ρ_1 denote the least solution of the corresponding constraint system (cs) which uses definitions (rs). Then for every program point v, $\rho_1\, v = (\rho\, v, \sigma\, v)$. ■

Summarizing, interprocedural analysis based on procedure summaries, as well as based on call-stacks of depth 0, can be implemented with ordinary worklist solvers. This is no longer the case for the functional approach based on partial tabulation.

3.6.3 Partial tabulation

Many practical inter-procedural analyses are based on complete lattices $\mathbb{D}$ where no effective representations for procedure summaries are known. This is already the case for full inter-procedural constant propagation. For such cases, Sharir and Pnueli [1981] propose an approach which conceptually represents procedure summaries by their value tables of which only those entries are computed which may affect the analysis result.

Formulated as a constraint system, following Cousot and Cousot [1977b], the constraint variables for the functional approach of are pairs $V = N \times \mathbb{D}$ where the second component records the *calling-context* of the current instance of the procedure. The value for the variable $[v, d]$, where v belongs to a procedure g, thus, represents the abstract value attained at program point v when g is called and the abstract state at start of the procedure is d.

For the moment, we just consider *static* procedure calls, i.e., call edges of the form $(u, g(), v)$ where the parameterless procedure g is called. Later we indicate how the approach can also be extended to procedure calls where the set of possibly called procedures may depend on the local state.

Following the approach from [Cousot and Cousot, 1977b], we obtain the following constraint system for the variables $[v, d]$:

$$\begin{aligned}
&[s_g, d] \sqsupseteq d && \forall g \in \mathsf{Proc} \\
&[v, d] \sqsupseteq [\![s]\!]^\sharp\ (\mathsf{get}\,[u, d]) && \forall (u, s, v) \in E \\
&[v, d] \sqsupseteq \mathsf{comb}_e^\sharp\ (\mathsf{get}\,[u, d])\ (\mathsf{get}\,[r_g, \mathsf{enter}_e^\sharp\ (\mathsf{get}\,[u, d])]) && \forall e = (u, g(), v) \in E
\end{aligned} \qquad (1)$$

Here the functions $\mathsf{enter}_e^\sharp \in \mathbb{D} \to \mathbb{D}$ and $\mathsf{comb}_e^\sharp \in \mathbb{D} \to \mathbb{D} \to \mathbb{D}$ describe the abstract semantics of procedure calls. Just as for the abstract semantics of statements, we demand these functions to be monotonic and strict in each of their arguments.

For an abstract state d, the application $\mathsf{enter}_e^\sharp\ d$ returns the abstract state in which g is called. The function $\mathsf{comb}_e^\sharp$ on the other hand, describes how the abstract value d_1 before the call must be combined with the abstract value d_2 returned by the call to obtain the abstract value after the call. The constraint for calling a procedure g at program point u, where d is the context of the caller, computes $d_1 = \mathsf{get}\,[u, d]$ and the context $d' = \mathsf{enter}_e^\sharp\ d_1$ of the called procedure g, and combines d_1 with the return state of the call $d_2 = \mathsf{get}\,[r_g, d']$.

Safety of (1) against the concrete semantics can be be reasoned as follows. Given $x \in S$ and $d \in \mathbb{D}$ where x is described by d, then for all $f \in \mathsf{Proc}$, $v \in N_f$ in the case that $([s_f], x) \twoheadrightarrow^* ([v], x', \_)$ it can be shown that x' is described by the value of $[v, d]$.

Definition 3.6.1 (Weak monotonicity).
A constraint system $\mathcal{C}$ is called weakly monotonic with respect to a given partial ordering $(\leq) \subseteq V \times V$, if the following two properties hold.

a) For all $(x, f) \in \mathcal{C}$ and $(y, g) \in \mathcal{C}$, where $x \leq y$, then for all monotonic variable assignments σ, $f\ \sigma \sqsubseteq g\ \sigma$.

b) If $\sigma_1 \sqsubseteq \sigma_2$ and at least one of the variable assignments σ_i is monotonic, then for every $(x, f) \in \mathcal{C}$, $f\ \sigma \sqsubseteq f\ \sigma_2$. ■

Even if all abstract functions $[\![s]\!]^\sharp$ and all $\mathsf{enter}_e^\sharp$ and $\mathsf{comb}_e^\sharp$ are monotonic, the right-hand sides of the constraint system (1) are not necessarily monotonic themselves. The second argument to combine is of the form get $[x, \text{get}\,[y, d]]$, and there is no guarantee that $\sigma_1\,[x,\,a_1] \sqsubseteq \sigma_2\,[x,\,a_2]$ just because $a_1 \sqsubseteq a_2$ and $\sigma_1 \sqsubseteq \sigma_2$. The expression is, however, monotonic in the case that $\mathcal{C}$ is *weakly monotonic* with equality as the variable ordering. In our setting, a variable assignment σ is monotonic, if for all program points v, we have $\sigma\,[v, a_1] \sqsubseteq \sigma\,[v, a_2]$ whenever $a_1 \sqsubseteq a_2$. This monotonicity is sufficient to enforce that constraint system (1) has a unique least solution which is monotonic [Fecht and Seidl, 1999].

The least solution, in some sense, describes the procedure summaries, i.e., the abstract effect of every function g for every context $a \in \mathbb{D}$—no matter whether the procedure g is called for a or not. E.g. $[s_g, \top]$ equals $\top$ by the first constraint in (1), regardless if enter$^\sharp$ for an edge calling g will ever return $\top$.

Computing the least solution using an ordinary worklist algorithm, however, is not generally possible. Adding contexts to variables makes the set of variables infinite, given that $\mathbb{D}$ is infinite. And even if $\mathbb{D}$ is finite, the number of variables depends on the number of elements in $\mathbb{D}$—which might be too large.

Often procedures are only called in few distinct abstract calling-contexts. In this case, *local* solving (as shown in Section 3.7) may succeed by starting from a set X of interesting variables, such as $X = \{[r_{\text{main}}, d_0]\}$, and return a *partial* solution which contains the return values of the procedure summary for the required abstract calling-contexts only.

The least (total) solution of constraint system (1) does not contain reachability information, and it is only by local fixpoint iteration that a set of possibly occurring contexts is identified. Instead of referring to the operational behavior of solvers, we can express reachability directly by means of the constraint system. In order to do so, we modify (1) by replacing the constraints $[s_g, a] \sqsupseteq a$ with

$$\begin{aligned} &[s_{\text{main}}, d_0] \sqsupseteq d_0 \\ &\quad [s_f, a] \sqsupseteq \bigsqcup\{\, a \mid \exists a'{\in}\mathbb{D}, a{=}\text{enter}^\sharp_e\,(\text{get}\,[u, a'])\,\} \qquad \forall e{=}(u, f(), v){\in}E \end{aligned} \tag{1'}$$

Note that for each potential entry state $a \in \mathbb{D}$, the second constraint joins over the same value a, so the result is either $\bigsqcup\emptyset{=}\bot$ or $\bigsqcup\{a\} = a$. The entry point of the called procedure f is constrained by the context a if there exists a call to f in some context a' (of the caller) that produces the entry state a (for the callee). This explicitly encodes reachability into the system. Therefore, in contrast to the constraint system (1), we require a dedicated constraint for the initial call of the main procedure. Assuming that the initial local state is d_0, the initial context is also d_0, as encoded by the first constraint.

The constraint system (1'), however, may have minimal solutions which are not monotonic. Assume, e.g., the case where the procedure main consists just of the program point s_{main}. Then the unique least solution is given by $[s_{\mathsf{main}}, b] = \bot$ for $b \neq d_0$ and $[s_{\mathsf{main}}, d_0] = d_0$—which is not a monotonic variable assignment. For non-monotonic variable assignments, however, right-hand sides of constraints need no longer be monotonic. As any constraint system over a complete lattice, the constraint system (1') has solutions, and if the domain is finite, one may compute it by an accumulating fix-point iteration, i.e., by taking the least upper bound of the new value of a variable and the value from the previous iteration.

If the lattice $\mathbb{D}$ is infinite, however, then the constraint system (1') not only contains infinitely many variables, but also has constraints where the evaluation of a single right-hand side may access infinitely many variables. This is the case for the newly introduced constraints for the entry points of procedures. In order to terminate, local solving requires that there be only finitely many right-hand sides for each variable, and that each constraint depend on finitely many variables only. Therefore, it cannot be applied to solve the system (1').

3.7 Local Solving

The central idea of local solving is that even though the constraint system $\mathcal{C}$ is defined in a very general way, there may exist a sub-system $\mathcal{C}' \subseteq \mathcal{C}$ that is smaller, but gives for some subset of variables $V' \subseteq V$ the same result. Intuitively, this allows us to compute the value of some variable x in a constraint system that has infinitely many variables, but only in the case where the solving algorithm is sure that it can isolate such a sub-system.

Use of local solvers can be considered if systems of equations are infeasibly large or even infinite. Such systems are, e.g., encountered for context-sensitive analysis of procedural languages [Cousot and Cousot, 1977b]. Local solvers query the system of equations for the value of a given variable of interest and try to evaluate only the right-hand sides of those variables that are needed for answering the query [Le Charlier and Van Hentenryck, 1992; Vergauwen et al., 1994; Fecht and Seidl, 1999].

Definition 3.7.1 (Partial solution).
A variable assignment $\eta : V' \to \mathbb{D}$, for a subset of variables $V' \subseteq V$, is called a partial solution of the constraint system $\mathcal{C}$, if

a) all constraints for the subset V' are satisfied, i.e., for all constraints (x, f) of the system $\mathcal{C}$, the following implication holds: $x \in V' \implies \eta\, x \sqsupseteq f\, \eta$, and

b) the right-hand sides of variables in V' only access the values of other variables in V', i.e., for all constraints $(x, f) \in \mathcal{C}$, where x is in the set V', the set of variables accessed using `get` during the evaluation of $f\, \eta$ is subsumed by V'. ■

Definition 3.7.2 (Local solver).
An algorithm which, given a constraint system $\mathcal{C}$ and a set of interesting variables X, will, upon termination, return a partial solution $\eta \in V' \to \mathbb{D}$ where $X \subseteq V'$, is called a local solver. ■

Assume that local fixpoint computation terminates with a partial solution $\eta \in X' \to \mathbb{D}$ where $X \subseteq X'$. Then it follows that the entry point s_g of a procedure g can only be reached with abstract values from $a \in \mathbb{D}$ with $[s_g, a] \in X'$. Accordingly, a program point v can only be reached by abstract values bounded by $\bigsqcup\{\, \eta\, [v, a] \mid [v, a] \in X'\}$, as observed in [Fecht, 1997; Fecht and Seidl, 1999].

A simple local solver **RRL** (Figure 3.7) can be constructed from the round-robin solver (Figure 3.5) by extending it with a function eval. The algorithm starts by looking only at the constraints of variables in the set $V' \supseteq X$ instead of the whole set V. During the evaluation of the right-hand side, if it turns out that the value of another variable $y \notin V'$ is requested using eval, only then it is included in the set V' and considered

```
foreach $x \in X$ do $\rho[x] \leftarrow \bot$;
V' $\leftarrow X$;
do
  dirty $\leftarrow$ false;
  foreach $(x,f) \in \mathcal{C} \wedge x \in$ V' do
    let eval $x$ =
        if $x \notin$ V' then
          dirty $\leftarrow$ true;
          $\rho[x] \leftarrow \bot$;
        V' $\leftarrow$ V' $\cup \{x\}$;
        $\rho[x]$
    in
    new $\leftarrow \rho[x] \sqcup f$ eval;
    if $\rho[x] \neq$ new then
      $\rho[x] \leftarrow$ new;
      dirty $\leftarrow$ true
while (dirty)
```

Figure 3.7: The local solver **RRL**.

in the inner loop. Additionally, the values of the new variables are initialized to $\bot$. The helper function eval performs a kind of introspection of the dependencies of the right-hand side.

A worklist solver can also be adapted to perform local solving, as shown in Figure 3.8. More specifically, to get the local solving algorithm **WL** from **W**, the following two changes are needed.

a) At start, the worklist W is initialized with the set of interesting variables X instead of the full set of variables, and

b) variables are initialized with $\bot$ at the time they are first encountered, instead of at the beginning of the algorithm.

3.8 Partial Context-Sensitivity

As we saw previously, local solving has difficulties when reachability is explicitly encoded in the constraint. Now we take this a step further and

```
W ← $X$;
while W = $\emptyset$ do
  ($x$,$f$) ← extract(W);
  if $x \notin$ dom($\rho$) then $\rho[x] \leftarrow \bot$;
  new ← $\rho[x] \sqcup f\ \rho$;
  if $\rho[x] \neq$ new then
    $\rho[x]$ ← new;
    W ← W ∪ infl$_x$
  }
```

Figure 3.8: The local solver **WL**.

consider analyses where the precision of explicitly computed reachability information affects the precision of the analysis. This is the case, e.g., if only *parts* of the abstract state are used to distinguish between different procedure calls.

Consider a complete lattice $\mathbb{D} = \mathbb{D}_1 \times \mathbb{D}_2$ which is the Cartesian product of complete lattices $\mathbb{D}_1, \mathbb{D}_2$, and assume that calls to procedures f are disambiguated by means of the second component b of a reaching abstract state $(a, b) \in \mathbb{D}$, while the first components corresponding to the same b are merged.

Conceptually, the constraints then take the following form:

$$\begin{aligned}
&[s_{\mathsf{main}}, \langle d_0\rangle_2] \sqsupseteq d_0 \\
&[v, b] \sqsupseteq [\![s]\!]^\sharp\ (\mathsf{get}\,[u, b]) && \forall(u, s, v)\in E \\
&[s_g, b] \sqsupseteq \bigsqcup\{d \mid \exists b'\in\mathbb{D}_2, d{=}\mathsf{enter}_e^\sharp\ (\mathsf{get}\,[u, b']), \langle d\rangle_2{=}b\} \\
& && \forall e{=}(u, g(), v)\in E \qquad (2) \\
&[v, b] \sqsupseteq \mathbf{let}\ d = \mathsf{enter}_e^\sharp\ (\mathsf{get}\,[u, b]) \\
&\qquad \mathbf{in}\ \mathsf{comb}_e^\sharp\ (\mathsf{get}\,[u, b])\ (\mathsf{get}\,[r_g, \langle d\rangle_2]) && \forall e{=}(u, g(), v)\in E
\end{aligned}$$

Here, the operator $\langle\cdot\rangle_i$ extracts the i-th component of a tuple. Technically, this constraint system is a smooth generalization of constraint system (1')—only that now program points v are not distinguished by the full context d in which the procedure of v has been called, but only the second component of d. Similarly to constraint system (1'), the constraint system (2) explicitly keeps track of reachability.

In the particular case where $\mathbb{D}_2$ is the unit domain $\mathbb{1} = \{\bullet\}$, constraint system (2) generalizes a constraint system for call-stack-0 (cs). In this case no contexts are distinguished, and all right-hand sides of the constraint system are monotonic.

For nontrivial contexts, though, constraint system (2), just as constraint system (1'), may have minimal solutions which are not monotonic. Still, in the next section we prove that every solution of (2) provides *sound* analysis information.

Assume for a moment that the complete lattice $\mathbb{D}_2$ of partial contexts is infinite. Then the same argument as for constraint system (1') can be applied to rule out local fixpoint iteration for solving the constraint system (2). But even if the number of partial contexts is finite, use of general fixpoint engines may be infeasible.

According to the constraint of (2) for the starting states of procedure g with the context b, the solver has to track contributions from *all* call sites that *may* call g in context b. Also in complete absence of context-sensitivity (i.e., where $\mathbb{D}_2 = \mathbb{1}$) but in presence of dynamic procedure calls, a local solver, for instance, will explore *all* possible call sites in order to determine the abstract value for the start node of g. The same already holds true in presence of partial context-sensitivity (i.e., both $\mathbb{D}_1$ and $\mathbb{D}_2$ are different from $\mathbb{1}$). We conclude that even in these simple cases, the number of variables considered by the local solver on constraint system (2) might be excessively large.

3.9 Proof of soundness

Before proceeding, we prove that *every* solution of constraint system (2) is a sound abstraction of the collecting semantics. For that proof, we assume, again, that program execution operates on a set S of *concrete* states whereas the analyzer operates with abstract states from a complete lattice $\mathbb{D} = \mathbb{D}_1 \times \mathbb{D}_2$. As before, we assume for the concretization that $\gamma(\bot) = \emptyset$, i.e., that no concrete state is described by $\bot$.

The collecting semantics of the program then can be restated as the least solution of the following constraint system over sets of concrete states:

$$\begin{array}{rll}
[s_{\mathsf{main}}, a_0] \supseteq \{a_0\} & & \\
[s_g, a] \supseteq \{\, a \mid a' \in S, a_1 \in \mathsf{get}\,[u, a'], a \in \mathsf{enter}_e\; a_1 \} & \forall e = (u, g(), v) \in E & \\
[v, a] \supseteq \bigcup \{\, [\![s]\!]\; a' \mid a' \in \mathsf{get}\,[u, a] \} & \forall (u, s, v) \in E & \text{(c)} \\
[v, a] \supseteq \bigcup \{\, \mathsf{comb}_e\; a_1\; a_2 \mid a_1 \in \mathsf{get}\,[u, a], & & \\
a' \in \mathsf{enter}_e\; a_1, a_2 \in \mathsf{get}\,[r_g, a'] \} & \forall e = (u, g(), v) \in E &
\end{array}$$

Each variable $[u, a]$ represents the set of states possibly reaching program point u of some procedure g when executing g in entering context a, i.e.,

$$\bigcup \{\, \sigma[v, a] \mid a \in S \,\} = \mathcal{S}[v]$$

where σ is the solution of the constraint system (c). The functions:

$$\begin{aligned}
[\![s]\!] &\in S \to 2^S \\
\mathsf{enter}_e &\in S \to 2^S \\
\mathsf{comb}_e &\in S \to S \to 2^S
\end{aligned}$$

formalize evaluation of basic statements or guards, concrete passing of parameters into calls and combining states before calls with returned states to the state after the calls, respectively. Their effects are assumed to be described by the corresponding abstract functions. This means that for all $a \,\Delta\, d$, $a' \,\Delta\, d'$ the following holds:

$$\begin{aligned}
[\![s]\!]\; a &\subseteq \gamma([\![s]\!]^\sharp\; d) \\
\mathsf{enter}_e\; a &\subseteq \gamma(\mathsf{enter}_e^\sharp\; d) \\
\mathsf{comb}_e\; a\; a' &\subseteq \gamma(\mathsf{comb}_e^\sharp\; d\; d')
\end{aligned}$$

Note that all right-hand sides of the concrete constraint system (c) are monotonic. Therefore, constraint system (c) has a unique least solution. We have:

Theorem 2. Let σ denote the least solution of the concrete constraint system (c), and $\sigma^\sharp$ any solution of the abstract constraint system (2) over the domain $\mathbb{D} = \mathbb{D}_1 \times \mathbb{D}_2$. Then the following holds for every program point v.

a) Assume that $d \,\Delta\, (a,b)$ and $(a,b) \sqsubseteq \sigma^\sharp\,[s_g, b]$. Then for every program point v of procedure g, $\sigma\,[v,d] \subseteq \gamma(\sigma^\sharp\,[v,b])$.

b) If $\sigma\,[s_g, d] \neq \emptyset$ and v is a program point of g, then there exists some $(a,b) \in \mathbb{D}$ such that $d \,\Delta\, (a,b)$ and $\sigma\,[v,d] \subseteq \gamma(\sigma^\sharp\,[v,b])$.

c) For all program points v,

$$\sigma\,[v,d] \subseteq \bigcup\{\, \gamma(\sigma^\sharp\,[v,b]) \mid \exists\, a \in \mathbb{D}_1.\, d \,\Delta\, (a,b) \,\} \qquad \blacksquare$$

Proof. The joint proof of statements *a*) and *b*) is by fixpoint induction of the concrete semantics. The base of the induction is trivial. For the inductive step, we only consider values added along a call edge $e = (u, h(), v)$ where u is from procedure g. Assume that $d \in \sigma\,[s_g, d]$, $d \,\Delta\, (a,b)$ and $d \,\Delta\, (\sigma\,[s_g, b])$. Assume that d'' has been added to $\sigma\,[v,d]$ along the edge e, i.e., $d'' \in \mathsf{comb}_e\, d_1\, d_2$ where $d_1 \in \sigma\,[u,d]$ and $d_2 \in \sigma\,[r_h, d']$ for some $d' \in \mathsf{enter}_e\, d_1$. By induction hypothesis, $d_1 \,\Delta\, (\sigma^\sharp\,[u,b])$. Moreover, $d' \,\Delta\, (a', b')$ for $(a',b') = \mathsf{enter}^\sharp_e\, (\sigma^\sharp\,[u,b])$. By construction of the constraint system, then also $(a',b') \sqsubseteq \sigma^\sharp\,[s_h, b']$ and therefore, $d' \,\Delta\, (\sigma^\sharp\,[s_h, b'])$. Consequently again by induction hypothesis, $d_2 \,\Delta\, (\sigma^\sharp\,[r_h, b'])$. Since $\mathsf{comb}^\sharp_e$ is a safe approximation of comb_e, we conclude that

$$d'' \;\Delta\; \mathsf{comb}^\sharp_e\, (\sigma^\sharp\,[u,b])\, (\sigma^\sharp\,[r_h, \langle \mathsf{enter}^\sharp_e\, (\sigma^\sharp\,[u,b])\rangle_2])$$

Statement *c*) directly follows from statements *a*) and *b*). $\blacksquare$

3.10 Flow-insensitive Analyses

One further challenge for general frameworks for automatic program analysis is added when certain pieces of information are meant to be accumulated *flow-insensitively*. Flow-insensitive analyses try to infer invariants which hold throughout the program execution. Such invariants are used, e.g., to reason about dynamic data-structures [Andersen, 1994; Steensgaard, 1996; Shapiro and Horwitz, 1997] or concurrently running threads [Vojdani, 2010].

Technically, flow-insensitive analyses can be constructed by introducing an extra finite set G of entities for which values are accumulated. Depending on the application, the elements of G can, e.g., be viewed as *global* variables, abstract locations of heap objects or the components of the interface through which concurrently running threads communicate. Thus, the effect of the statement s at an edge in the control flow graph may now additionally depend on the values of the globals in G as well as the predecessor state and may also return contributions to the values of some of the globals.

In the following we assume that global and local information are represented by the same lattice $\mathbb{D}$—if this is not the case, one can, for example, use the Cartesian product (with product ordering) of the domains and set the unused pair entry to $\bot$.

One way to describe the effects of a statement s is by modifying the abstract semantics function $[\![s]\!]^\sharp$ to:

$$[\![s]\!]^\sharp \in \mathbb{D} \to (G \to \mathbb{D}) \to \mathbb{D} \times (G \to \mathbb{D})$$

which jointly specifies the contribution to the next program point as well as to certain elements of G. Again, we assume this function to be monotonic in its arguments and strict, at least in its first argument. This means that any call $[\![s]\!]^\sharp\ \bot\ \tau$ should return a pair $(\bot, \underline{\bot})$ where $\underline{\bot}$ maps every global to $\bot$.

In absence of procedures, we thus may put up the following constraint system for approximating invariants for the globals in G:

$$\begin{aligned} [v] &\sqsupseteq \langle [\![s]\!]^\sharp\ (\mathsf{get}\,[u])\ \mathsf{get}\rangle_1 & \forall (u,s,v) \in E \\ [y] &\sqsupseteq \langle [\![s]\!]^\sharp\ (\mathsf{get}\,[u])\ \mathsf{get}\rangle_2\ y & \forall y \in G, (u,s,v) \in E \end{aligned} \tag{3}$$

In absence of procedures, this constraint system can be solved, for example, with solvers **RR** or **W**. While it cannot easily be combined with constraint system (1), it can be combined with the constraint system (2). Assume the complete lattice $\mathbb{D}$ is of the form $\mathbb{D} = \mathbb{D}_1 \times \mathbb{D}_2$ where elements $b \in \mathbb{D}_2$ may serve as contexts. Then we modify the constraint system (2) by replacing the constraints for statements s with:

$$\begin{aligned} [v,b] &\sqsupseteq \langle [\![s]\!]^\sharp\,(\mathsf{get}\,[u,b])\ \mathsf{get}\rangle_1 & \forall (u,s,v) \in E \\ [y] &\sqsupseteq \bigsqcup \{\, \langle [\![s]\!]^\sharp\,(\mathsf{get}\,[u,b])\ \mathsf{get}\rangle_2\ y \mid b \in \mathbb{D}_2 \,\} & \forall y \in G, (u,s,v) \in E \end{aligned} \tag{4}$$

Example 3. As an example, we generate constraints for inter-procedural constant propagation analysis on the following small C program in Figure 3.9. The example program uses a global variable y, one helper procedure up, and a main procedure. We examine the case where globals are handled flow-insensitively; variables and formal parameters flow-sensitively. As parameter passing is not natively supported, we simulate it with a flow-sensitive global variable v. For this program, we would obtain the following system of constraints:

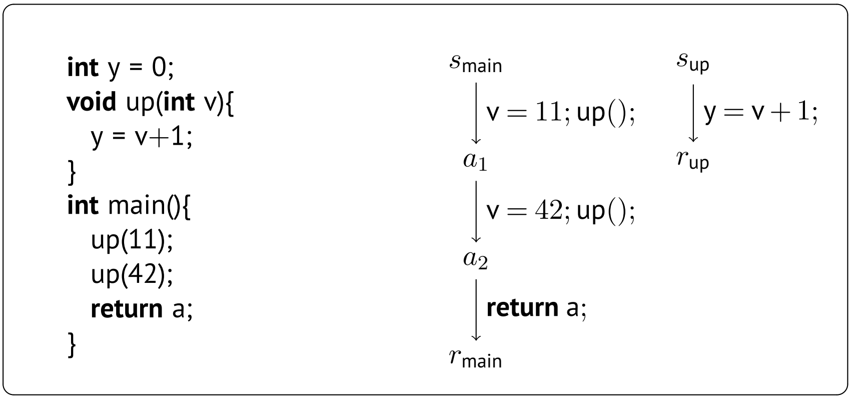
$$\begin{array}{rll}
[s_{\text{main}}, \langle d_0 \rangle_2] \sqsupseteq d_0 & & \\
[a_1, b] \sqsupseteq & \mathsf{get}\ [r_{\text{up}}, [\![v = 11;]\!]^\sharp\ (\mathsf{get}\ [s_{\text{main}}, b])] & \forall b {\in} \mathbb{D}_2 \\
[a_2, b] \sqsupseteq & \mathsf{get}\ [r_{\text{up}}, [\![v = 42;]\!]^\sharp\ (\mathsf{get}\ [a_1, b])] & \forall b {\in} \mathbb{D}_2 \\
[r_{\text{main}}, b] \sqsupseteq & [\![\textbf{return}\ a;]\!]^\sharp\ (\mathsf{get}\ [a_2, b]) & \forall b {\in} \mathbb{D}_2 \\
[s_{\text{up}}, b] \sqsupseteq & \bigsqcup\{d \mid \exists b' {\in} \mathbb{D}_2, d{=}[\![v{=}11;]\!]^\sharp\ (\mathsf{get}\ [s_{\text{main}}, b']), \langle d \rangle_2{=}b\} & \forall b {\in} \mathbb{D}_2 \\
[s_{\text{up}}, b] \sqsupseteq & \bigsqcup\{d \mid \exists b' {\in} \mathbb{D}_2, d{=}[\![v{=}42;]\!]^\sharp\ (\mathsf{get}\ [a_1, b']), \langle d \rangle_2{=}b\} & \forall b {\in} \mathbb{D}_2 \\
[r_{\text{up}}, b] \sqsupseteq & \mathsf{get}\ [s_{\text{up}}, b] & \forall b {\in} \mathbb{D}_2 \\
[y] \sqsupseteq & [\![v + 1]\!]^\sharp\ (\mathsf{get}\ [s_{\text{up}}, b]) & \forall b {\in} \mathbb{D}_2
\end{array}$$

■

Just as for constraint system (2), a local fixpoint algorithm for the enhanced constraint system will behave badly—in order to determine the value for some global y, the algorithm would explore *all* variables $[u, b]$ for which there is a control-flow edge (u, s, v) which *may* contribute to the value of y. If the number of potential contexts is infinite, we again obtain constraints where right-hand sides access infinitely many constraint variables. In the next section, though, we provide alternative formulations of constraint systems (2) and (4) which can be solved by means of partial tabulation.

3.11 Constraint Systems with Side-Effects

Ordinary constraints allow to specify precisely in which order variables are read while the single writing occurs at the very end, namely, to the variable at the left-hand side. Side-effecting constraints generalize this



```
int y = 0;
void up(int v){
  y = v+1;
}
int main(){
  up(11);
  up(42);
  return a;
}
```

Figure 3.9: An Example Program

by allowing multiple writes and also specify precisely in which order constraint variables are not only read, but also written to. In particular, which variable to read or write to next can depend on the values previously read. Even the sets of accessed variables may change. Each execution of a constraint thus yields a sequence of reads and writes which is terminated with an assignment to a left-hand side.

Definition 3.11.1 (Side-effecting constraint system).
A side-effecting constraint system over the complete lattice $(\mathbb{D}, \sqsubseteq)$ is a set of pairs (x, f), where $x \in V$ is a variable and $f \in (V \to \mathbb{D}) \to (V \to \mathbb{D} \to \mathsf{unit}) \to \mathbb{D}$ the function that computes a contribution to x, and, additionally, sends contributions to other variables using the second argument of f. ■

Definition 3.11.2 (Solution of a side-effecting constraint system).
A variable assignment $\sigma \in V \to \mathbb{D}$ is a solution to the side-effecting constraint system $\mathcal{C}$ if for all constraints $(x, f) \in \mathcal{C}$ we have that $\sigma\, x \sqsupseteq f\, \sigma\, \mathsf{set}$ where for every call $\mathsf{set}\; y\; d$ arising from the evaluation of f we have $\sigma\, y \sqsupseteq d$. ■

For a *side-effecting* constraint (x, f), a call $\mathsf{get}\, y$ of the first argument function of f to some variable $y \in V$ is meant to return the value of y in the current variable assignment. A call $\mathsf{set}\, y\, d$ of the second argument function during the evaluation of f for $y \in V$, $d \in \mathbb{D}$ is meant to provide the contribution d to the value of y in the current variable assignment. If f is defined as λ get set . e for an expression e, we again represent the constraint (x, f) by "$x \sqsupseteq e$".

Note that the second argument of a right-hand side—the function set, provided by the solver, should perform a weak update, i.e., overwrite the current value with the least upper bound of the current and the new value.

Definition 3.11.3 (Partial solution of a side-effecting constraint system). A variable assignment $\eta \in X' \to \mathbb{D}$ for a subset $X' \subseteq V$ is a partial solution of a side-effecting constraint system $\mathcal{C}$, if for every constraint $(x, f) \in \mathcal{C}$ where $x \in X'$, f get set is defined if get $= \eta$, and for any call set $y\ d$ with $d \neq \bot$ which arises from f we have $y \in X'$ and $\eta\, y \sqsupseteq d$.

A local solver for side-effecting constraint systems is called with a (small) subset X of interesting variables and tries to determine a partial solution which is defined for as few variables as possible. ∎

A generic *local* solver for side-effecting constraint systems lies at the heart of the analyzer generator Goblint. A slightly simplified version is provided in Section 3.12. It uses *self-observation* for dynamically keeping track of variable dependences.

Side-effecting constraint systems allow us to conveniently specify partially context-sensitive interprocedural analyses. Instead of defining the constraints for the starting point of some function g by means of the *inverse* of the $\mathsf{enter}_e^\sharp$ function as in constraint system (2), we attribute the contributions to the respective call sites as side-effects.

As in Section 3.8, consider an analysis where the domain is the Cartesian product $\mathbb{D}_1 \times \mathbb{D}_2$ of two complete lattices $\mathbb{D}_1$ and $\mathbb{D}_2$. Assume again that calls to procedures should only be distinguished w.r.t. the second component $b \in \mathbb{D}_2$ of reaching states. The constraints for statements or guards are identical to the constraint system (2). We modify the constraints generated for every procedure call edge $(u, g(), v) \in E$ in the following way:

$$\begin{aligned}
[v, b] \sqsupseteq \textbf{let}\ d &= \mathsf{enter}_e^\sharp\,(\mathsf{get}\ [u, b]) \\
() &= \mathsf{set}\ [s_g, \langle d\rangle_2]\ d \\
\textbf{in}\ &\mathsf{comb}_e^\sharp\,(\mathsf{get}\ [u, b])\,(\mathsf{get}\ [r_g, \langle d\rangle_2])
\end{aligned} \tag{5}$$

For an infinite complete lattice $\mathbb{D}_2$, the constraint system (5) requires infinitely many constraint variables. This is identical to constraint system (2). In contrast, however, to system (2), the number of constraint variables accessed in the right-hand side of every single constraint is finite, while there are still only finitely many right-hand sides for each variable. Moreover, we have:

Theorem 3. The constraint systems (5) and (2) are equivalent. This means that every solution of the constraint system (2) is a solution of the constraint system (5) and vice versa, every solution of the constraint system (5) is a solution of the constraint system (2). ■

Proof. We prove equivalence by proving implication in both directions.

a) Given a solution σ for (5) we have to show that σ is a solution to (2). We are going to look at the two constraints explicitly mentioned in (2).

1. We have to show that

$$\sigma\,[s_g, b] \sqsupseteq \bigsqcup\{d \mid \exists b'{\in}\mathbb{D}_2, d = \mathsf{enter}^\sharp\,(\sigma\,[u, b']), \langle d\rangle_2 = b\}\,.$$

It is sufficient to show that $[s_g, b]$ is an upper bound of $d =$ $\mathsf{enter}^\sharp\,(\sigma\,[u, b'])$ for all b' where $\langle d\rangle_2 = b$. This we get from a side-effect of the following constraint in (5):

$$\begin{aligned}\sigma\,[v, b'] \sqsupseteq\ &\mathbf{let}\ d = \mathsf{enter}^\sharp_e\,(\sigma\,[u, b'])\\ &\qquad () = \mathsf{set}\,[s_g, \langle d\rangle_2]\ d\\ &\mathbf{in} \ldots\end{aligned}$$

2. We have to show that

$$\begin{aligned}\sigma\,[v, b] \sqsupseteq\ &\mathbf{let}\ d = \mathsf{enter}^\sharp_e\,(\sigma\,[u, b])\\ &\mathbf{in}\ \mathsf{comb}^\sharp_e\,(\sigma\,[u, b])\,(\mathsf{get}\,[r_g, \langle d\rangle_2])\end{aligned}$$

Ignoring the side-effect, this is exactly the effect to $[v, b]$ in (5).

b) Given a solution σ for (2) we have to show that σ is a solution to (5). We only consider the constraint explicitly mentioned in (5). The crucial part is to verify that all eventual side-effects are covered.

For that, we have to show that for $d = \mathsf{enter}^\sharp\ (\sigma\ [u,b])$ it holds that $\sigma\ [s_g, \langle d\rangle_2] \sqsupseteq d$. This is given by the following constraint in (2):

$$[s_g, b''] \sqsupseteq \bigsqcup\{d \mid d = \mathsf{enter}^\sharp\ (\mathsf{get}\ [u,b']), \langle d\rangle_2 = b''\}$$

if we take $b' = b$. ■

Since every solution of constraint system (2) is a sound abstraction of the concrete semantics, Theorem 3 implies that every solution of constraint system (5) is a sound abstraction of the concrete semantics. In contrast to constraint system (2), constraint system (5) now can be solved by means of local fixpoint iteration.

Side-effecting constraint systems also provide a way to realize flow-insensitive invariants as considered in Section 3.10—even in presence of procedure calls which are analyzed by means of partial tabulation of summaries. The corresponding constraint system is obtained from the constraint system (5) by modifying the constraints for statement or guard edges $(u, s, v) \in E$ by taking the modified abstract semantics $[\![s]\!]^\sharp$ into account:

$$\begin{array}{l} [v,b] \sqsupseteq \textbf{let}\ (d,\tau) = [\![s]\!]^\sharp\ (\mathsf{get}\ [u,b])\ \mathsf{get} \\ \qquad\qquad\qquad () = \textbf{forall}\ (y \in G\ \textbf{with}\ \tau\ y \neq \bot) \\ \qquad\qquad\qquad\qquad\qquad \mathsf{set}\ y\ (\tau\ y) \\ \qquad\quad \textbf{in}\ \ d \end{array} \tag{6}$$

The remaining constraints are as in the constraint system (5). Due to this formulation, contributions to globals y are only collected for contexts b which occur during fixpoint iteration.

Example 4. Using side-effecting constraints, we may now reformulate the constraints from Example 3 as follows:

$$[s_{\text{main}}, \langle d_0 \rangle_2] \sqsupseteq d_0$$

$$[r_{\text{main}}, b] \sqsupseteq [\![\textbf{return } a;]\!]^\sharp \; (\mathsf{get}\; [a_2, b]) \qquad \forall b \in \mathbb{D}_2$$

$$\begin{aligned}[][a_1, b] \sqsupseteq\ & \textbf{let}\ d = [\![v = 11;]\!]^\sharp \; (\mathsf{get}\; [s_{\text{main}}, b]) \\ & \quad () = \mathsf{set}\; [s_{\text{up}}, \langle d \rangle_2]\; d \\ & \textbf{in}\ \mathsf{comb}_e^\sharp\; d\; (\mathsf{get}\; [r_{\text{up}}, \langle d \rangle_2]) \end{aligned} \qquad \forall b \in \mathbb{D}_2$$

$$\begin{aligned}[][a_2, b] \sqsupseteq\ & \textbf{let}\ d = [\![v = 42;]\!]^\sharp \; (\mathsf{get}\; [a_1, b]) \\ & \quad () = \mathsf{set}\; [s_{\text{up}}, \langle d \rangle_2]\; d \\ & \textbf{in}\ \mathsf{comb}_e^\sharp\; d\; (\mathsf{get}\; [r_{\text{up}}, \langle d \rangle_2]) \end{aligned} \qquad \forall b \in \mathbb{D}_2$$

$$\begin{aligned}[][r_{\text{up}}, b] \sqsupseteq\ & \textbf{let}\ () = \mathsf{set}\; [y]\; ([\![v + 1]\!]^\sharp \; (\mathsf{get}\; [s_{\text{up}}, b])) \\ & \textbf{in}\ \mathsf{get}\; [s_{\text{up}}, b] \end{aligned} \qquad \forall b \in \mathbb{D}_2$$

■

3.12 A Generic local solver

A simplified generic local solver for side-effecting constraint systems is given in Figure 3.10. The algorithm maintains a partial assignment σ of variables to values in the complete lattice $\mathbb{D}$, together with a partial assignment I which maintains for every encountered variable x the set of variables which have been accessed during the last evaluation of right-hand sides of x. Since both assignments are changes during a execution of the algorithm, we prefer to denote accesses to σ and I in array notation. Initially, both partial assignments are empty. Additionally, the algorithm maintains a set S of *stable* variables. Stable variables either satisfy their constraints or have been started to be processed and are currently under evaluation. Initially, the set S is also empty.

```
let rec solve $x$ =
  if $x \notin$ dom($\sigma$) then
    $\sigma[x] \leftarrow \bot$;
    I$[x] \leftarrow \emptyset$;
  S $\leftarrow$ S $\cup \{x\}$;
  set $x$ ($\bigsqcup\{f$ (eval $x$) set | $(x,f) \in \mathcal{C}\}$)

and eval $x$ $y$ =
  if $y \notin$ S then solve $y$;
  I$[y] \leftarrow$ I$[y] \cup \{x\}$;
  $\sigma[y]$

and set $x$ $d$ =
  if $x \notin$ dom($\sigma$) then solve $x$;
  if $d \not\sqsubseteq \sigma[x]$ then
    $\sigma[x] \leftarrow \sigma[x] \sqcup d$;
    let $X$ = I$[x]$ in
      S $\leftarrow$ S $\setminus X$;
      I$[x] \leftarrow \emptyset$;
      solve_all $X$

and solve_all $X$ =
  foreach $x \in X$ do solve $x$
```

Figure 3.10: The local solving algorithm.

When calling the main function solve_all with a set X of interesting variables, the recursive worker function solve is called for each variable $x \in X$.

For a variable x, the worker function solve first checks whether σ is already defined for x. If this is not the case, $\sigma[x]$ and $I[x]$ are initialized with $\bot$ and $\emptyset$, respectively. Then x is added to the set S of stable variables. In the next step, the contribution to the new value for x is determined. This new value, the least upper bound of all contributions from the right-hand sides of x, is combined with the old value of x by means of the auxiliary function set.

The evaluation of a right-hand side f, however, is executed not for the assignment σ directly but for the partially applied function call eval x. In the end, when called for another variable y, the value $\sigma[y]$ will be returned. Before that, however, first solve is called for y in order to determine the best possible value for y, and then the variable dependencies of x on y is recorded in $I[x]$.

The second argument passed to f, set is responsible for creating the side effects. The function set, when called for a variable y and a value d, first checks whether y is already in the domain of σ. If this is not the case, y is solved first. Then the value d is compared with the value $\sigma[y]$. If d is not subsumed by $\sigma[y]$, then $\sigma[y]$ is updated to $d \sqcup \sigma[y]$. Since then the current value of y has changed, the variables influenced by y must be reevaluated. This means that they must be removed from the set S of stable variables and scheduled for reevaluation by means of solve. As soon as their reevaluation has been scheduled, they can be removed from the set $I[y]$.

3.13 Dynamic Procedure Calls

Constraints for procedure calls can be extended to deal with *dynamic* calls, i.e., the procedure, to be called, may depend on the current program state. Let this dependence be formalized by means of a modified functionality of the abstract functions $\mathsf{enter}_e^\sharp$

$$\mathsf{enter}_e^\sharp \in \mathbb{D} \to \mathsf{Proc} \to \mathbb{D}$$

where $\mathsf{enter}_e^\sharp\ d\ g = \bot$ indicates that procedure g is definitely not called at the edge e when the concrete state is described with d. Here we only consider the extension of the side-effecting constraint system for partial contexts with dynamic calls. Therefore, assume again that the complete lattice $\mathbb{D}$ of abstract states is of the form $\mathbb{D} = \mathbb{D}_1 \times \mathbb{D}_2$ where the elements in $\mathbb{D}_2$ are used to distinguish between different calls. We get the constraint system by replacing the procedure call constrains in (5) with constraints for every procedure $g \in \mathsf{Proc}$:

$$\begin{aligned}[v, b] \sqsupseteq \textbf{let}\ d &= \mathsf{enter}_e^\sharp\ (\mathsf{get}\ [u, b])\ g \\ () &= \mathsf{set}\ [s_g, \langle d\rangle_2]\ d \\ \textbf{in}\ &\mathsf{comb}_e^\sharp\ (\mathsf{get}\ [u, b])\ (\mathsf{get}\ [r_g, \langle d\rangle_2])\end{aligned} \tag{7'}$$

For efficiency reasons, we do not want to analyze procedures which are not called, i.e., for which $\mathsf{enter}_e^\sharp$ returns $\bot$. In order to avoid that, an extra test first checks whether $\mathsf{enter}_e^\sharp\ d\ g$ has returned $\bot$ or not. Only if that value is different from $\bot$, a side-effect to the start point of g is triggered and the return value of g is combined with the state before the call. This optimization results in:

$$
\begin{aligned}
[v,b] \sqsupseteq \textbf{match}\ & \mathsf{enter}^\sharp_e\ (\mathsf{get}\ [u,b])\ g\ \textbf{with} \\
& \mid \bot \to \bot \\
& \mid d\ \to \textbf{let}\ () = \mathsf{set}\ [s_g, \langle d\rangle_2]\ d \\
& \qquad\quad \textbf{in}\ \mathsf{comb}^\sharp_e\ (\mathsf{get}\ [u,b])\ (\mathsf{get}\ [r_g, \langle d\rangle_2])
\end{aligned}
\tag{7}
$$

3.14 Forward Propagation

The algorithm of Sharir and Pnueli for partially tabulating procedure summaries has proven to be surprisingly efficient in practice. It can also be applied to partially tabulating partial contexts. Interestingly, its algorithmic characteristics are quite different from locally solving ordinary constraint systems. Instead of recursively descending into variable dependences starting from the return point of the initial call to main, i.e. $[r_{\mathsf{main}}, d_0]$, it is based on *forward propagation*. Whenever the abstract state at an variable $[u,b]$ changes, the abstract effects corresponding to all outgoing edges (u,s,v) are executed to trigger the necessary updates for the end points v. This behavior is mandatory for the analysis of *binary* code where the control-flow graphs are not given before-hand but are successively built up while the program is decoded [Flexeder et al., 2010].

We show that this forward propagation can be achieved if the following variant of the side-effecting constraint system (5) is used:

$$
\begin{aligned}
& [s_{\mathsf{main}}, \langle d_0\rangle_2] \sqsupseteq d_0 \\
& \qquad [u,b] \sqsupseteq \textbf{let}\ () = \mathsf{set}\ [v,b]\ ([\![s]\!]^\sharp\ (\mathsf{get}\ [u,b]))\ \textbf{in}\ \bot \qquad \forall (u,s,v) \in E \\
& \qquad [u,b] \sqsupseteq \textbf{let}\ d\ = \mathsf{enter}^\sharp_e\ (\mathsf{get}\ [u,b]) \\
& \qquad\qquad\qquad\ \ () = \mathsf{set}\ [s_g, \langle d\rangle_2]\ d \\
& \qquad\qquad\qquad\ \ () = \mathsf{set}\ [v,b]\ (\mathsf{comb}^\sharp_e\ (\mathsf{get}\ [u,b])\ (\mathsf{get}\ [r_g, \langle d\rangle_2])) \\
& \qquad\qquad\quad \textbf{in}\ \bot \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \forall e = (u, g(), v) \in E
\end{aligned}
\tag{5'}
$$

The constraints for both statements s and function calls $g()$ are now given in the right-hand side for the source program point of CFG the edge, instead of the target point. The contribution to the source program point itself, however, is always $\bot$. The contribution to the target program point is given instead as a side-effect—with the function set—using the same computation as the normal contributions of (5) for the target program point.

Theorem 4. The constraint systems (5) and (5') are equivalent, which means that every solution to constraint system (5) is also a solution to constraint system (5'), and vice versa, every solution to constraint system (5') is also a solution to system (5). ∎

Proof. We are going to show that the second constraint in (5') has the same semantics as the second constraint in (2) (that is inherited by (5)), and that the last constraint from (5') has exactly the same semantics as the constraint form (5). All other cases are trivial.

a) The constraint (from (5'))

$$\begin{aligned} [u,b] \sqsupseteq \textbf{let}\ () = \mathsf{set}\ [v,b]\ (\llbracket s \rrbracket^\sharp\ (\mathsf{get}\ [u,b])) \qquad & \forall (u,s,v) \in E \\ \textbf{in}\ \bot \qquad & \end{aligned}$$

has only a bottom contribution to $[u,b]$, but it may have a contribution to $[v,b]$. It can, clearly, without losing information, be rewritten as

$$[v,b] \sqsupseteq \llbracket s \rrbracket^\sharp\ (\mathsf{get}\ [u,b])$$

With this operation we got exactly the second constraint in (2) (that is inherited by (5)). Note, that the rewriting can also be performed in the other direction.

b) Similarly to the previous case, the following constraint from (5')

$$\begin{aligned} [u,b] \sqsupseteq \textbf{let}\ & d = \mathsf{enter}_e^\sharp\ (\mathsf{get}\ [u,b]) \\ & () = \mathsf{set}\ [s_g, \langle d \rangle_2]\ d \\ & () = \mathsf{set}\ [v,b]\ (\mathsf{comb}_e^\sharp\ (\mathsf{get}\ [u,b])\ (\mathsf{get}\ [r_g, \langle d \rangle_2])) \\ \textbf{in}\ & \bot \\ & \qquad\qquad \forall e = (u, g(), v) \in E \end{aligned}$$

can be rewritten as

$$\begin{aligned}[v,b] \sqsupseteq \textbf{let } d = &\ \mathsf{enter}^{\sharp}_e\ (\mathsf{get}\ [u,b]) \\ () = &\ \mathsf{set}\ [s_g, \langle d\rangle_2]\ d \qquad\qquad \forall e = (u, g(), v) \in E \\ \textbf{in }&\ \mathsf{comb}^{\sharp}_e\ (\mathsf{get}\ [u,b])\ (\mathsf{get}\ [r_g, \langle d\rangle_2])\end{aligned}$$

■

Assume that local solving is applied to the constraint system (5'), and a variable $[u, b]$ has changed its value. Since for every constraint (but the very first one) the variable of the left-hand side also occurs on the right-hand side, all constraints for $[u, b]$ will be evaluated and the change propagated through the control-flow graph and into calls via side effects. If then a variable $[r_g, b']$ (corresponding to the return point of the procedure g) changes its value, re-evaluation will be triggered for every constraint for a corresponding call to procedure g and produce the required contributions to the end points of these calls.

Thus, the *operational* behavior of a local fixpoint solver applied to this system emulates the behavior of the original algorithm of Sharir and Pnueli. The advantage, though, is that this effect is not achieved by implementing a dedicated algorithm, but solely by changing the specification of the constraints. Moreover, this formulation is flexible enough to allow for an extension which deals with side-effects to globals as well.

3.15 Experimental Evaluation

Side-effecting constraint systems are at the heart of Goblint—an analyzer generator for concurrent C programs. This implementation of our Swiss army knife approach allows us to conduct experimental comparisons between configurations for the same analysis.

We considered a lockset analysis for concurrent C, where the goal is to guarantee absence of data races by accumulating for every global g, sets of definitely held static locks when accessing g. This analysis requires a detailed value analysis which provides points-to information for pointers as well as constant values for variables and resolves function pointers on-the-fly. Given that, the actual sets of definitely held locks are propagated and recorded at each access to a shared variable. In order to increase precision, path-sensitivity is added to relate conditional lock operations with corresponding conditional unlock operations [Das et al., 2002].

We considered a suite of the following concurrent programs using Posix threads:

aget	A multithreaded HTTP download accelerator, version 0.4.
automount	Autofs kernel-based automounter for Linux, version 5.0.2.
ctrace	C tracing library sample program, version 1.2.
knot	Knot web-server, stable release from SOSP CD.
pfscan	Parallel file scanner, version 1.0.
smtprc	A network open mail relay checker, version 2.0.3.
ypbind	Linux NIS daemon: ypbind-mt, version 1.19.1.
zfs-fuse	ZFS filesystem for FUSE/Linux: release 0.4.0_beta2.

The sizes of these benchmarks vary between 1280 LoC and 24097 LoC where LoC counts the lines of post-processed and merged C code.

For these benchmarks, we compared the analysis based on Cousot-style constraint system (5) (extended with dynamic function calls and side effects) with the analysis based on constraint system (5') corresponding to the forward propagating algorithm of Sharir and Pnueli. In both cases, we considered three instances where procedures are analyzed with full context, with partial context or no context, respectively. As partial context, we chose the information about the pointer variables together with the lockset information.

For benchmarking we used an OS X (10.9) laptop with a dual core 2.7GHz Intel Core i7 processor, of which the analyzer currently utilizes a single core, and 8.0GB DDR3 memory. For all benchmarks and all configurations, the analyzer performs reasonably well—the full context analysis of the 24kloc program zfs-fuse takes 6.6 seconds only.

For a fair comparison between the different configurations, we counted the number of evaluations of right-hand sides. Figure 3.11 displays these numbers for the given list of benchmarks, sorted according to their sizes. The Table 3.1 shows, for each program, the number of lines where a data race could not be ruled out.

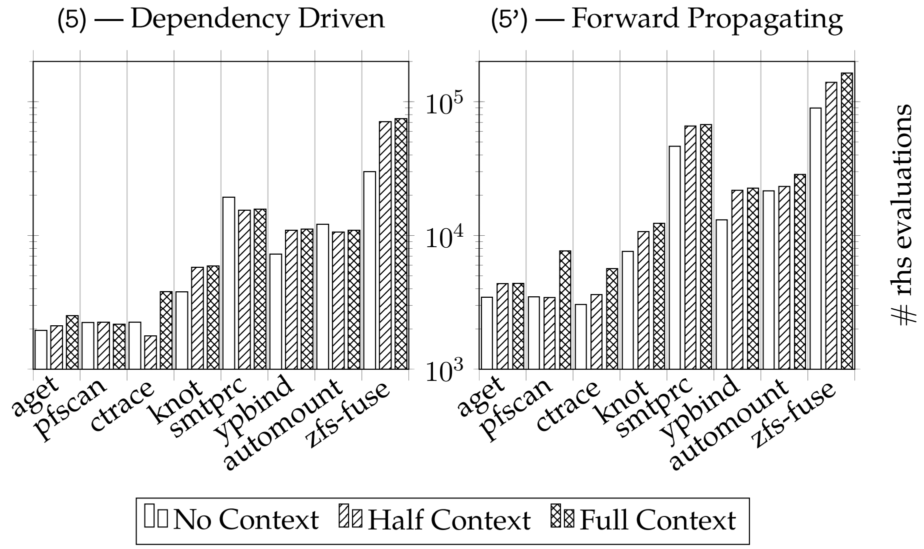


Figure 3.11: Timing results.

As expected, the analysis without context is less precise. Less expected, we found no difference in precision between full or half context in these benchmarks. Concerning precision, no distinction is made between dependency-driven solving and forward propagation, as these approaches produce the same result for the same context configuration.

Surprisingly, an analysis without context was not always the most efficient, i.e., for dependency driven analysis of smtprc fewer right-hand sides had to be evaluated in full context case than the no context case. This shows that trading the number of updates against the number of constraint variables does not necessarily pay off.

Generally, for these benchmarks forward propagation required the evaluation of significantly more right-hand sides than the dependency-driven approach. This confirms the intuition that forward propagation is prone to analyze the code, after two branches have met, twice.

		Lines with Warnings		
Name	Size(LoC)	No Ctx.	Half Ctx.	Full Ctx.
aget	1280	122	100	100
pfscan	1295	5	5	5
ctrace	1407	34	33	33
knot	2255	89	33	33
smtprc	5787	899	113	113
ypbind	6596	106	46	46
automount	20624	225	225	225
zfs-fuse	24097	1282	1281	1281

Table 3.1: Precision Results.

3.16 Conclusion

We have shown that side-effecting constraint systems are well-suited to express interprocedural program analyses with quite different characteristics such as flow insensitivity for global properties, as well as flow sensitivity for locals where procedure summaries are tabulated for varying parts of the abstract state.

Encoding different well-known techniques using side-effecting constraint systems allows us to freely combine different types of program analyses into a single analysis, while preserving soundness guarantees. This combination into a single analysis is critical for precision because different analyses may mutually complement each other. It also enables us to generically apply *property simulation* [Das et al., 2002], which we have used to deal with conditional locking, by letting the user select which analyses should be seen as the relevant property and which analyses are part of the simulation.

The approach through side-effecting constraint systems has been realized in the analyzer generator Goblint. Any analysis that can be expressed using side-effecting constraint system can be plugged into the analysis framework. An experimental evaluation on practical programs shows that competitive run times can be obtained for larger programs and quite complicated analyses.

4 Adding Widening and Narrowing

Right from the beginning of abstract interpretation, it has been observed that many interesting invariants are best expressed by means of lattices that have *infinite* strictly ascending chains. Possibly infinite strictly ascending chains, though, imply that naive Kleene iteration may not terminate. For that reason, Cousot and Cousot proposed a *widening* iteration, followed by a *narrowing* iteration in order to determine reasonably precise invariants or, technically speaking, reasonably small *post* solutions [Cousot and Cousot, 1976].

The widening phase can be considered as a Kleene iteration that is accelerated by means of a widening operator which enforces that only finitely many increases of values occur for a variable. While enforcing termination, it may result in a crude over-approximation of the invariants of the program. In order to compensate for that, the subsequent narrowing iteration tries to improve a given post solution by means of a downward fixpoint iteration, which again may be accelerated by means of a *narrowing* operator.

One problem with the approach is that trying to recover precision once it has been thrown away, though, in general is not possible (see, e.g., [Halbwachs and Henry, 2012] for a recent discussion). Some attempts try to improve precision by reducing the number of points where widening is applied [Cousot, 1981; Bourdoncle, 1993], others rely on refined widening or narrowing operators (see, e.g., [Simon and King, 2006; Cortesi and Zanioli, 2011]). Recent work has focused on methods to guide or stratify the exploration of the state space [Gopan and Reps, 2007, 2006; Gulavani et al., 2008; Monniaux and Guen, 2011; Henry et al., 2012], including techniques for automatic transformation of irregular loops [Gulwani et al., 2009; Sharma et al., 2011].

Our approach here is complementary to such techniques and can, possibly, be combined with these. Our idea is not to postpone narrowing to a second phase where all losses of information have already occurred and been propagated. Instead, an attempt is made to systematically improve the current information by narrowing immediately. In particular, this means that widening and narrowing are applied in an *interleaved* manner. A similar idea has been proposed for the static analyzer Astrée [Blanchet et al., 2003].

Narrowing and the narrowing operator are only guaranteed to return meaningful results, i.e., a post solution of the given system of equations, if the right-hand sides of equations are *monotonic*. This assumption, however, is violated in presence of widening. As we saw in the previous chapter, this requirement is also not met by the equation systems for context-sensitive interprocedural analysis. Moreover, for more complicated abstract domains, these systems may be infinite and thus can be handled by *local* solvers only. Local solvers query the value of an interesting variable and explore the space of variables only insofar as required for answering the query. For this type of algorithm, the set of evaluated variables is not known beforehand. In particular the values of new variables may be queried in the narrowing phase that have not yet been considered before. As a consequence, the strict separation into a widening and a narrowing phase can no longer be maintained. Although, narrowing iteration in case of non-monotonicity has been considered before [Cousot et al., 2007], such approaches are not directly applicable for local solving.

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

4.1 Classical widening/narrowing

Definition 4.1.1 (Widening operator).
For a lattice $(\mathbb{D}, \sqsubseteq)$, the binary operator $\sqcup\!\!\!\!\sqcup : \mathbb{D} \to \mathbb{D} \to \mathbb{D}$ is called a widening operator if the following two properties are satisfied:

a) for all $x, y \in \mathbb{D}$ it holds that $x \sqcup y \sqsubseteq x \sqcup\!\!\!\!\sqcup\, y$, and

b) for all increasing chains $x_0 \sqsubseteq x_1 \sqsubseteq \dots$, the increasing chain defined by

$$y_0 = x_0, \dots, y_{i+1} = y_i \sqcup\!\!\!\!\sqcup\, x_{i+1}, \dots$$

is not strictly increasing. ■

Example 5. The integer intervals $\mathbb{D} = (\mathbb{Z} \cup \{-\infty\}) \times (\mathbb{Z} \cup \{\infty\})$ are sets of pairs where the lower bound is in an integer or $-\infty$, and the upper bound an integer or ∞. The ordering $\sqsubseteq$ is ordinary interval inclusion.

Widening for intervals is classically defined as follows:

$$[x_l, x_u] \sqcup\!\!\!\!\sqcup\, [y_l, y_u] = [z_l, z_u]$$

where

$$z_l = \begin{cases} x_l & \text{if } x_l = y_l \\ -\infty & \text{otherwise} \end{cases} \quad \text{and} \quad z_u = \begin{cases} x_u & \text{if } x_u = y_u \\ \infty & \text{otherwise.} \end{cases}$$ ■

Definition 4.1.2 (Narrowing operator).
For a lattice $(\mathbb{D}, \sqsubseteq)$, the binary operator $\sqcap\!\!\!\!\sqcap : \mathbb{D} \to \mathbb{D} \to \mathbb{D}$ is called a narrowing operator if the following two properties are satisfied:

a) for all $x, y \in \mathbb{D}$ it holds that $(y \sqsubseteq x) \Rightarrow (y \sqsubseteq (x \sqcap\!\!\!\!\sqcap\, y) \sqsubseteq x)$, and

b) for all decreasing chains $x_0 \sqsubseteq x_1 \sqsubseteq \dots$, the decreasing chain defined by

$$y_0 = x_0, \dots, y_{i+1} = y_i \sqcap\!\!\!\!\sqcap\, x_{i+1}, \dots$$

is not strictly decreasing. ■

Example 6. Narrowing for intervals is classically defined as follows:

$$[x_l, x_u] \sqcap\!\!\!\!\sqcap [y_l, y_u] = [z_l, z_u]$$

where

$$z_l = \begin{cases} y_l & \text{if } x_l = -\infty \\ x_l & \text{otherwise} \end{cases} \qquad \text{and} \qquad z_u = \begin{cases} y_u & \text{if } x_u = \infty \\ x_u & \text{otherwise.} \end{cases} \qquad \blacksquare$$

Finding a post-fixpoint of the monotonic function $F : \mathbb{D} \to \mathbb{D} \to \mathbb{D}$ using the widening/narrowing approach [Cousot and Cousot, 1976, 1977a, 1992] consists of first computing the upward iteration sequence:

$$\begin{aligned} X_0 &= \bot \\ X_{i+1} &= X_i && \text{if } F\,X_i \sqsubseteq X_i \\ &= X_i \sqcup\!\!\!\!\sqcup F\,X_i && \text{otherwise.} \end{aligned}$$

The iteration starts ($i = 0$) with the least value $X_0 = \bot$, and then stays the same if $F\,X_i \sqsubseteq X_i$, otherwise increases the value using $X_i \sqcup\!\!\!\!\sqcup F\,X_i$. Because of the properties of the widening operator, it can be shown that the sequence is ultimately stationary, i.e., there exists n so that for all $m > n$, $X_n = X_m$. Additionally, the reached stationary point X_n is a post-fixpoint for the function F, i.e., $X_n \sqsupseteq F\,X_n$, but it might not be a fixpoint.

Next, a downward iteration is performed in the attempt to find a smaller post-fixpoint. Starting with the result from the upward iteration, it decreases the value using $Y_i \sqcap\!\!\!\!\sqcap F\,Y_i$.

$$\begin{aligned} Y_0 &= X_n \\ Y_{i+1} &= Y_i \sqcap\!\!\!\!\sqcap F\,Y_i \end{aligned}$$

The downward iteration is, also, ultimately stationary. Additionally, each step, although decreasing, is still a post-fixpoint in the case that F is monotonic.

Example 7. In the case that applying F leads to the following infinite chain

$F^0\,\bot$	$F^1\,\bot$	$F^2\,\bot$	$F^3\,\bot$	$\dots$
$\bot$	$[0,0]$	$[0,1]$	$[0,2]$	$\dots$

the upward iteration using $\sqcup$ will stabilize quickly:

X_0	X_1	X_2	X_3	$\ldots$
$\bot$	$[0,0]$	$[0,\infty]$	$[0,\infty]$	$\ldots$

■

In the case of non-monotone functions and domains with infinitely ascending chains, we cannot use narrowing—the newly computed value might be larger than the current value. Therefore, we would like to have a *simple* and *robust* technique that works as well as the known solvers for functions with good properties and gives its best effort for functions with bad properties.

4.2 Equation systems

Previously we talked about constraint systems—systems where each variable could have any number of constraints. This is in some cases practically preferred as one may want to only re-compute the constraints whose dependencies have changed their value. However, for use with the widening/narrowing approach, the equational form is needed.

Definition 4.2.1 (Equation system).
For a set of variables V, and values D, the function $\mathcal{F} \in V \to (V \to D) \to D$ is called an equation system. For each variable $x \in V$, the partially applied function $\mathcal{F}\, x$ represents the right-hand side of x. ■

The equational form is optional for the widening iteration. Because, conceptually, the widening operator throws away information, we try to use it in as few places as possible. Therefore, we still prefer to use widening for all the constraints together, not for each separate constraint.

For narrowing, however, it is absolutely necessary to combine all constraints for a variable, as narrowing with a single contribution from a constraint might lead to a variable assignment that is no longer a post-solution.

Note, that a constraint system $\mathcal{C} \subseteq V \times ((V \to \mathbb{D}) \to \mathbb{D})$ is a binary relation between left- and right-hand sides. So it is natural to look at a equation system $\mathcal{F}$ as a function from a left-hand side to the right-hand side $V \to (V \to \mathbb{D}) \to \mathbb{D}$. A constraint system $\mathcal{C}$ can be converted to equational form using the following scheme:

$$\mathcal{F}\, x\, \sigma = \bigsqcup \{ f\, \sigma \mid \forall f,\ (x, f) \in \mathcal{C} \}\ .$$

4.3 Chaotic fixpoint iteration

For static program analysis, constraint systems over complete lattices are used [Cousot and Cousot, 1977a], but equation systems can be more general. Therefore, we generalize the definitions for equation systems in a way that allows to use other mathematical structures.

We consider a system $\mathcal{F}$ of equations, for a set of variables $x \in V$, and over a set D of values. Furthermore, let $\Box : D \to D \to D$ be a binary operator to combine old values with the new contributions of the right-hand sides.

Definition 4.3.1 ($\Box$-solution).
A $\Box$-solution of $\mathcal{F}$ is an assignment $\rho \in V \to D$ such that for all variables $x \in V$, $\rho[x] = \rho[x] \Box \mathcal{F}\, x\, \rho$ holds. In the case that $\Box$ is defined as $a \Box b = b$, a $\Box$-solution is an ordinary solution of the system, i.e., a mapping ρ with $\rho[x] = \mathcal{F}\, x\, \rho$ for all variables x. ■

In case D is a lattice, and the $\Box$-operator is defined as the least upper bound operator $\sqcup$, a $\Box$-solution is a post-solution of the system, i.e., a mapping ρ with $\rho[x] \sqsupseteq \mathcal{F}\, x\, \rho$ for all variables x. Likewise in case $\Box$ equals the greatest lower bound operator $\sqcap$, a $\Box$-solution is a *pre*-solution of the system, i.e., a mapping ρ with $\rho[x] \sqsubseteq \mathcal{F}\, x\, \rho$ for all variables x.

The operator $\Box$ can also be instantiated with widening and narrowing operators. As stated previously, a widening operator $\sqcup\!\!\!\!\sqcup$ for a lattice D must satisfy that $a \sqcup b \ \sqsubseteq\ a \sqcup\!\!\!\!\sqcup b$ for all $a, b \in D$. This implies that a $\Box$-solution then again provides a post-solution of the original system $\mathcal{F}$.

The situation is slightly more complicated for narrowing operators. For a narrowing operator $\sqcap\!\!\!\!\sqcap$, $a \sqsupseteq b$ implies that $a \sqsupseteq (a \sqcap\!\!\!\!\sqcap b) \sqsupseteq b$ and any narrowing sequence cannot be strictly descending. This means that narrowing can only be applied if the right-hand side of equations are guaranteed to return values that are less than or equal to than the values of the current left-hand sides. Thus a mapping ρ can only be a $\sqcap$-solution, if it is a post-solution of the system.

Definition 4.3.2 (chaotic solver).
A (chaotic) solver for systems of equations is an algorithm that maintains a mapping $\rho \in V \to D$ and performs a sequence of *update steps*, starting from an initial mapping ρ_0. Each update step selects an variable x, evaluates the right-hand side $\mathcal{F}\,x$ of x w.r.t. the current mapping ρ_i and updates the value for x, i.e.,

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

Then the algorithm is a $\square$-solver if upon termination the final mapping (after completing n steps) ρ_n is a $\square$-solution of $\mathcal{F}$. The algorithm is a *generic* solver if it works for any binary update operator $\square$. ∎

In this sense, the round-robin iteration of Figure 4.1 is a generic solver. Note that, in most cases, we omit update step indices and, additionally, use imperative assignment syntax of the form $\rho[x] \leftarrow w$ to change the value of the variable x to w

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

Not every solver algorithm, though, may consider right-hand sides of equations as black boxes, as the round-robin algorithm does. The worklist algorithm from Figure 4.2 can only be used as a generic solver—given that all dependences are provided before-hand. In the case that the value of some variable y changes, all right-hand sides of variables in the set infl_y must be re-computed.

Note that whenever an update to a variable y provides a new value, $\mathbf{W}_\square$ re-schedules y for evaluation as well. This is a precaution for the case

```
do
  dirty ← *false*;
  foreach $x \in$ V do
    new ← $\rho[x] \,\square\, \mathcal{F}\; x\; \rho$;
    if $\rho[x] \neq$ new then
      $\rho[x]$ ← new;
      dirty ← *true*;
while (dirty)
```

Figure 4.1: The solver $\mathbf{RR}_\square$.

```
W ← V;
while W $\neq \emptyset$ do
  $x$ ← extract(W);
  new ← $\rho[x] \,\square\, \mathcal{F}\; x\; \rho$;
  if $\rho[x] \neq$ new then
    $\rho[x]$ ← new;
    W ← W $\cup$ infl$_x$
done
```

Figure 4.2: The Solver $\mathbf{W}_\square$.

that the operator $\square$ is *not* (right) idempotent. Here, an operator $\square$ is called *idempotent* if the following equality:

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

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

4.4 Two-phased widening/narrowing

The simplest way to apply widening/narrowing to a monotonic equation system $\mathcal{F}$ over the complete lattice $\mathbb{D}$ is to proceed in two-phases—similarly to the iterations in Section 4.1. Using a generic chaotic solver, we first generate a $\sqcup\!\!\!\!\!\,\sqcup$-solution σ^+ of $\mathcal{F}$ starting with the value $\bot$ for all variables. After that, we generate a $\sqcap\!\!\!\!\!\,\sqcap$-solution σ^-, where we start with the values $\sigma^+\, x$ for each variable $x \in V$.

Fact 4.4.1. Assume that all right-hand sides of the system $\mathcal{F}$ of equations over a lattice $\mathbb{D}$ are monotonic and that ρ_0 is a post-solution of $\mathcal{F}$, and $\square$ is a narrowing operator $\sqcap\!\!\!\!\!\,\sqcap$. Then the sequence $\rho_0, \rho_1, \ldots$ of mappings produced by a generic $\square$-solver, is defined and decreasing. ■

Thus, any generic solver can be applied to improve a post-solution by means of a narrowing iteration—given that all right-hand sides of equations are monotonic. This means that both variable assignments σ^+ and

σ^- will be post-solutions of the system $\mathcal{F}$, and additionally, the variable assignment σ^- is no larger than σ^+.

4.5 The combined update operator

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

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

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

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

and therefore also

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

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

Lemma 1. Consider a finite system $\mathcal{F}$ of equations over a lattice $\mathbb{D}$. Then every $\boxminus$-solution ρ of $\mathcal{F}$ is a post-solution of $\mathcal{F}$, i.e., for all variables x, $\rho[x] \sqsupseteq \mathcal{F}\, x\, \rho$. ■

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

$$\rho[x] \;=\; \rho[x] \boxminus \mathcal{F}\, x\, \rho \;=\; \rho[x] \sqcup\!\!\!\!\sqcup\, \mathcal{F}\, x\, \rho \;\sqsupseteq\; \mathcal{F}\, x\, \rho$$

in contradiction to our assumption! Accordingly, ρ must be a post-solution of the system of equations $\mathcal{F}$. ■

Thus, every generic solver for lattices $\mathbb{D}$ can be turned into a solver computing post-solutions by using the combined widening and narrowing operator. The intertwined application of widening and narrowing, which naturally occurs when solving the system of equations by means of $\boxminus$, has the additional advantage that values may also *shrink* in-between. Improving possibly too large values, thus, may take place immediately resulting in overall smaller, i.e., better post-solutions. Moreover, no restriction is imposed any longer concerning monotonicity of right-hand sides.

4.6 Enforcing termination

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

Example 8. Consider the system:

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

with $\mathbb{D} = \mathbb{N} \cup \{\infty\}$, the lattice of non-negative integers, equipped with the natural ordering $\sqsubseteq$ given by $\leq$ and extended with ∞.

Consider a widening $\sqcup\!\!\!\!\sqcup$ where $a \sqcup\!\!\!\!\sqcup b = a$ if $a = b$ and $a \sqcup\!\!\!\!\sqcup b = \infty$ otherwise, together with a narrowing $\sqcap\!\!\!\!\sqcap$ where, for $a \geq b$, $a \sqcap\!\!\!\!\sqcap b = b$ if $a = \infty$, and $a \sqcap\!\!\!\!\sqcap b = a$ otherwise. Round-robin iteration with the combined operator $\boxminus$ for this system starting from the mapping $\rho_0 = \{x_1 \mapsto 0, x_2 \mapsto 0, x_3 \mapsto 0\}$, will produce the following sequence of mappings:

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

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

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

Example 9. Consider the two equations:

$$\begin{aligned} x_1 &= (x_1 + 1) \sqcap (x_2 + 1) \\ x_2 &= (x_2 + 1) \sqcap (x_1 + 1) \end{aligned}$$

using the same lattice as in Example 8 where $\sqcap$ denotes minimum, i.e., the greatest lower bound. Assume that the work-set is maintained with a LIFO discipline. For $W = [x_1, x_2]$, worklist iteration, starting with the initial mapping $\rho_0 = \{x_1 \mapsto 0, x_2 \mapsto 0\}$, results in the following iteration sequence:

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

which does not terminate. ■

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

For both algorithms, we assume that we are given a fixed *linear ordering* on the set of variables so that $V = \{x_1, \ldots, x_n\}$. The ordering will affect the iteration strategy, and therefore, as shown by Bourdoncle [1990], has a significant impact on performance. Hence, the linear ordering should be chosen in a way that innermost loops would be evaluated before iteration on outer loops. For variables x_i for $i = 1, \ldots, n$, the new round-robin algorithm is shown in Figure 4.3.

Let us call the new algorithm $\mathbf{SRR}_\square$ (*structured* round-robin). For a given initial mapping ρ_0, structured round-robin is started by calling solve n. The idea of the algorithm is, when called for a number i, to iterate on the variable x_i until stabilization. Before every update of the variable

```
void solve($i$) {
  if $i$ = 0 then return;
  solve($i-1$);
  new ← $\rho[x_i] \,\Box\, \mathcal{F}\, x_i\, \rho$;
  if $\rho[x_i] \neq$ new then
    $\rho[x_i]$ ← new;
    solve($i$);
}
```

Figure 4.3: The new solver $\mathbf{SRR}_\Box$.

```
$Q \leftarrow \emptyset$;
for $i$ = 1 to n do add $Q\ x_i$;
while $Q \neq \emptyset$ do
  $x_i$ ← extract_min($Q$);
  new ← $\rho[x_i] \,\Box\, \mathcal{F}\, x_i\, \rho$;
  if $\rho[x_i] \neq$ new then
    $\rho[x_i]$ ← new;
    foreach $x_j \in$ infl$_i$
do add $Q\ x_j$
done
```

Figure 4.4: The new solver $\mathbf{SW}_\Box$.

x_i, however, all variables x_j, $j < i$ are recursively solved. Clearly, the resulting algorithm is a generic $\Box$-solver.

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

Theorem 5. Consider the algorithm $\mathbf{SRR}_\Box$ for a system of n equations over a complete lattice where all right-hand sides are monotonic and $\Box = \boxminus$. Then the following holds:

a) Assume that the lattice has bounded height h and $\sqcup\!\!\!\!\sqcup = \sqcup$. Then $\mathbf{SRR}_\Box$ when started with the initial mapping $\rho_0 = \{x_i \mapsto \bot \mid i = 1, \ldots, n\}$, terminates after at most $n + \frac{h}{2}n(n+1)$ evaluations of right-hand sides.

b) Also in presence of unbounded ascending chains, the algorithm $\mathbf{SRR}_\Box$ will terminate for *every* initial mapping. ∎

Proof. Recall that ordinary round-robin iteration performs at most $h \cdot n$ rounds due to increases of values of variables plus one extra round to detect termination, giving in total

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

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

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

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

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

giving us statement *a*).

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

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

Example 10. Recall the equation system, for which round-robin iteration did not terminate. With structured round-robin iteration, however, we obtain the following sequence of updates:

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

where the evaluations of variables not resulting in an update have been omitted. Thus, structured fix-point solving quickly stabilizes for this example. ■

The idea of structured iteration can also be lifted to worklist iteration. Consider again a system with $V = \{x_1, \ldots, x_n\}$. As before for each variable x_j, let infl_j denote the the set consisting of the variable x_j together with all variables influenced by x_j. Instead of a plain worklist, the modified algorithm maintains the set of variables to be reevaluated within a *priority queue* Q. In every round, not an arbitrary element is extracted from Q—but the variable with the least index. The resulting algorithm is presented in Figure 4.4.

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

Let us call the resulting algorithm $\mathbf{SW}_\square$ (structured worklist iteration). Clearly, the resulting algorithm is a generic solver.

Example 11. Consider again the system from example Example 9. Structured worklist iteration using $\boxminus$ for this system results in the following iteration:

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

and thus terminates. ■

In general, we have:

Theorem 6. Assume the algorithm $\mathbf{SW}_\square$ is applied to a system of equations over a complete lattice $\mathbb{D}$ and that each right-hand side is monotonic.

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

b) When instantiated with $\square = \boxminus$ and started on any mapping, the algorithm is guaranteed to terminate and, thus, always to return a post-solution. ■

The first statement of the theorem indicates that $\mathbf{SW}_\square$ behaves complexity-wise like ordinary worklist iteration: the only overhead to be paid for is an extra logarithmic factor for maintaining the priority queue. The second statement, perhaps, is more surprising: it provides us with a termination guarantee for the operator $\boxminus$.

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

For a contradiction assume that $\mathbf{SW}_\square$ does not terminate for the system of equations for V. First assume that the variable x_n is extracted from the queue Q only finitely many times, say k times where d is the last value computed for x_n. This means that after the last extraction, an infinite iteration occurs on the subsystem on the variables $V' = V \setminus \{n\}$ where for $x_r \in V'$, the right-hand side is given by $\mathcal{F}'\, x_r\, \rho = \mathcal{F}\, x_r\, (\rho \oplus \{x_n \mapsto d\})$. By inductive hypothesis, however, the algorithm $\mathbf{SW}_\square$ for this system terminates—in contradiction to our assumption. Therefore, we may assume that the variable x_n is extracted infinitely often from Q.

Let $\rho_i, i \in \mathbb{N}$, denote the sequence of mappings at these extractions. Since Q is maintained as a priority queue, we know that for all variables x_r

with $r < n$, the inequalities $\rho_i[x_r] \sqsupseteq \mathcal{F}\, x_r\, \rho_i$ hold. Let $d_i = \rho_i[x_n]$. If for any i, $f_n\, \rho_i \sqsubseteq d_i$, the next value d_{i+1} for x_n then is obtained by $d_{i+1} = d_i \sqcap\!\!\!\!\sqcap\, \mathcal{F}\, x_n\, \rho_i$ which is less or equal to ρ_i. By monotonicity, this implies that in the subsequent iteration, the values for all variables $x_r, r \leq n$, may only decrease. The remaining iteration is a pure narrowing iteration and therefore terminates.

In order to obtain an infinite sequence of updates for z, we conclude that for no i, $\mathcal{F}\, x_n\, \rho_i \sqsubseteq d_i$. Hence for every i, $d_{i+1} = d_i \sqcup\!\!\!\!\sqcup\, \mathcal{F}\, x_n\, \rho_i$ where $d_i \sqsubseteq d_{i+1}$. This, however, is impossible due to the properties of the widening operator. In summary, we conclude that x_n is extracted only finitely often from Q. Hence the fixpoint iteration terminates. ■

Since the algorithm $\mathbf{SW}_{\square}$ is a generic solver, it can also be applied to non-monotonic systems. There, however, termination can no longer be guaranteed. One generic idea, though, to enforce termination for *all* $\boxminus$-solvers and in all cases, is to equip each variable with a separate counter that counts how often the solver has switched from narrowing back to widening. That number then may be taken into account by the $\boxminus$-operator, e.g., by choosing successively less aggressive narrowing operators $\sqcap\!\!\!\!\sqcap_0, \sqcap\!\!\!\!\sqcap_1, \ldots$, and, ultimately, to give up improving the obtained values. The latter is achieved by defining $a \sqcap\!\!\!\!\sqcap_k b = a$ for a certain threshold k.

4.7 Local generic solvers

Similar to generic solvers, we define generic *local* solvers. For that, it seems convenient that the *dynamic* dependences between variables are approximated. For the left-hand side x and a mapping ρ, a set $\mathsf{dep}_x \subseteq V$ subsumes all dynamic dependences of a function $\mathcal{F}\, x : (V \to \mathbb{D}) \to \mathbb{D}$ w.r.t. ρ in the case that $\mathcal{F}\, x\, \rho = \mathcal{F}\, x\, \rho'$ whenever $\rho'|_{\mathsf{dep}_x} = \rho|_{\mathsf{dep}_x}$. Such sets can be constructed on the fly whenever the function $\mathcal{F}\, x$ is *pure* in the sense of [Hofmann et al., 2010a].

Essentially, purity for a right-hand side f means that evaluating f for a mapping ρ operationally consists of a finite sequence of value lookups in ρ where the next variable whose value has to be looked up, may only depend on the values that have already been queried. Once the sequence of lookups has been completed, the final value is determined, depending on the sequence of values, and finally returned.

In this case, the set X' can be chosen as the set of all variables y for which the value $\rho\, y$ is queried when evaluating (an implementation of) the function f for the argument ρ. Let us denote this set by $\mathsf{dep}_x\, \rho$.

Definition 4.7.1 (Partial □-solution).
A *partial* □-solution of an (infinite) system of pure equations $\mathcal{F}$ is a set $\mathsf{dom} \subseteq V$ and a mapping $\rho : \mathsf{dom} \to D$ with the following two properties:

a) $\rho[x] = \rho[x] \,\square\, (\mathcal{F}\, x\, \rho)$ for all $x \in \mathsf{dom}$; and

b) $\mathsf{dep}_x\, \rho \subseteq \mathsf{dom}$ for all $x \in \mathsf{dom}$. ■

In essence, a partial □-solution is a set dom and a □-solution of the subsystem of $\mathcal{F}$ restricted to variables in dom.

Example 12. The following equation system (for $n \in \mathbb{N} = D$)

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

is infinite as it uses infinitely many variables. However, the equation system has at least one finite partial max-solution—the set $\mathsf{dom} = \{y_1, y_2, y_4\}$ together with the mapping $\rho = \{y_1 \mapsto 2, y_2 \mapsto 2, y_4 \mapsto 2\}$. ■

Definition 4.7.2 (Local generic □-solver).
A *local* generic solver instantiated with an operator □, then, is an algorithm that, when given a system of pure equations $\mathcal{F}$, a initial mapping ρ_0 for all variables, and an variable $x_0 \in V$, performs a sequence of update steps that, upon termination, results in a partial □-solution (dom, ρ), such that $x_0 \in \mathsf{dom}$. ■

```
let rec solve $x$ =
  if $x \notin$ stable then
    stable $\leftarrow$ stable $\cup \{x\}$;
    tmp $\leftarrow \sigma[x] \sqcup \mathcal{F}\, x$ (eval $x$);
    if tmp $\neq \sigma[x]$ then
      W $\leftarrow$ infl[$x$];
      $\sigma[x] \leftarrow$ tmp;
      infl[$x$] $\leftarrow \emptyset$;
      stable $\leftarrow$ stable $\setminus$ W;
      foreach $x \in$ W do solve $x$
    end
  end
and eval $x$ $y$ =
  solve $y$ ;
  infl[$y$] $\leftarrow$ infl[$y$] $\cup \{x\}$;
  $\sigma[y]$
in
  stable $\leftarrow \emptyset$;
  infl $\leftarrow \emptyset$;
  $\sigma \leftarrow \emptyset$;
  solve $x_0$;
  $\sigma$
```

Figure 4.5: The solver **RLD**$_\sqcup$ from [Hofmann et al., 2010b].

4.8 The structured local recursive solver

One more elaborate algorithm for local solving is formalized by Hofmann et al. [2010b], namely the solver **RLD**$_\sqcup$ as shown in Figure 4.5. This algorithm has the benefit of visiting nodes in a more efficient order, first stabilizing innermost loops before iterating on outer loops. However, when enhanced with an operator $\Box$, this algorithm is *not* a generic solver in our sense, since it is not guaranteed to execute as a sequence of *atomic* updates. Due to the recursive call to procedure solve at the beginning of eval, one evaluation of a right-hand side may occur nested into the evaluation of another right-hand side. Therefore, conceptually, it may happen that an evaluation of a right-hand side uses the values of variables from several different mappings ρ_i from the sequence $\rho_0, \rho_1, \ldots, \rho_n$, instead of the latest mapping ρ_n. Accordingly, the algorithm **RLD**$_\Box$ is not guaranteed to return a $\Box$-solution—even if it terminates. Here, we therefore, provide a variant of **RLD**$_\sqcup$ where right-hand sides (conceptually) are executed atomically.

Clearly, a local generic solver does not terminate if infinitely many variables are encountered. Therefore, a reasonable local solver will try to consider as few variables as possible. Our solver, thus, explores the values of variables by recursively descending into solving variables *newly* detected while evaluating a right-hand side.

Certain equation systems, though, introduce infinite chains of dependences for the variables of interest. Those systems then cannot be solved by any local solver. Here, we show that the new solver is guaranteed to terminate for the operator $\boxminus$ at least for equation systems which are monotonic and either finite or infinite but where only finitely many variables are encountered.

Let us call the new solver, on Figure 4.6, $\mathbf{SLR}^1_\square$ (*structured local recursive* solver). The new algorithm maintains an explicit set $\mathsf{dom} \subseteq X$ of variables that have already been encountered. Beyond $\mathbf{RLD}_\sqcup$, it additionally maintains a counter count which counts the number of elements in dom, and a mapping $\mathsf{key} : \mathsf{dom} \to \mathbb{Z}$ that equips each variable with its priority. Moreover, a global assignment $\mathsf{infl} : \mathsf{dom} \to 2^X$ records for each variable in $y \in \mathsf{dom}$, the variable y itself together with the set of variables $x \in \mathsf{dom}$ with the following two properties:

- the last evaluation of $\mathcal{F}\,x$ has accessed the variable y;
- since then, the value of the variable y has not changed.

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

As in the algorithm $\mathbf{RLD}_\sqcup$, right-hand sides $\mathcal{F}\,x$ are not directly evaluated for the current mapping ρ, but instead for a helper function eval which in the end, returns values for variables. Before that, however, the helper function eval provides extra book keeping of the encountered dependence between variables. Also, if the encountered variable is new, then it tries not just to return the value $\bot$, but to compute the best possible value for the new variable before-hand.

In order to be able to track dependences between variables, the helper function eval receives as a first argument the variable x whose right-hand side is under evaluation. The function eval first checks whether the variable y is already contained in the domain dom of ρ. If this is not the

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

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

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

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

Figure 4.6: The new solver $\mathbf{SLR}^1_\Box$.

case, y is first initialized by calling the procedure init. Subsequently, the best possible value for y is computed by calling the procedure solve for y. Then eval records the fact that x depends on y, by adding x to the set infl[y]. Only then is the corresponding value $\rho[y]$ returned.

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

The main fixpoint iteration is implemented by the procedure solve. When solve is called for a variable x, we assume that there is currently no variable $x' \in$ dom with key[x'] $<$ key[x] that violates its equation, i.e., for which $\rho[x'] \neq \rho[x'] \,\Box\, (\mathcal{F}\; x'\; \rho)$ holds. In the procedure solve for x, the call min_key Q returns the minimal key of an element in Q, and extract_min Q returns the variable in Q with the minimal key and additionally removes it from Q.

Besides the global priority queue Q, the procedure solve also requires a set stable. For all variables x' in stable $\subseteq$ dom one of the following properties holds at each call of the procedure solve:

- a call to the procedure solve x' has been started and the update of $\rho[x']$ has not yet occurred; or
- the equality $\rho[x'] = \rho[x'] \mathbin{\square} (\mathcal{F}\, x'\, \rho)$ holds.

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

Example 13. Consider again the infinite equation system from Example 12. The solver $\mathbf{SLR}^1_{\sqcup}$, when solving for y_1, will return the partial max-solution $\{y_0 \mapsto 0, y_1 \mapsto 2, y_2 \mapsto 2, y_4 \mapsto 2\}$. ■

The modifications of the algorithm $\mathbf{RLD}_{\sqcup}$ to obtain algorithm $\mathbf{SLR}^1_{\square}$ allow us not only to prove that it is a generic local solver, but also a strong result concerning termination. Our main theorem of this chapter is:

Theorem 7. The following two statements about $\mathbf{SLR}^1_{\square}$ hold:

a) When applied to any system of pure equations and an interesting variable x_0, the algorithm $\mathbf{SLR}^1_{\square}$ returns a partial $\square$-solution—whenever it terminates.

b) Assume that $\mathbf{SLR}^1_{\square}$ is applied to a system of pure equations over a complete lattice $\mathbb{D}$ where each right-hand side is monotonic. If the operator $\square$ is instantiated with $\boxminus$, then for any initial mapping ρ_0 and interesting variable x_0, $\mathbf{SLR}^1_{\square}$ is guaranteed to terminate and thus always to return a partial post solution—whenever only finitely many variables are encountered. ■

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

In order to prove that $\mathbf{SLR}^1_\Box$ is a local generic solver, it therefore remains to verify that upon termination, ρ is a partial $\Box$-solution with $x_0 \in \mathsf{dom}$. Since x_0 is initialized before solve x_0 is called, x_0 must be contained in dom. Upon termination, no evaluation of a variable is still in process and the priority queue is empty. All variables in $\mathsf{dom} \setminus \mathsf{stable}$ are either fresh and therefore solved right-away, or non-fresh and then inserted into the priority queue. Therefore, we conclude that the equation $\rho[x] = \rho[x] \,\Box\, (\mathcal{F}\,x\,\rho)$ holds for all $x \in \mathsf{dom}$. Furthermore, the invariant for the map infl implies that upon termination, $x \in \mathsf{infl}[y]$ whenever $x = y$ or $y \in \mathsf{dep}_x\,\rho$. In particular, infl is defined for y implying that $y \in \mathsf{dom}$.

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

For a proof of statement *b*), assume that all equations are monotonic and only finitely many variables are encountered during the call solve x_0. Let dom denote this set of variables. We proceed by induction on key values of variables in dom. First consider the variable $x \in \mathsf{dom}$ with minimal key value. Then for all mappings ρ and infl, the call solve x will perform a sequence of updates to $\rho[x]$. In an initial segment of this sequence, the operator $\boxminus$ behaves like $\sqcup$. As soon as the same value $\rho[x]$ or a smaller value is obtained, the operator $\boxminus$ behaves like the operator $\sqcap$. Due to monotonicity, the remaining sequence may only consist of narrowing steps.

By the properties of widening and narrowing operators, the sequence therefore must be finite.

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

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

4.9 Localized $\boxminus$ in SLR

So far we have applied the operator $\boxminus$ at every right-hand side. It has been long known for the 2-phase widening and narrowing approach, however, that precision can be gained by applying widening and thus also narrowing only at selected variables. In case, the equation systems are derived from control-flow graphs, for example, it suffices to apply these operators at one node of each loop only. For the more general setting of systems of equations with static dependences between variables, a sufficiently large set of variables where widening and narrowing is to be placed, can be determined by means of a weakly topological ordering [Bourdoncle, 1993].

In our application where variable dependences may change, we cannot perform any pre-computation on the dependence graph between variables. What we are given, though, is an assignment key of variables to priorities together with a *linear* ordering on the priorities. This enables us to dynamically identify *back-edges*. Here, a back-edge $x \to y$ consists of variables x, y where the value of x is queried in the right-hand side of y where $\mathsf{key}[x] \geq \mathsf{key}[y]$. Note that this does not correspond to the standard definition of back-edge, but we use the same terminology since both may be used to identify the head of loops. Evaluations within which no back-edges are encountered will necessarily terminate (given that the evaluation of each single right-hand side terminates). Therefore, it suffices to apply the operator $\boxminus$ at variables x only for which a back-edge $x \to y$ has been detected. The resulting improvement to the solver, when applied to the algorithm $\mathbf{SLR}^1_\square$, will be called $\mathbf{SLR}^2_\square$.

Interestingly for our suite of benchmark programs, the algorithm $\mathbf{SLR}^2_\square$ did not significantly improve the precision of the resulting interval analysis.

Example 14. Consider, e.g., the program in Figure 4.8. The control-flow graph corresponding to this program is also shown in the same figure where each node v is marked with the priority assigned to v when the function solve of $\mathbf{SLR}^1_\square$ is called for the endpoint of the program for an interval analysis.

We are looking for nodes that influence nodes with smaller priority. In the example, these are the nodes with priorities -1 and -5, respectively, i.e., exactly the loop heads.

After the first iteration for interval analysis on this program, the interval $[0, 0]$ has been established for the program variable i at all program points of the inner loop. Then a second iteration of the outer loop is performed. Even if the operator $\boxminus$ is only applied at the loop heads, we obtain the interval $[0, \infty]$ for i at the loop head of the outer loop. In the subsequent iteration of the inner loop, the new interval for variable i at the inner loop head is $[0, 99]$.

Since the update operator $\boxminus$ is meant to be applied at that program point -5, the interval $[0, 0] \boxminus [0, 99] = [0, \infty]$ is recorded for i and subsequently also propagated to all other program points of the inner loop. Subsequent narrowing will *not* be able to recover from the loss of the upper bound for i. ∎

Such loss of precision, as seen in Example 14, could be avoided, if we allow the set wpoint of variables where to apply $\boxminus$ not only to grow monotonically, but also to *shrink*. Our second idea therefore is to *remove* a variable x from wpoint before the right-hand side of x is evaluated. The resulting algorithm $\mathbf{SLR}^3_{\Box}$ is shown in Figure 4.7. Note that back-edges are detected by the call eval x y which therefore may insert y into the set wpoint, while the variable x is removed from wpoint inside the call solve x.

Theorem 8. When applied to any system of pure equations over a complete lattice $\mathbb{D}$ and interesting variable x_0, the algorithm $\mathbf{SLR}^3_{\Box}$ returns a post solution, whenever it terminates. If each right hand side is monotonic, then $\mathbf{SLR}^3_{\Box}$ is guaranteed to terminate, whenever only finitely many variables are encountered. ∎

Proof. The considerations in the original proof for $\mathbf{SLR}^1_{\Box}$ regarding atomicity of evaluation of right-hand sides still hold. The same is true for partial correctness. The only difference w.r.t. $\mathbf{SLR}^3_{\Box}$ is that, upon termination, for a variable x either $\rho[x] = \rho[x] \boxminus \mathcal{F}\,x\,\rho$ or $\rho[x] = \mathcal{F}\,x\,\rho$. In both cases, ρ is a post-solution.

The most interesting part is the proof of termination. So, assume all right hand sides are monotonic and only finitely many variables are encountered during the call of solve x_0. Assume the algorithm does not terminate. It means there are variables x whose value $\rho[x]$ is updated infinitely many times. Let x be one of such variables, namely the one with maximum priority. From a certain point in the execution of the algorithm, no fresh variable is encountered and no $\rho[y]$ for a variable y with priority bigger than x is ever updated.

Assume we have reached this point in the execution of the algorithm. Moreover, assume that x is extracted. This means that in the queue there are no variables with priority less than y. Since all variables with priorities greater than x are not subject to update (hence their evaluation does not add elements to the queue), but x should be extracted again, the only possibility is that:

```
let rec solve $x$ =
  wpx $\leftarrow$ if $x \in$ wpoint then *true* else *false*;
  wpoint $\leftarrow$ wpoint $\setminus \{x\}$;
  if $x \notin$ stable then
    stable $\leftarrow$ stable $\cup \{x\}$;
    tmp $\leftarrow$ if wpx
      then $\rho[x] \boxminus \mathcal{F}\ x$ (eval $x$)
      else $\mathcal{F}\ x$ (eval $x$)
    if tmp $\neq \rho[x]$ then
       $\rho[x] \leftarrow$ tmp;
       W $\leftarrow$ if wpx then infl$[x] \cup \{x\}$ else infl$[x]$;
       foreach $y \in$ W do add $Q$ $y$;
       infl$[x] \leftarrow \emptyset$;
       stable $\leftarrow$ stable $\setminus$ W;
       while $(Q \neq \emptyset) \wedge$ (min_key $Q \leq$ key$[x]$) do
         solve (extract_min $Q$);

and init $y$ =
  *as in the original* ***SLR***$^1_\square$

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

in
  wpoint $\leftarrow \emptyset$
  *as in the original* ***SLR***$^1_\square$
```

Figure 4.7: The algorithm **SLR**$^3_\square$, which is **SLR** with simple localized widening. Colored in red are then changes w.r.t. **SLR**$^1_\square$.

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

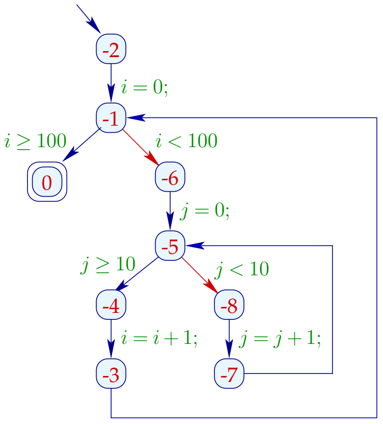


Figure 4.8: Example program with nested loops and its control-flow graph.

a) in solve x should be tmp $\neq \rho[x]$;

b) there is $y \in$ infl$[x]$ with key$[y] \leq$ key$[x]$, and y is put in the queue.

When y is evaluated (it will happen before solve x is called again), x will be added to wpoint, hence wpx will always be true when evaluating solve x. However, by properties of $\boxminus$, this means that x cannot be updated infinitely many times, which is an absurd. Therefore the algorithm terminates.

Example 15. Let us again consider the program from Figure 4.8. The solver $\mathbf{SLR}^3_\Box$ iterates through the program points of the inner loop until stabilization before the next iteration on the program points of the outer loop is performed. After this iteration, the interval $[0, 0]$ has been established for the program variable at all program points of the inner loop. Since the variable corresponding to the loop head of the inner loop is now stable, it is no longer contained in the set wpoint.

When during the next iteration of the outer loop the interval $[0, 99]$ arrives for program variable i, this interval will replace the current interval $[0, 0]$ for i (without application of the operator $\boxminus$). Accordingly, the subsequent iteration on the inner loop will propagate this interval throughout the

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

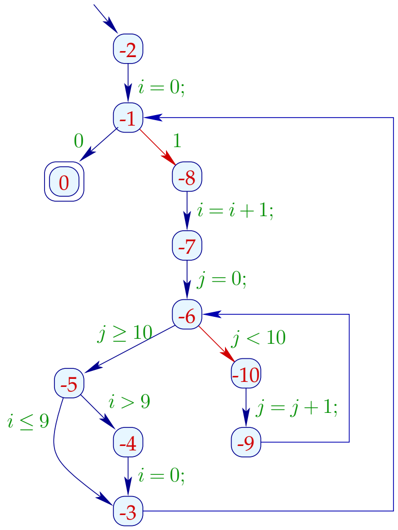


Figure 4.9: Example program hybrid from [Halbwachs and Henry, 2012] together with its control-flow graph.

inner loop without change. Therefore no upper bound ∞ for i is ever generated within the inner loop. This effect is comparable to the concept of localized widening as proposed by Amato and Scozzari [2013]. ■

4.10 Restarting in SLR

Besides localization of widening and narrowing, Amato and Scozzari [2013] present a second idea to improve precision of fixpoint iteration in presence of infinite increasing chains.

Example 16. Consider the program in Figure 4.9. In this example, the program variable i takes values from the interval $[0, 10]$ whenever the inner loop is entered.

The upper bound 10, though, is missed both by the vanilla version of **SLR** as well as of **SLR** enhanced with localized placement of $\boxminus$. The reason is that the inner loop is iterated with the interval $[1, \infty]$ for i until stabilization before, triggered by a narrowing iteration of the outer loop, the value $[1, 10]$ for i arrives at the entry point of the inner loop. Since $[1, 10] \sqcup [1, \infty] = [1, \infty]$, the finite upper bound of i at the entry point cannot be recovered. ■

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

In the following, we indicate how this strategy may be integrated into the generic solver $\mathbf{SLR}^3_\square$ (see Figure 4.10). The resulting algorithm $\mathbf{SLR}^4_\square$ requires a function restart. This function when called with a priority r and a variable x, recursively traverses the $\mathsf{infl}[x]$ and sets it to the empty set. Each found variable y is added to the priority queue Q and removed from the set stable. Moreover, if the priority of y is less than r, then the value $\rho[y]$ is reset to $\bot$ and restarting recursively proceeds with r and the variables from the set $\mathsf{infl}[y]$. The function restart then is called within the function solve for a variable x whenever x is currently contained in wpoint and the new value tmp for x is less than the current value for x. In this case, all variables in the set $\mathsf{infl}[x]$ are restarted (w.r.t. the priority of x). Otherwise, the algorithm behaves like the algorithm $\mathbf{SLR}^3_\square$.

Example 17. Consider again the program from Figure 4.9. As soon as narrowing the head of the outer loop recovers the interval $[0, 9]$ for the program variable i, recursively, the values for the reachable program points with lower priorities are reset to $\bot$. This refers to all program points in the body of the outer loop and thus also to the complete inner loop. Reevaluation of all these program points with the value $[0, 9]$ for i at the outer loop head provides us with the invariant $1 \leq i \leq 10$ throughout the inner loop. ■

The algorithm $\mathbf{SLR}^4_\square$ will return a $\boxminus$-solution whenever it terminates. A guarantee, however, of termination is no longer possible even if right-hand sides are monotonic and only finitely many variables are visited. Intuitively, the reason is the following. Assume that the value for a variable x has decreased. Then we might expect that restarting the iteration

```
let rec restart r $y$ =
    add $Q$ $y$;
    stable $\leftarrow$ stable $\setminus$ $\{y\}$;
    if key[$y$] < r then
      $\rho[y]$ $\leftarrow$ $\bot$;
      M $\leftarrow$ infl[$y$];
      infl[$y$] $\leftarrow$ $\emptyset$;
      foreach $z \in$ M do restart r $z$
in
let rec solve $x$ =
  ...
  if tmp $\neq \rho[x]$ then
    if wpx $\wedge$ tmp $\sqsubseteq \rho[x]$ then
      foreach $z \in$ infl[$x$] $\cup \{x\}$ do restart key[$x$] $z$;
    else
      W $\leftarrow$ if wpx then infl[$x$] $\cup \{x\}$ else infl[$x$];
      foreach $y \in$ W do add $Q$ $y$;
      stable $\leftarrow$ stable $\setminus$ W;
    infl[$x$] $\leftarrow$ $\emptyset$;
    $\rho[x]$ $\leftarrow$ tmp;
    while ($Q \neq \emptyset$) $\wedge$ (min_key $Q$ $\leq$ key[$x$]) do
      solve (extract_min $Q$);
```

Figure 4.10: Parts of the solver $\mathbf{SLR}^3_\Box$ with restarting.

for lower priority variables results in a smaller next approximation for x. Due to the non-monotonicity introduced by widening, this need not necessarily be the case. Accordingly, we are no longer able to bound the number of switches between increasing and decreasing phases for x. There are simple practical remedies for nontermination, though. We may, for example, bound for each variable the number of restarts which do not lead to the same value or a decrease.

4.11 Solver for side-effecting equations systems

Example 18. Consider the following program.

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

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

Generic solving, as we have discussed in the preceding sections cannot generally be extended to right-hand sides that not only return a value for the left-hand side x of the equation, but additionally may produce side-effects to other variables. In general, several side effects may occur to the same variable z. Over an arbitrary domain of values, though, it remains unclear how the multiple contributions to z should be combined. Therefore in this section, we assume that the values of variables are taken from a lattice $\mathbb{D}$ and also that right-hand sides are pure.

For side-effecting constraint systems this means that evaluating a right-hand side $\mathcal{F}\,x$ applied to functions $\mathsf{get} : X \to \mathbb{D}$ and $\mathsf{side} : X \to \mathbb{D} \to$ **unit**, consists of a sequence of value lookups for variables by means of calls to the first argument function get and side effects to variables by means of calls to the second argument function side which is terminated by returning a contribution in $\mathbb{D}$ for the corresponding left-hand side. Subsequently, we assume that each right-hand side $\mathcal{F}\,x$ produces no side effect to x itself and also to each variable $z \neq x$ at most one side effect.

Technically, the right-hand side $\mathcal{F}\,x$ of x with side effects can be considered as a succinct representation of a function

$$\overline{\mathcal{F}\,x} \in (V \to \mathbb{D}) \to (V \to \mathbb{D})$$

that takes a mapping ρ and does not just return a single value, but again another mapping ρ' where $\rho'[x]$ equals the return value computed by $\mathcal{F}\,x$ for get $= \rho$, and for $z \neq x$, $\rho'[z] = d$ if during evaluation of $\mathcal{F}\,x$ get side, side is called for z and d. Otherwise, i.e., if no side effect occurs to z, $\rho'[z] = \bot$.

Similarly to constraint systems, a post solution of an equation system with side-effects $\mathcal{F}$ is a mapping $\rho : V \to \mathbb{D}$ such that for every $x \in V$, $\rho \sqsupseteq \overline{\mathcal{F}\,x}\,\rho$. A *partial* post solution with domain dom $\subseteq V$ is a mapping ρ : dom $\to \mathbb{D}$ such that for every $x \in$ dom, evaluation of $\mathcal{F}\,x$ for ρ accesses only variables in dom and also produces side effects only to variables in dom.

Moreover, for equation systems (5) and (6), it can be shown that $\overline{\rho} \sqsupseteq \overline{\mathcal{F}\,x}\,\overline{\rho}$ where $\overline{\rho}$ is the total variable assignment obtained from ρ by setting $\overline{\rho}[y] = \bot$ for all $y \notin$ dom. This demonstrates that partial solutions of equation systems generated for static program analysis can sometimes be extended to "full" solutions.

In the following, we present a side-effecting variant $\mathbf{SLR}^{1+}_{\Box}$ of the algorithm $\mathbf{SLR}^{1}_{\Box}$ from Section 4.7 that for such systems returns a partial $\Box$-solution—whenever it terminates. Moreover, the enhanced solver $\mathbf{SLR}^{1+}_{\Box}$ is guaranteed to terminate whenever all right-hand sides $\mathcal{F}\,x$ are *monotonic*, i.e., the functions $\overline{\mathcal{F}\,x}$ are all monotonic.

Example 19. Consider again the analysis of Example 18. The contributions to the global program variable g by different contexts may well be combined individually by widening to the current value of the global. When it comes to narrowing, though, an individual combination may no longer be sound. Therefore, the extended local solver $\mathbf{SLR}_{\Box}^{1+}$ should collect all occurring contributions into a *set*, and use the *joint value* of all these to possibly improve the value of g. ■

Conceptually, the algorithm $\mathbf{SLR}_{\Box}^{1+}$ therefore creates for each side effect to variable z inside the right-hand side of x, a fresh variable $\langle x, z\rangle$ which receives that single value during evaluation of the right-hand side $\mathcal{F}\,x$. Furthermore, the algorithm maintains for every variable z an auxiliary set $\mathsf{set}[z]$ which consists of all variables x whose right-hand sides may possibly contribute to the value of z by means of side effects. Accordingly, the original system of side-effecting equations is (implicitly) transformed in the following way:

a) Inside a right-hand side f_x, the side effect $\mathsf{side}\, z\, d$ is implicitly replaced with

$$\mathsf{side}\,\langle x, z\rangle\, d$$

while additionally, x is added to the set $\mathsf{set}[z]$.

b) The new right-hand side for a variable x is extended with a least upper bound of all $\langle z, x\rangle$, $z \in \mathsf{set}[x]$.

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

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

We define:

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

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

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

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

Additionally, recursive solving of variables with smaller or equal priorities in solve has to be done even if the return value does not change. This is due to the fact that the side-effects, implemented in side, might have destabilized some variables during the evaluation of the right-hand side $\mathcal{F}\,x$.

Example 20. Consider again interval analysis for the program from Example 18. Concerning the global program variable g, the initialization

$g = 0$ is detected first, resulting in the value $\rho[g] = [0,0]$. Then g is scheduled for reevaluation. This occurs immediately, resulting in no further change. Then the calls $f(1), f(2)$ are analyzed, the side effects of 2 and 3 are recorded and g is rescheduled for evaluation. When that happens, the value $\rho[g]$ is increased to

$$[0,0] \boxminus [0,3] = [0,0] \mathbin{\sqcup\mkern-15mu-} [0,3] = [0,\infty]$$

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

$$[0,\infty] \boxminus [0,3] = [0,\infty] \mathbin{\sqcap\mkern-15mu-} [0,3] = [0,3]$$

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

Analogously to Theorem 7 from the last section, we obtain:

Theorem 9. The following two statements about $\mathbf{SLR}^{1+}_{\Box}$ hold:

a) When applied to any system of pure equations with side effects and interesting variable x_0, the algorithm $\mathbf{SLR}^{1+}_{\Box}$ returns a partial post solution—whenever it terminates.

b) Assume that $\mathbf{SLR}^{1+}_{\Box}$ is applied to a system of pure equations over a complete lattice $\mathbb{D}$ where each right-hand side is monotonic. Then for any initial mapping ρ_0 and interesting variable x_0, $\mathbf{SLR}^{1+}_{\Box}$ is guaranteed to terminate and thus always to return a partial post solution—whenever only finitely many variables are encountered.

■

The proof of Theorem 9 is analogous to the proof of Theorem 7. It is worth-while noting, though, that the argument there breaks down if the assumption on the priorities in side-effects is not met: in that case, any re-evaluation of a high-priority variable x may have another effect onto a low-priority variable y—even if x does not change. No guarantee therefore can be given that the overall sequence of values for y will eventually become stable. If on the other hand, the side-effected variable y has priority greater than x, at re-evaluation time of y, the evaluation of x has already terminated where only the final contributions to y are taken into account. Since only finitely many such contributions are possible, the algorithm is overall guaranteed to terminate.

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

Example 21. Consider the following program:

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

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

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

the widening is immediately compensated by the consecutive narrowing. The same phenomenon occurs at every successive update of the value for g, implying that $\mathbf{SLR}^{1+}_{\Box}$ will not terminate.

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

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

and the fixpoint computation terminates. ■

In practical applications where the side-effected unknowns correspond to globals, the extra condition on priorities in Theorem 9 can be enforced, e.g., by ensuring that the initializers of globals are always analyzed *before* the call to the procedure main.

Theorem 9 only speaks about the extension of the base version of the algorithm $\mathbf{SLR}^1$ to systems of equations with side effects. A similar extension is also possible to the solvers with localized application of $\boxminus$. In order to ensure termination also in this case, however, we additionally must insert every side-effected unknown into the set wpoint of unknowns where the operation $\boxminus$ is to be applied. For the side-effecting version of $\mathbf{SLR}^3_\Box$, we therefore define:

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

With this definition, termination of the algorithm $\mathbf{SLR}^{3+}_\Box$ can be guaranteed under the same assumptions as for the algorithm $\mathbf{SLR}^{1+}_\Box$.

4.12 Experimental evaluation

We have implemented the various generic local solvers and included them into the analyzer Goblint for multi-threaded C programs. Goblint uses CIL as C front-end [Necula et al., 2002] and is written in OCaml.

The tests were performed on a 2.7GHz Intel Core i7 laptop, with 8GB DDR3 RAM, running OS X 10.9.

In a first series of experiments we tried to clarify the increase of precision possibly attained by means of the various $\boxminus$-solvers w.r.t. the standard two-phase solving using widening and narrowing according to [Cousot and Cousot, 1976]. For these experiments, we used the benchmark suite[1] from the Märdalen WCET research group [Gustafsson et al., 2010] which collects a series of interesting small examples for WCET analysis, varying in size from about 40 lines to 4000 lines of code. We have extended this benchmark suite by four tricky programs from [Amato and Scozzari, 2013]: a) `hh.c`, b) `hybrid.c`, c) `nested.c`, and d) `nested2.c`.

On top of standard analyses of pointers, we performed an interval analysis which soundly approximates 32bit integers with wrap-around semantics. For widening, this means that the operator widens the lower and upper bounds first to minint and maxint, respectively, and, if an underflow or overflow cannot be excluded, also the corresponding upper and lower bounds. In order to enable two-phase solving, we performed context-insensitive analysis only.

Within this setting, we determined the precision achieved by the $\boxminus$-solvers compared to the corresponding solver which realizes a distinct widening phase, followed by a distinct narrowing phase. The results of this comparison are displayed in Figures 4.11 to 4.14. Figure 4.11 reports the percentage of program points where solver $\mathbf{SLR}^{1+}_{\boxminus}$ returns better results than two-phase solving. In the vast majority of cases, $\mathbf{SLR}^{1+}_{\boxminus}$ returned significantly better results—supporting the claim that $\boxminus$-solving may improve the precision.

Figure 4.12 reports the percentage of program points where an improvement over $\mathbf{SLR}^{1+}_{\boxminus}$ can be achieved if the operator $\boxminus$ only is applied at widening points, as implemented by solver $\mathbf{SLR}^{2+}_{\boxminus}$. Here, our experiments show that, at least for the given simple form of interval analysis, an improvement can only be observed for very few examples. The reason might be that, applying narrowing, intertwined with widening can quite often recover some of the precision lost by the superfluous widenings.

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

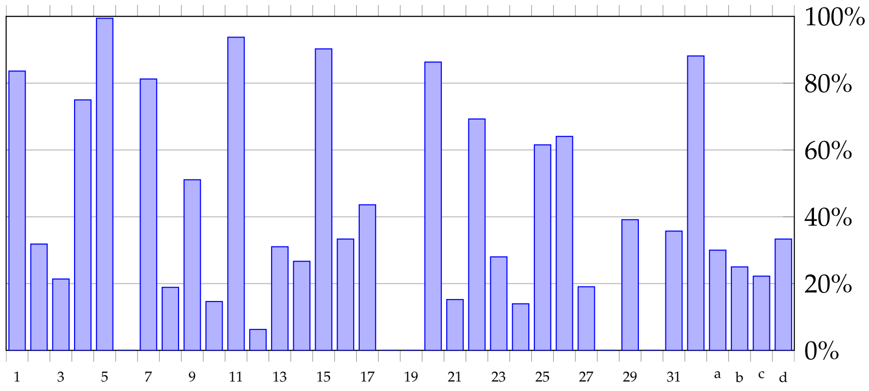


Figure 4.11: The relative improvement of $\mathbf{SLR}_{\boxminus}^{1+}$ over two-phase solving.

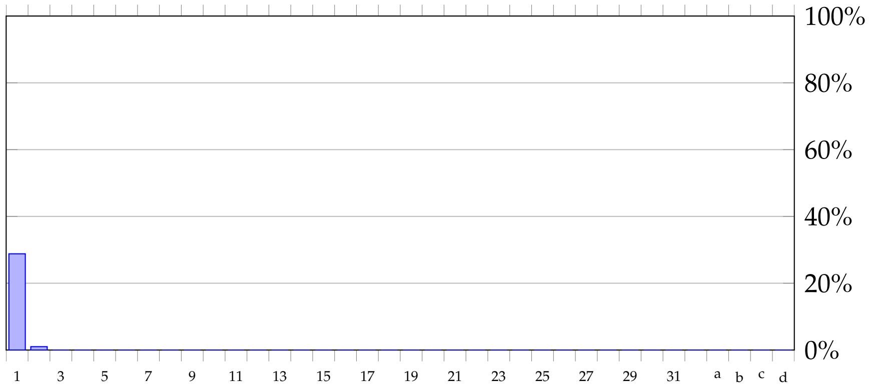


Figure 4.12: The relative improvement of $\mathbf{SLR}_{\boxminus}^{2+}$ over $\mathbf{SLR}_{\boxminus}^{1+}$.

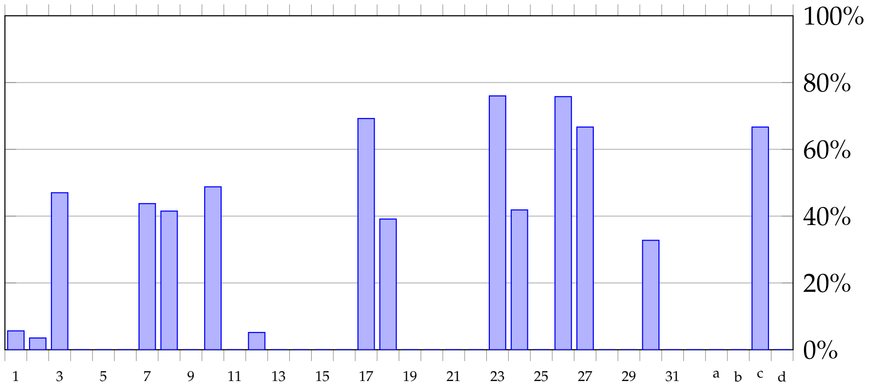


Figure 4.13: The relative improvement of $\mathbf{SLR}^{3+}_{\boxminus}$ over $\mathbf{SLR}^{2+}_{\boxminus}$.

Figure 4.13 then reports the relative further improvement when additionally widening points can dynamically be removed during solving. In 15 of 37 cases, we again obtain an improvement, in some cases even for over 70% of program points! This strategy therefore seems highly recommendable to achieve good precision.

Figure 4.14 finally explores the impact of restarting. Here, the picture is not so clear. For the second benchmark, restarting resulted even in a loss of precision for a small fraction of program points, while still for a larger fraction improvements were obtained. In two further benchmarks, program points with incomparable results where found. For benchmark program 3, these make up about 4% of the program points, while for program 7, the fraction goes even up to 31%. In principle such a behavior is not surprising, considering the non-monotonicity of widening. Still, for two more example programs, drastic improvements are found. One of these comes from the WCET benchmark suite, while the other has been provided in [Amato and Scozzari, 2013], admittedly, as an example where restarting is beneficial.

In a second experiment, we explored the relative efficiencies of our implementation of the generic local $\boxminus$-solvers. For this type of experiments, we selected all benchmarks from the SpecCpu2006 benchmark suite, which can be handled by the C front-end CIL that is used in our analyzer. The

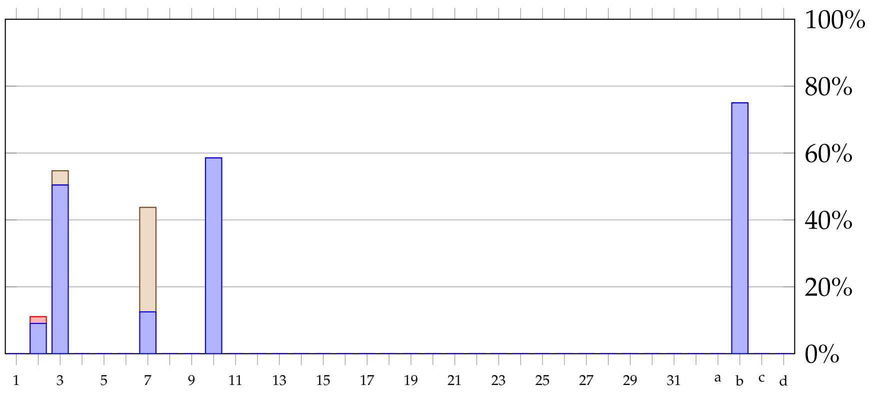


Figure 4.14: Comparison of $\mathbf{SLR}^{4+}_{\boxminus}$ with $\mathbf{SLR}^{3+}_{\boxminus}$ indicating the percentage of program points where the results are incomparable (brown), better (blue) or worse (red).

set of selected benchmarks consist of seven programs in the range of 1 to 33 kloc, `400.perlbench` with 175 kloc, and `445.gobmk` with 412 kloc of C code.

For the chosen programs, we performed interval analysis where local variables are analyzed depending on a calling context which includes all non-interval values of locals, while the values of globals are analyzed flow-insensitively. Such kind of analysis cannot be performed by the two-phase approach, since right-hand sides are not monotonic and the sets of contexts and thus also the sets of unknowns encountered during the widening and narrowing phases may vary. The numbers of evaluations of right-hand sides for the side-effecting solvers $\mathbf{SLR}^{1+}_{\boxminus}$ to $\mathbf{SLR}^{4+}_{\boxminus}$ are reported in Figure 4.15. For a comparison we also included the numbers of evaluations if the solver $\mathbf{SLR}^{1+}_{\square}$ uses plain widening only.

The analysis of the seven smallest programs could be handled in less than 13 seconds. The program `400.perlbench` could also be handled by our solvers but with running times between 18 minutes (using $\mathbf{SLR}^{3+}_{\boxminus}$) and 4 hours (using $\mathbf{SLR}^{4+}_{\boxminus}$). For the largest benchmark `445.gobmk` with 412 kloc, however, interprocedural interval analysis by means of our solvers did not terminate within 5 hours.

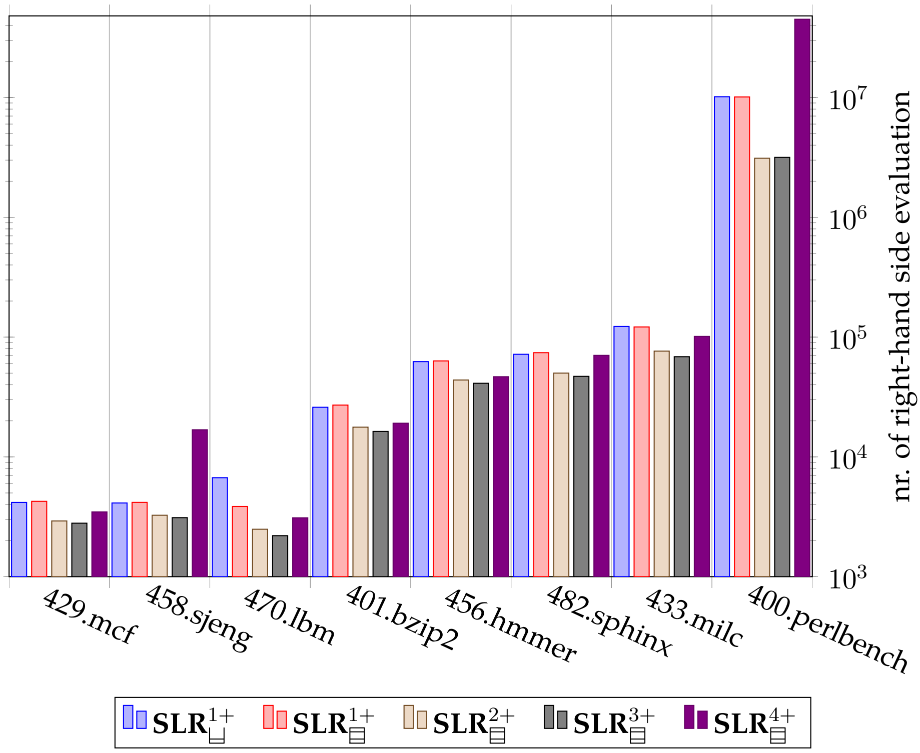


Figure 4.15: Interval analysis of SpecCpu2006 programs.

The first observation is that $\mathbf{SLR}^{1+}_{\square}$ is only marginally slowed down if widening is enhanced to $\boxminus$, i.e., narrowing is added. The second observation is that the efficiency of fixpoint computation is greatly improved when restricting the application of $\boxminus$ to widening points. Improvements of about 30% could consistently be obtained. Enhancing solver $\mathbf{SLR}^{2+}_{\boxminus}$ to solver $\mathbf{SLR}^{3+}_{\boxminus}$, on the other hand, comes with a significant improvement in precision, and additionally results in another slight reduction of the number of evaluated right-hand sides. To us, these numbers came at a surprise, since even in those scenarios where we could theoretically establish termination of the algorithms, still exponentially worse behavior than for widening alone, could not be excluded.

Restarting, finally, adds another dimension of potential inefficiency to fixpoint iteration. Yet, our numbers for $\mathbf{SLR}^{4+}_{\square}$ on the benchmark suite show that the practical slowdown over the fastest solver $\mathbf{SLR}^{3+}_{\Box}$ is in many cases still better than solving with $\mathbf{SLR}^{1+}$ with widening alone. For the programs `458.sjeng` and `400.perlbench`, however, $\mathbf{SLR}^{4+}_{\Box}$ is slower by a factor of 5 and 14, respectively.

In summary, the solver $\mathbf{SLR}^{3+}_{\boxminus}$ turns out to be a robust algorithm with decent run-times. Interestingly, the increase in precision over plain widening as well as over $\boxminus$-solving by means of $\mathbf{SLR}^{1+}_{\boxminus}$ is combined with a significant speedup. The new solver allows to significantly improve precision over the two-phase widening/narrowing approach and also is successfully applicable in more general analysis scenarios, where the two-phase approach was not applicable.

4.13 Conclusion

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

It has not been clear before-hand, though, how well the resulting algorithms behave for real-world program analyses. In order to explore this question, we have provided an implementation within the analysis framework Goblint. In our experimental set-up, we considered interprocedural interval analysis. Our experiments confirm that fixpoint iteration based on the combined operator $\boxminus$ may increase precision considerably, and that it behaves well for interprocedural analysis where right-hand sides of equations can no longer be considered as monotonic. We demonstrated that the add-on of localizing $\boxminus$ operators increases precision further, while efficiency is improved at the same time. An equally clear picture could not be identified for the extra optimization of restarting. While we found clear improvements in selected cases and generally still an acceptable efficiency, we also found exceptional cases where a (minor) loss of precision occurs at some program points or where analysis time is considerably increased.

Our experiments were performed for standard interval analysis with the obvious widening and narrowing operators. It remains for future work to explore how well our methods work also for other domains and for more sophisticated widening and narrowing operators.

5 Implementation in Goblint

The methods described in the previous two chapters were used to improve the Goblint analyzer that is developed at Technische Universität München (TUM) and University of Tartu (UT). Goblint is a constraint system based static analysis framework with a focus on analyzing locking behavior of multi-threaded C programs. It is written in OCaml and it makes use of the CIL [Necula et al., 2002] framework as the front-end.

A high-level overview of Goblint is given in Figure 5.1. First, the control flow graph is constructed from the standard textual representation. From the control flow graph and a given analysis specification, the side-effecting constraint system is then generated. Next, the constraint system is solved by the selected generic solver. All of the presented high level components are implemented in Goblint as modules or functors in OCaml—more detail is given in Section 5.2.

The rest of the chapter will be used to look at some of the improvements and more interesting implementation details: the configuration subsystem, architecture of the analyzer, handling of unreachable code for analyses, an optimization to save memory, the query subsystem, and composition of analysis specifications, implementation of a precision improvement coming from conditional locking, and dynamic solver selection implemented as a static OCaml module.

5.1 Configuration

Most of the analyses implemented in Goblint are parametrized in some way. Different analysis parameters are used, for example, to enable experimental features in the hope to increase the precision of the analysis,

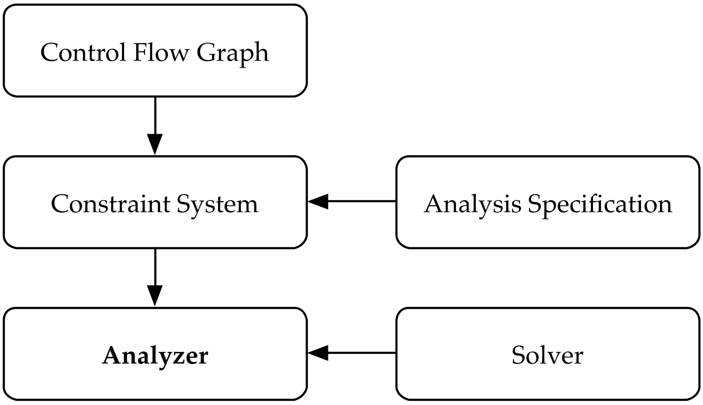


Figure 5.1: General structure of constraint system based analysis tool.

or to decrease the precision in the hope to increase the speed of the analysis. In some cases an analysis requires extra information about the code to be analyzed, e.g., is it a Linux device driver, an OSEK program, or a program using the POSIX interface. At time of writing this thesis, the Goblint tool contained 92 different parameters.

To allow flexible configuration of the analyzer, a JSON (JavaScript Object Notation) based sub-framework was developed. The framework simplifies the task of adding new options and querying the value of the options inside the analyzer. All options and values are stored centrally to allow for better overview of the settings. Additionally, the system allows grouping of settings, merging of settings from different sources, and writing the active settings back to a file.

Inside Goblint, all configuration options are stored in a specific JSON object—a mapping of string to JSON values. A JSON value can be NULL, true or false, a number, a string, a mapping of strings to JSON values, or a list of JSON values. An example configuration file can be seen in Figure 5.2.

At start-up, the default configuration is active, however, there are several ways to change the settings—before they are accessed. One way is to merge an external JSON file with the currently active configuration. For that, the name and path of the external file can be passed to Goblint

```
{ "ana"          : { "activated" : [["base", "mutex"]],
                   , "int"       : { "interval" : true
                                   , "trier "   : false
                                   }
                   }
, "dbg"          : { "verbose" : true
                   }
, "printstats "  : true
, "noverify"     : false
, "result "      : "html"
, "solver"       : "slr3 "
}
```

Figure 5.2: Example configuration file.

via the `--conf` command line option. The merging works as follows: all JSON values except the mappings are over-written form the external file, mappings, however, are merged in a way that retains the values for keys that are missing in the external source.

Another way to change the active configuration is by specifying the values using the command line parameter `--set`. This parameter takes two arguments: first the path inside the configuration value to the sub-value to be changed, and second, the new value. Paths inside the configuration are specified using a dot for selecting a field and square brackets for lists—the usual way for JavaScript and other programming languages of the C language family, e.g., one might specify `--set dbg.verbose false`. The value portion of the `--set` parameter is JSON with the exception that single quotes (′) are used as string delimiters instead of double quotes (").

There is a convenience command line parameter `--sets` that behaves similarly to `--set`, but interprets the value as a string. This is convenient as it allows to avoid the otherwise necessary double quoting, e.g., `--sets v test` instead of `--set v " 'test' "` .

Finally, there is a command line option `--writeconf` that prints out the currently active configuration. Note, however, that the command line parameters are parsed from left to right and all other parameters as well as `--writeconf` will be performed during parsing. E.g., after running

`./goblint --writeconf x.json --sets result html` the file `x.json` will contain the default values only—as the analysis result will be switched to output HTML files only after writing to `x.json`.

From the analyzer side, querying configuration is most often done using one of the following functions:

$$\begin{array}{lcl} \mathsf{get\_bool} & : & \mathit{string} \to \mathit{bool} \\ \mathsf{get\_int} & : & \mathit{string} \to \mathit{int} \\ \mathsf{get\_string} & : & \mathit{string} \to \mathit{string} \end{array}$$

In each case, the first argument to these functions is the path—similarly to the command line arguments. The resulting value is either the value in the configuration or an OCaml exception in the case that the value is not of the correct type. Additionally, more general functions exist to cover other, more involved, use cases, e.g., lists.

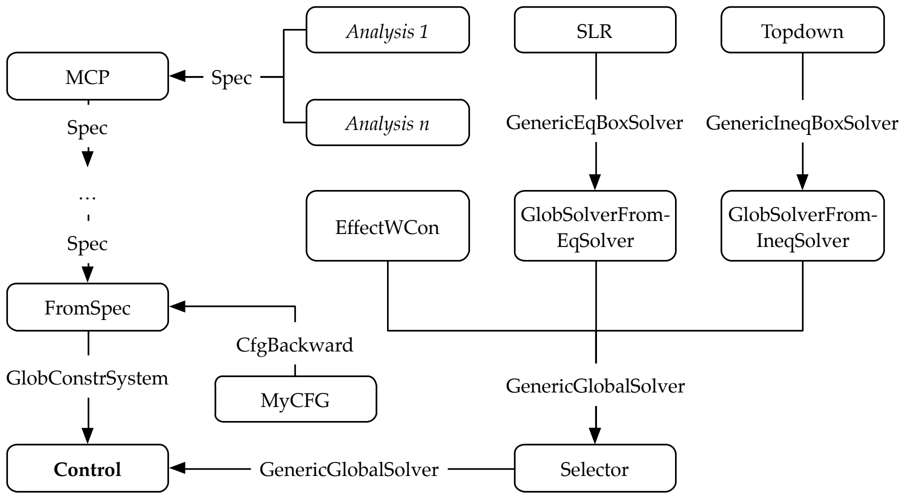


Figure 5.3: Overview of important module signatures.

5.2 Structure of Goblint

To get a more detailed overview of the analyzer, we next look at how the different modules are abstracted, i.e., which module signatures are used, and how different important modules can be combined. A simplified overview of the most important module signatures is presented in the Figure 5.3 where the nodes are module names and edges between modules represents use of the module. Labels on graph edges specify module types that the using module relies on, e.g., the Selector can use any solver module that has the type GenericGlobalSolver. Edges that join together allow for a choice, e.g., Selector can either use the EffectWCon solver directly, or SLR through GlobSolverFromEqSolver, depending on the active configuration.

The analysis combining mechanism MCP is able to use any analysis specification that conforms to the module type Spec. Analysis specifications can be applied to FromSpec to generate constraint systems of type GlobConstrSystem, given a control flow graph of type CfgBackward, which is provided by the MyCFG module.

There are three different module types for solvers: □-solvers for equation systems GenericEqBoxSolver, □-solvers for constraint systems GenericIneqBoxSolver, and solvers with separate local and global variables GenericGlobalSolver. SLR implements an equation system solver, Topdown implements a constraint system solver, and EffectWCon is an implementation of **RLD** that separates local and global variables.

The module Control, then, takes the constraint system that was made with FromSpec and solves it with the Selector solver.

One of the most central pieces of abstraction is the abstraction of the lattice structure. In Goblint there exists a module signature Lattice and its subtype Printable (Figure 5.4). Printable means that the type t can be used inside compare-based tree data structures as well as inside hashtables. Additionally Printable requires functions to convert the value into short textual form, into pretty-printable textual form, or to XML form.

The Lattice module type builds on top of Printable by requiring lattice functions for the greatest and least elements, a function that defines the

```
module type Printable =
sig
  type t
  val short    : int  → t → string
  val pretty   : unit → t → doc
  val printXml : α output → t → unit

  val compare  : t → t → int
  val equal    : t → t → bool
  val hash     : t → int
  ...
end
```

```
module type Lattice =
sig
  include Printable
  val top    : unit → t
  val bot    : unit → t
  val leq    : t → t → bool
  val join   : t → t → t
  val meet   : t → t → t
  val widen  : t → t → t
  val narrow : t → t → t
  ...
end
```

Figure 5.4: The Printable and Lattice module signatures.

ordering relation, a binary least upper bound operator as well as a binary greatest lower bound operator. In addition, implementations of the widening and narrowing operators are required.

Specifications for analyses conform to the module signature of Spec (Figure 5.5). Each specification must define a lattice D, a context C, and functions context and val_of to move between them. Context values are used as a part of equation system variables, as described in Section 3.8, whereas elements of D are used as values representing abstract states.

Another lattice G must be provided for analysis specifications, that will represent the abstract values of global/flow-insensitive variables. Global program variables will be used as the flow-insensitive constraint system variables, and their values (from type G.t) will come exclusively from side-effects.

Each analysis needs to have a name, a starting value for the state at the beginning of the main function, and a query function. The name of the analysis will be used in the configuration system to refer to the analysis.

The querying subsystem allows for communication between analyses—each analysis must provide a function query that must be able to answer questions according to the information available to the analysis at hand. This is possible, as the resulting type of query is a complete lattice where

```
module type Spec =
sig
  module D : Lattice
  module C : Printable
  module G : Lattice

  val name  : string

  val context  : D.t → C.t
  val val_of   : C.t → D.t

  val startvalue : unit → D.t
  val query      : (D.t, G.t) ctx → Query.t → Answer.t

  val assign     : (D.t, G.t) ctx → lval → exp → D.t
  val branch     : (D.t, G.t) ctx → exp → bool → D.t
  val body       : (D.t, G.t) ctx → fundec → D.t
  val return     : (D.t, G.t) ctx → exp option → fundec → D.t
  val special    : (D.t, G.t) ctx → lval option → varinfo → exp list → D.t
  val enter      : (D.t, G.t) ctx → lval option → varinfo → exp list → (D.t * D.t) list
  val combine    : (D.t, G.t) ctx → lval option → exp → varinfo → exp list → D.t → D.t
  ...
end
```

```
type (α,β) ctx =
  { local  : α
  , global : varinfo → β
  , ask    : Query.t → Answer.t
  , spawn  : varinfo → α → unit
  , split  : α → exp → bool → unit
  , sideg  : varinfo → β → unit
  }
```

Figure 5.5: Analysis specification signature, including type for ctx.

the greatest value ⊤ represents the fact that the answer is unknown to the analysis. There is a risk of non-termination arising from circularly defined queries—special care should be taken to avoid such situations.

Finally, each analysis needs to define seven specific transfer functions. Transfer functions need to be provided for assignment statements, guard statements, entering into, and returning from functions. The special transfer function is applied for thread spawning and other API functions whose code is not provided. For function calls, where the code is provided, enter and combine, as introduced in Section 3.6, will be used.

The types lval, exp, fundec, and varinfo come from the CIL framework and represent left-values, pure expressions, function definitions, and variable names, respectively. The type doc also comes from CIL and represents pretty-printed documents as introduced by Wadler [2003]. Italic is used to emphasize standard OCaml types.

Note, that the first argument of each transfer function is an OCaml record of the type (D.t,G.t) ctx. That argument contains several pieces of information. Namely, the local and the global state, side-effecting function for globals, function spawn for spawning threads, function split for splitting paths, and a function ask that answers queries.

Using the function ask, other analyses can possibly be queried—in the case that other analyses have been configured to run besides the analysis at hand. Note, however, that as the resulting type of ask is a lattice, one must always be prepared to act in a safe manner if the answer is unknown, i.e., the result is $\top$.

The function spawn is a specialized side-effecting function that is able to send side-effects to appropriate starting points of spawned functions. In the case that multiple analyses are combined together, spawn must first gather all starting values for each spawned function.

A function call split a e b, however, is used to return multiple values form a single function call. Each such value a that is returned using split is considered to have passed a positive or negative guard (depending on b) of the expression e. Such behavior is used to correlate the multiple return values coming from multiple analyses.

The directory structure of Goblint is depicted in Figure 5.6; also showing dependencies of important modules. Generic lattice modules, such as lattices for maps and sets, can be found in the "domains" subdirectory, whereas C language specific and analysis specific lattices are in "cdomains". All analyses specifications are in the directory "analyses", including the analysis combinator MCP that applies all the analyses that are enabled using the configuration subsystem. All different solver implementations and solver transformation combinators reside in the "solvers" directory. Control flow graph generation, modules dealing with constraint system generation and transformation live in the "framework" directory. The module that combines all necessary modules into an analyzer—Control—also resides in "framework".

The Figure 5.7 shows module signatures for control flow graphs and equation and inequation systems. Notice that here we only need to access CFG edges in the reverse direction of control flow. Also, the figure

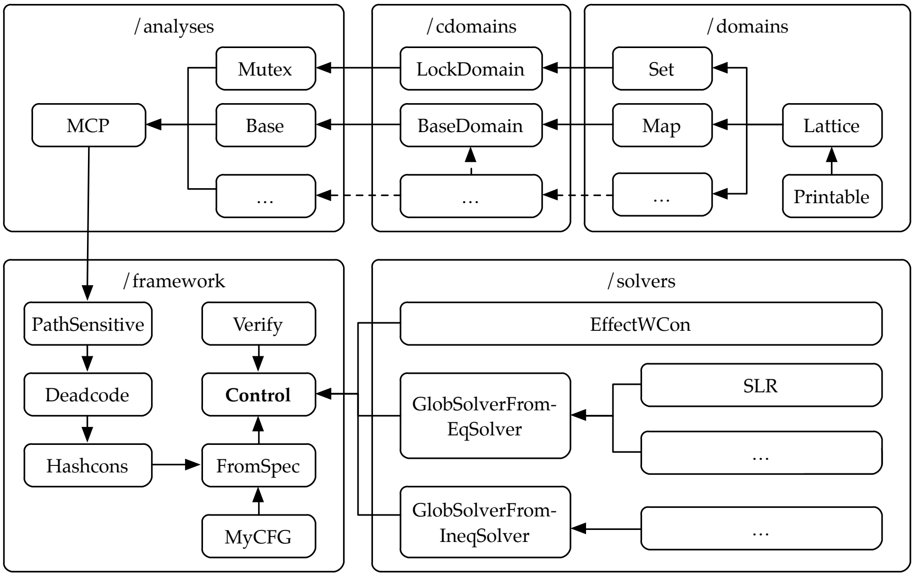


Figure 5.6: The directory structure in relation to important modules.

shows that equation and inequation systems only differ in the use of a data type—a list for inequation systems and option for equation systems.

One simple way of transforming an inequation system into an equation system, as discussed in Section 4.2, can be realized by the functor EqConstrSysOf—shown in Figure 5.8. Moreover, this method of transformation retains the types of values and of the variables.

Definitions of local equation- and inequation $\Box$-solvers are displayed in Figure 5.9. Each solver functor takes, first, the system argument, and, second, the definition of a hash-map as arguments, and returns a module that implements a solve function. The first argument to the solve function is the $\Box$-operator itself. Additionally, as the second argument of solve, starting values can be provided—represented as an association list. The final argument to solve is the list of interesting variables. The result is a hash-map that stores a partial solution (candidate) of the system.

Another, more specialized, definition of constraint system is used in Goblint—namely, GlobConstrSystem (Figure 5.10). This definition makes a clear separation between flow-sensitive constraint system variables LVar

```
module type CfgBackward =
sig
  val prev : node → (edge * node) list
end
module type MonSystem =
sig
  type α m
  module V : Printable
  module D : Lattice
  val system : V.t → ((V.t → D.t) → (V.t → D.t → unit) → D.t) m
end
module type EqConstrSys = MonSystem with type α m ← α option
module type IneqConstrSys = MonSystem with type α m ← α list
```

Figure 5.7: the CFG and constraint systems.

```
module EqConstrSysOf (S:IneqConstrSys) : EqSystem =
struct
  module V = S.Var
  module D = S.Dom
  let system x =
    match S.system x with
      | [] → None
      | f :: fs →
        let combine g h get set = D.join (g get set) (h get set) in
          Some (fold_left combine f fs)
end
```

Figure 5.8: One way of transforming inequation system into equation systems

and flow-insensitive variables GVar. Each kind of variable uses its own separate lattice representation: D, or G, respectively. The function system, representing the constraints, uses separate get and set functions for LVars and GVars.

The Figure 5.11 shows, first, the module signature of a local solver for constraint systems from type GlobConstrSys. The functor signature for the solver is named GenericGlobSolver. Here, separate hash-map definitions can be given for flow-sensitive and flow-insensitive constraint variables. For historical reasons, the □-operator cannot be directly given to

```
module type GenericEqBoxSolver (S : EqConstrSys)
                                 (H : Hash.H with type key=S.V.t) =
sig
  let solve : (S.V.t → S.D.t → S.D.t → S.D.t)
              → (S.V.t * S.D.t) list → S.V.t list → S.D.t H.t
end

module type GenericIneqBoxSolver (S : IneqConstrSys)
                                   (H : Hash.H with type key=S.V.t) =
sig
  let solve : (S.V.t → S.D.t → S.D.t → S.D.t)
              → (S.V.t * S.D.t) list → S.V.t list → S.D.t H.t
end
```

Figure 5.9: Signatures for □-solvers.

```
module type GlobConstrSystem =
sig
  module LVar : Printable
  module GVar : Printable
  module D : Lattice
  module G : Lattice
  let system : LVar.t → ((LVar.t → D.t) → (LVar.t → D.t → unit) →
                         (GVar.t → G.t) → (GVar.t → G.t → unit) → D.t) list
end
```

Figure 5.10: Specialized inequation system with separate lattice for global values.

the solver. However, starting values for both LVar and GVar variables can be given, as well as the set of interesting flow-sensitive variables. For obvious reasons, the set of interesting variables cannot contain flow-insensitive solver variables—this would require to solve the constraint system in its entirety. In addition, transformations into GenericGlobSolver exist from both: GenericEqBoxSolver, and also from GenericIneqBoxSolver.

As shown in Figure 5.12, a given module specification S:Spec and control flow graph Cfg:CfgBackward can be combined into a GlobConstrSys using the functor FromSpec. The functor FromSpec does this by specifying the constraint system types as follows:

```
module type GenericGlobSolver (S : GlobConstrSys)
                              (LH : Hash.H with type key=S.LVar.t)
                              (GH : Hash.H with type key=S.GVar.t) =
sig
  val solve : (S.LVar.t*S.D.t) list → (S.GVar.t*S.G.t) list
              → S.LVar.t list → S.D.t LH.t * S.G.t GH.t
end

module GlobSolverFromEqSolver (S:GenericEqBoxSolver) : GenericGlobSolver

module GlobSolverFromIneqSolver (S:GenericIneqBoxSolver) : GenericEqBoxSolver
```

Figure 5.11: Signatures for solvers with separate global values and transformations.

```
module FromSpec (S:Spec) (Cfg:CfgBackward) : GlobConstrSys

module Verify
    (S:GlobConstrSys)
    (LH:Hash.H with type key=S.LVar.t)
    (GH:Hash.H with type key=S.GVar.t) :
sig
  val verify : D.t LH.t → G.t GH.t → unit
end
```

Figure 5.12: Generator of constraint systems and verification of a result.

```
type LVar.t = node * S.C.t
type GVar.t = varinfo
type D.t = S.D.t
type G.t = S.G.t
```

where node is the type representing the set N of all control flow graph nodes, and varinfo, as already mentioned, the type of (global) program variables.

An interesting difference between constraint systems from Chapter 3 and FromSpec can be seen in the structure of flow-sensitive constraint system variables. In the theoretical portion of the thesis, partial contexts were realized using tuples as lattice elements. However, practically, this is inconvenient as changes to the structure of the lattice would have to be made. Moreover, a fine-grained and configurable solution is preferred. As the proposed result, not only projecting out the second component of the tuple, but *any* projection is allowed.

In other aspects, the constraint system in FromSpec is defined similarly to the constraint systems in Chapter 3—by pattern matching the edge labels and applying appropriate functions from the specification S.

One advantage of the constraint system based approach is the fact that it is straightforward to verify that a partial solution candidate is *really* a partial solution. Simply, the solution condition is checked for each constraint. In Goblint, this is done by the module Verify (signature in Figure 5.12).

5.3 Dead-code lifting

The fact that a program point is dead is in many cases detected by an analysis that resides deep inside the hierarchy of analyses and transformations. This information needs to be propagated to the appropriate level so that the whole analysis can benefit from the information. For that purpose, Goblint contains an analysis transformation DeadCodeLifter, that extends its argument analysis with a special value for unreachable code. To lessen the overhead of messaging DeadCodeLifter, this propagation is done using an OCaml exception.

The described transformation allows to relegate handling of unreachable code to one single analysis specification module and other analyses can then assume that program points are live. Additionally, if an analysis detects unreachable code, simply raising an exception is enough to notify DeadCodeLifter of that fact.

In the default configuration of Goblint, the transformation DeadCodeLifter will be done last—directly before generating the constraint system with FromSpec.

```
type α hobj = private {
  obj   : α;
  tag   : int;
  hcode : int;
}
```

Figure 5.13: The hobj type from the Batteries library.

5.4 Hash consing

Hash consing is a technique used in functional programming languages for reusing objects in memory with the hope to minimize memory usage. It is particularly useful if, during execution, the program generates a large amount of values of the same type.

In OCaml, hash consing consists of wrapping the values of some type in values of hobj type using the hashcons function. The function will check internally if a the same value has already been seen by the function—using a hash-table. In the case that the value has *not* been seen, the value will be wrapped inside a hobj and inserted into the hash-table, after that the wrapped value is, additionally, returned. Otherwise, the equivalent value from the hash-table will be returned.

Note that hashcons can be seen as an identity function. However, if only the value returned by hashcons will be used, and not the input value, then the memory allocated for the copy can be re-claimed by the garbage collector of OCaml.

An additional possible advantage, for using hash consing is that the hash of a value is stored with the object and does not have to be recomputed. This detail minimizes the overhead of applying hash consing for use cases where the values are stored in hash-maps anyway—like in the solver implementation in Goblint. Also, using the hash consing, it is possible to optimize equality checking—namely, using a unique integer identification number tag that is generated for all hash-consed values.

In Goblint, hash consing is realized as an analysis specification transformation and it is applied next to last—directly before DeadCodeLifter is applied. Hash consing can be detrimental to efficiency in some cases, e.g., for the analysis of small programs. For those cases it can be disabled by setting the configuration variable "ana.hashcons" to *false*.

5.5 The query system

In the following section, a simplified formalization of the querying system is presented. Details about the implementation are given in the Section 5.6.

Given a n-tuple lattice $\mathbb{D} = \mathbb{D}_1 \times \ldots \times \mathbb{D}_n$, where each component $\mathbb{D}_i$ is the analysis domain for the i-th analysis for some fixed ordering of the performed analyses. Given a set of queries Q, a complete lattice of answers $\mathbb{A}$, and additionally, a query function for each analysis

$$q_i \in \mathbb{D}_i \to (Q \to \mathbb{A}) \to Q \to \mathbb{A} \; .$$

The answer lattice, similarly to analysis domains, is interpreted using a description relation. In this case, however, the description relation correlates concrete states with tuples consisting of a query and its answer: $\Delta \subseteq S \times (Q \times \mathbb{A})$. In addition, though, the lattice must ensure that the greatest lower bounds of x and y, of $\mathbb{A}$, that both describe a concrete state s for a query $k \in Q$, also describe s for the same query, i.e,

$$s \; \Delta \, (k, x) \; \wedge \; s \; \Delta \, (k, y) \implies s \; \Delta \, (k, x \sqcap y) \; .$$

The combined analysis that uses the domain $\mathbb{D}$ can implement its query function as the following *recursive* function

$$q_{\mathsf{C}} \; (d_1, \ldots, d_n) \; x = \bigsqcap \{ q_i \; d_i \; (q_{\mathsf{C}} \; (d_1, \ldots, d_n)) \; x \mid 1 \leq i \leq n \} \; .$$

Note that the combined query function q_{C} with partially applied current local state $(d_1, \ldots, d_n)$ is made available to all transfer functions *and* query functions of the child-analyses. This allows a form of communication between analyses, possibly using several levels of sub-queries. Care must be taken, however, to avoid non-termination arising from cyclic queries.

5.6 Combining analysis specifications

Registering analyses

The Goblint framework allows combining analyses specifications in a way that allows each analysis to benefit from other analyses. This feature has proven to be very useful, as it allows users of Goblint to quickly add new analyses without changing anything in the framework, other analyses, or even the build system. The combinator itself is implemented as the MCP analysis, conforming to the Spec module specification. First, however, the available analyses must be collected.

Using new OCaml 4.00 features that allow to use modules as first-class values, a function register_analysis is provided (see Figure 5.14). That function then populates a table of combinable analyses analyses_table in a way that assigns each module an integer ID.

OCaml Obj module

The standard library of OCaml contains the module Obj (type in Figure 5.15), that allows to circumvent the type-system of OCaml by casting a value of any type into a special type Obj.t and, more importantly, any value of type Obj.t into a value of any type.

Obviously, these functions cannot be used for arbitrary type conversion. The function repr can be used freely, however, the function obj must only

```
type spec_modules = { spec : (module Spec)
                    ; dom  : (module Lattice.S)
                    ; glob : (module Lattice.S)
                    ; cont : (module Printable.S)
                    }
val register_analysis : (module Spec) → unit
val analyses_table    : (int * spec_modules) list ref
```

Figure 5.14: Registering analyses

be used to convert values back to their original type (or type with the *exact* same structure).

The module Obj is a powerful tool that allows implementation of many interesting features, e.g., printf style output functions and generic serializing/deserializing of any OCaml value. Its usage, however, is discouraged as it is difficult to debug.

The Master Control Program analysis

At the initialization phase, the MCP module asks the configuration subsystem to provide the set of *active* analyses—the analyses that are scheduled to be run. Additionally, for each active analysis, it checks whether that analysis should be run context sensitively or not.

The lattices for flow sensitive and insensitive constraint variables are then constructed in the following manner: lattice elements are lists where each active analysis is represented as one tuple in the list.

```
type MCP.D.t = (int * Obj.t) list
type MCP.G.t = (int * Obj.t) list
type MCP.C.t = (int * Obj.t) list
```

The first component of the tuple is the ID of the analysis and the second is the Obj.t representation of the lattice value. This means that all used lattice elements are lists where the list length equals the number of active analyses.

The context values, however, only contain tuples for analyses that were configured to run context sensitively. This means that the MCP.context

```
module Obj :
sig
  type t
  val obj  : t → α
  val repr : α → t
  ...
end
```

Figure 5.15: The type of the Obj module.

function filters out all tuples that are context sensitive and the MCP.val_of inserts $\top$ values of the "missing" analysis values.

Communication between analyses, as discussed earlier, is realized using a query system. Each active child analysis has provided a query function that answers questions based on the knowledge of the analysis alone, or in combination with a recursive query.

Examples of currently implemented queries are: thread uniqueness, thread priority, pointer reachability, may-points to information, and integer evaluation queries. Evaluation of function pointers, to allow for dynamic procedure calls, is also implemented using a query. Adding new queries is done by simply extending the variant type Query.t with a new constructor. The lattice of answers Answer.t is also a variant type, allowing for integer, boolean, expression set, and left-value set values—among others.

The recursive function MCP.query, as shown previously, then simply computes the greatest lower bound of the answers of the active analyses, and additionally, through the ctx value, provides *itself* for use in recursive queries.

The query function of the MCP analysis is provided to all transfer function calls of active child-analyses. This means that each child-analysis can query the greatest lower bound value of a set of centrally defined questions.

```
void foo(int do_work){
  if (do_work)
    lock(&m);
  . . .
  if (do_work)
    work++;
  . . .
  if (do_work)
    unlock(&m);
}
```

Figure 5.16: A path-sensitivity example.

5.7 Path-sensitivity

The classical example that needs path-sensitivity [Vojdani and Vene, 2009] comes from data-race analysis, and deals with the case where taking a lock and modifying some variable is done conditionally, as in Figure 5.16. Note, that there are only two possible paths through the code in the example (in the case that do_work is not modified in another thread). Either the variable do_work is *true* in both if-statements or *false* in both if-statements.

Given a value analysis domain $\mathbb{V}$ and must-lockset domain $\mathbb{L}$, the naive combination of these analyses using the Cartesian product $\mathbb{D} = \mathbb{L} \times \mathbb{V}$ would lose such conditional information. After each conditional statement in the example, both domains from each branch must be joined, respectively. The least upper bound operation for must-locksets will, therefore, not retain any information about the lock m.

The solution to the example, as presented by Vojdani and Vene [2009], is to store the analysis information as implications from must-lockset to the values that hold for that lockset: $\mathbb{D} = \mathbb{L} \to \mathbb{V}$. Given that the value analysis performs Conditional Constant Propagation [Wegman and Zadeck, 1991], data-race freedom can be shown for this example. Moreover, as the lockset domain $\mathbb{L}$ is finite, the resulting implication domain will contain infinite ascending chains only when $\mathbb{V}$ contains infinite ascending chains.

The implementation of this feature in Goblint is realized as a transformation of the analysis specification—using the functor PathSensitive. By default, it is applied directly to MCP and is given as an input to the hash cons lifter. The functor PathSensitive receives an analysis specification over the lattice $\mathbb{D}$, that additionally contains a function should_join, and returns an analysis specification over a special power-set lattice. The binary least upper bound operator for that lattice is defined as $X \sqcup Y = n(X \cup Y)$ using normalization function

$$n(Z) = \begin{cases} n(\{x \sqcup y\} \cup (Z \setminus \{x, y\})) & \text{if } \exists x, y \in Z, x \neq y \wedge \mathsf{should\_join}(x, y) \\ Z & \text{otherwise .} \end{cases}$$

Using this construction, the analysis that uses the Cartesian product lattice $\mathbb{D} = \mathbb{L} \times \mathbb{V}$ can be transformed to be path-sensitive using the function

$$\mathsf{should\_join}\,(l_1, v_1)\ (l_2, v_2) = \begin{cases} \mathsf{true} & \text{if } l_1 = l_2 \\ \mathsf{false} & \text{otherwise} \end{cases}$$

A similar definition is given for the analysis combinator MCP. First, the configuration subsystem is queried for analyses that should be run path-sensitively. Then, it filters out non-path-sensitive components of both arguments of should_join and then performs an equality check.

Although the functor PathSensitive modifies the analysis domain to represent a set of paths, the context is not changed. The different paths are simply distributed to different contexts, where each context represents only one path, e.g., instead of using a single solver variable for both called paths $[r_f, \{a, b\}]$ we can instead use the union of $[r_f, a]$ and $[r_f, b]$. This is safe as the different paths cannot interfere with each other, but it is also advantageous as it decreases the number of needed contexts and possibly allows more reuse, e.g., $[r_f, a]$ and $[r_f, b]$ might be used separately for other calls.

5.8 Solver selection

To simplify the dynamic selection of solvers, Goblint provides a special solver Selector, that applies the solver specified by the configuration variable "solver". Similarly to analysis specifications, solver implementations that conform to the module type GenericGlobSolver can be registered using the function add_solver.

Registering a solver is implemented as adding the solver module to an association list. At the call to solve, the association list is traversed and one solver, according to configuration, is selected. The solve function of the selected solver is then used.

As dynamic solver selection in this convenient way is only possible by storing modules in a list data structure, it requires new features added in OCaml 4.00.

6 Conclusion

Building static analysis tools takes a lot of time and effort. Novel ideas in static program analysis are, in many cases, based on previous work, and therefore, should theoretically allow to reuse previous implementations. Those previously available tools, however, might not be flexible enough to support the new ideas.

Building a new, fully featured analysis tool for *each* new analysis, on the other hand, is not practical. Even for prototype implementations, a lot of effort is required for reimplementing classical analyses—or other commonly needed structures.

As a partial solution, analyzers for toy-languages could be created instead. This would allow to forego the issues that arise from supporting a non-trivial subset of some widely used language. It is clear, however, that this does not directly help to improve the quality of safety-critical software.

Having a single theoretical program analysis framework that is general enough to support a large variety of analyses would help to concentrate the efforts to write practical, useful implementations. The existence of generic and configurable implementations would encourage incremental research—it would make it easier to implement new analyses and test new analyses against real-world examples.

As one candidate for such a single analysis framework, we presented side-effecting constraint systems that extend the constraint systems that are known in static program analysis. Additionally, side-effecting constraint systems allows, among others, to implement partial context sensitivity—a feature that could be crucial to achieve reasonable efficiency in some situations. We saw that the techniques of widening and narrowing from

Cousot and Cousot can be efficiently used within the (side-effecting) constraint system framework—even in the case where local solving is required.

The side-effecting constraint system representation forms the core of Goblint—making it possible, among other things, to add new analyses in a straightforward manner, and to combine analyses in a way that allows analyses to be combined to make use of the other analyses knowledge. Goblint allows flexible configuration, vis-à-vis context-, flow-, and path-sensitivity.

I hope that the presented theoretical frameworks and the implementation in OCaml will be useful to others and provide interested people a better starting position to develop new analyses for the C language.

Although only forward analyses were considered in this thesis (and in Goblint), some backwards analyses, e.g., liveness of variables, are also classically represented using a constraint system. For such cases, side-effecting could also potentially be beneficial, however, it is still unclear how to combine backward and forward analyses in a generic manner.

Context sensitive analysis, as presented in the thesis, start with a query about the value of a program point in some specific context. One interesting future direction is making the query to the analysis more fine-grained, e.g., one might want to know whether a specific error can occur. One might hope that such specific queries might be easier to compute than the full information.

Another open question is the possibility of combining side-effecting constraint systems based analyses with other, more precise, techniques, e.g., strategy iteration [Gawlitza and Seidl, 2007].

Bibliography

G. Amato and F. Scozzari. Localizing widening and narrowing. In F. Logozzo and M. Fändrich, editors, *Static Analysis*, pages 25–42. Springer, 2013.

L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

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

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

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

C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL'09*, pages 289–300. ACM Press, 2009.

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

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

P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In B. Robinet, editor, *Second International Symposium on Programming, Paris, France*, page 106—130. Dunod, Paris, 1976.

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

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

P. Cousot and R. Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *PLILP*, volume 631 of *LNCS*, pages 269–295. Springer, 1992.

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

P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *ESOP'05*, volume 3444 of *LNCS*, pages 21–30. Springer, 2005.

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

M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI'02*, pages 57–68. ACM Press, 2002.

C. Fecht. *Abstrakte Interpretation logischer Programme: Theorie, Implementierung, Generierung*. PhD thesis, Universität des Saarlandes, 1997.

C. Fecht and H. Seidl. A Faster Solver for General Systems of Equations. *Sci. Comput. Programming*, 35(2):137–161, 1999.

A. Flexeder, B. Mihaila, M. Petter, and H. Seidl. Interprocedural control flow reconstruction. In *APLAS'10*, volume 6461 of *LNCS*, pages 188–203. Springer, 2010.

T. Gawlitza and H. Seidl. Precise fixpoint computation through strategy iteration. In *Programming Languages and Systems*, pages 300–315. Springer, 2007.

D. Gopan and T. Reps. Lookahead widening. In T. Ball and R. Jones, editors, *Computer Aided Verification*, volume 4144 of *LNCS*, pages 452–466. Springer, 2006.

D. Gopan and T. Reps. Guided static analysis. In H. Nielson and G. Filé, editors, *Proc. of the 14th International Static Analysis Symposium (SAS)*, volume 4634 of *LNCS*, pages 349–365. Springer, 2007.

D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Trans. Prog. Lang. Syst.*, 23:685–746, Nov. 2001.

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

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

J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks — past, present and future. In B. Lisper, editor, *WCET2010*, pages 137–147, Brussels, Belgium, July 2010. OCG.

S. Z. Guyer and C. Lin. Client-Driven pointer analysis. In *SAS'03*, volume 2694 of *LNCS*, pages 214–236. Springer, 2003.

N. Halbwachs and J. Henry. When the decreasing sequence fails. In A. Miné and D. Schmidt, editors, *SAS*, volume 7460 of *LNCS*, pages 198–213. Springer, 2012. ISBN 978-3-642-33124-4.

J. Henry, D. Monniaux, and M. Moy. Succinct representations for abstract interpretation. In A. Miné and D. Schmidt, editors, *Static Analysis Symposium (SAS'12)*, volume 7460 of *LNCS*, pages 283–299. Springer Berlin / Heidelberg, 2012.

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

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

B. Jørgensen. Finding fixpoints in finite function spaces using neededness analysis and chaotic iteration. In *SAS'94*, volume 864 of *LNCS*, pages 329–345. Springer, 1994.

G. Kildall. A unified approach to global program optimization. In *POPL'73*, pages 194–206. ACM Press, 1973.

J. Knoop and B. Steffen. The Interprocedural Coincidence Theorem. In *Fourth International Conference on Compiler Construction, CC'92*, pages 125–140, Paderborn, Germany, 1992. Springer-Verlag, LNCS 641.

C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO'04*, pages 75–88. IEEE Press, 2004.

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

N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

O. Lhoták and K.-C. A. Chung. Points-to analysis with efficient strong updates. In *POPL'11*, pages 3–16. ACM Press, 2011.

A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Transactions on Software Engineering and Methodology*, 14:1–41, Jan. 2005.

D. Monniaux and J. L. Guen. Stratified static analysis based on variable dependencies. In *The Third International Workshop on Numerical and Symbolic Abstract Domains*, 2011.

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

K. Poulsen. Software bug contributed to blackout, Feb. 2004. URL `http://www.securityfocus.com/news/8016`.

T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL'95*, pages 49–61. ACM Press, 1995.

N. Rinetzky, J. Bauer, T. W. Reps, S. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL'05*, pages 296–309, 2005.

H. Seidl and V. Vojdani. Region analysis for race detection. In *SAS'09*, volume 5673 of *LNCS*, pages 171–187. Springer, 2009.

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

M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *POPL'97*, pages 1–14. ACM Press, 1997.

M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Application*, pages 189–233. Prentice-Hall, 1981.

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

A. Simon and A. King. Widening polyhedra with landmarks. In N. Kobayashi, editor, *APLAS*, volume 4279 of *LNCS*, pages 166–182. Springer, 2006. ISBN 3-540-48937-1.

B. Steensgaard. Points-to analysis in almost linear time. In *POPL'96*, pages 32–41. ACM Press, 1996.

R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A java bytecode optimization framework. In *CASCON'99*. IBM Press, 1999.

B. Vergauwen, J. Wauman, and J. Lewi. Efficient fixpoint computation. In *SAS'94*, volume 864 of *LNCS*, pages 314–328. Springer, 1994.

V. Vojdani. *Static Data Race Analysis of Heap-Manipulating C Programs*. PhD thesis, University of Tartu, 2010.

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

P. Wadler. A prettier printer. *The Fun of Programming, Cornerstones of Computing*, pages 223–243, 2003.

M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, Apr. 1991.

R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Not.*, 29:31–37, 1994.