Side-Effecting Constraint Systems:
A Swiss Army Knife for Program Analysis

Kalmer Apinis, Helmut Seidl, and Vesal Vojdani

Lehrstuhl für Informatik II, Technische Universität München
Boltzmannstraße 3, D-85748 Garching b. München, Germany
{apinis,seidl,vojdanig}@in.tum.de

Abstract. Side-effecting constraint systems were originally introduced for the analysis of multi-threaded code [22]. In this paper, we show how this formalism provides a unified framework for realizing efficient interprocedural analyses where the amount of context-sensitivity can be tweaked and where the context-sensitive analyses of local properties can be combined with *flow-insensitive* analyses of global properties, e.g., about the heap. Side-effecting constraint systems thus form the ideal basis for building general-purpose infrastructures for static analysis. One such infrastructure is the analyzer generator Goblint, which we used to practically evaluate this approach on real-world examples.

1 Introduction

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 [29].

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.

Thus, when precision is crucial, an integrated approach is preferred. This is the case in sound static analyzers, such as Astrée [6] or Goblint [23]. 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 [13].

We suggest that *side-effecting* constraint systems, introduced in [22] 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 paper is to show that many analysis problems, whose solving seem 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:

1. tabulation of procedure summaries for parts of the calling context only, also in the presence of dynamic procedure calls;
2. integrated analysis which accumulates certain 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. All proofs in this paper have been omitted — but can be found in the technical report [2].

Related Work. The seminal paper by Kildall [15] 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 [4] 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 [7] or directly interpreting the input program abstractly [6]. As one instance, also a framework for analyzing programs with procedures has been provided [5].

Various further approaches to interprocedural analysis are reviewed by Sharir and Pnueli [25] — one based on call-strings while the other, similar to [5], relies on (partially) computing abstract procedure summaries. Following Sharir and Pnueli, later, restricted frameworks for interprocedural analyses have been provided [20] which, however, only work for specific simple domains and therefore may not serve as general program analysis frameworks.

Partial contexts are important for scalability since it enables more compositional approaches. 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 [3, 21]. 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 [18].

Generic frameworks for program analysis and code optimization [16, 19, 27, 30] 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 [12] monitors the performance of an initial flow-insensitive analysis to decide which pointers to track flow-sensitively. Lhoták and Chung [17], 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.

Organization of the paper. Section 2 is meant as a gentle introduction to constraint systems for specifying program analyses. Basic notions are introduced for the case of intra-procedural analyses. Section 3 extends the specification formalism of constraint systems so that it also covers interprocedural analysis in the sense of [5] and indicates how these systems can be extended to make reachability information explicit. Section 4 extends these constraint systems to allow for fine-tuning the amount of context by which procedure calls are distinguished. The resulting constraint systems may be neither monotonic, nor finite. Even worse, some variables of the constraint system may formally depend on infinitely many other variables. Section 5 provides further extensions to the constraint system which supports a flow-insensitive treatment of global information. Section 6 indicates how side-effects, added to the specification formalism of constraints, allow for smooth reformulations of the constraint systems from Sections 4 and 5. The new constraint systems, however, have the advantage that for every abstract lattice, the variables only depend on *finitely* many other variables — thus allowing

to apply generic local fixpoint algorithms as universal solver engines. Sections 7 and 8 provide further evidence of the usefulness of the framework by indicating how dynamic procedure calls as well as the alternative approach to interprocedural analysis of Sharir/Pnueli [25] can also be specified. Section 9 demonstrates that this approach is not only expressive but also results in competitive realizations of analyzers.

2 Intra-Procedural Constraint Systems

We consider programs which consist of a finite set Proc of procedures. Each procedure g is given by a distinct control flow graph (N_g, E_g), where N_g is the set of program points of g and $E_g \subseteq N_g \times L \times N_g$ the set of edges with labels from a set L. An edge label $s \in L$ represents either elementary statements or conditional guards of the source language. Additionally, we have call edges with labels $f()$. The call edge syntax does not allow explicit arguments to functions; passing of arguments or returning results may be simulated, e.g., by means of global variables. Each procedure g has one start node s_g and one return node r_g, and we ensure that every program point $v \in N_g$, even when semantically unreachable, can be formally (i.e., ignoring the semantics of edge labels) reached from s_g, and likewise, r_g can be formally reached from v.

The goal of the analysis of such a program is to infer program invariants. Following the approach of abstract interpretation, program invariants are represented by elements from a complete lattice $(\mathbb{D}, \sqsubseteq)$, where $\mathbb{D}$ is the set of program invariants, and $\sqsubseteq$ the implication ordering between invariants. Let us for the moment consider a program with just one procedure main and without procedure calls. Analyzing such programs is referred to as *intra-procedural* analysis. Assume that we are interested in inferring one invariant for each program point of the analyzed program. Such an analysis is referred to as *flow-sensitive*. Flow-sensitive intra-procedural invariants can conveniently be expressed as solutions of a *constraint system*.

Let V denote a set of constraint variables or unknowns. For intra-procedural analysis, the set of unknowns are simply program points $V = N_{\mathsf{main}}$. Any pair (x, f) where $x \in V$ and f is a function $(V \to \mathbb{D}) \to \mathbb{D}$ is called a constraint, were the right-hand side f is meant to provide a contribution to the value of x depending on some other values of constraint variables. The variable x is called the left-hand side of the constraint, whereas we refer to f as the right-hand side. A set of constraints form a *constraint system*. A variable assignment $\sigma \in V \to \mathbb{D}$ is a *solution* of the constraint system $\mathcal{C}$ if for all $(x, f) \in \mathcal{C}$, we have $\sigma\, x \sqsupseteq f\, \sigma$. Every constraint system has one trivial solution, namely, a function which maps all variables to the top element $\top \in \mathbb{D}$. In practice, though, we aim at computing *least*, or at least *non-trivial*, solutions of constraint systems.

Assume that $d_0 \in \mathbb{D}$ describes the program state before starting procedure main and that 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.

As usual in program analysis, these functions are assumed to be monotonic. Also, we assume the functions $[\![s]\!]^\sharp$ to be *strict*, i.e., preserve the value $\bot$. In the following, the value $\bot \in \mathbb{D}$ always represents the empty set of concrete program states, i.e., can only be assumed at a program point which is *unreachable*. Then, an initial abstract state $\bot \neq d_0 \in \mathbb{D}$ for the start point s_{main} together with the edges of the control flow graph give rise to the following system of constraints:

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

For better readability, each constraint $(x,\ \mathbf{fun}\ \mathsf{get} \to e)$ is denoted as "$x \sqsupseteq e$"; that is, get will always be the name of the first parameter of functions representing right-hand sides.

Since all abstract functions $[\![s]\!]^\sharp$ are assumed to be monotonic, constraint system (0) has a unique least solution. Moreover, the whole constraint system uses finitely many unknowns only, where the evaluation of the right-hand side of each constraint may also access finitely many unknowns. 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 some variant of worklist solver [14, 15, 28].

3 Analyzing Procedures

Sharir and Pnueli [25] describe two approaches to interprocedural program analysis. The *functional* approach tries to summarize the abstract effect of a procedure into a summary function. Many practical inter-procedural analyses, though, are based on complete lattices $\mathbb{D}$ where no effective representations for procedure summaries are known. This is already the case for inter-procedural full constant propagation. For such cases, Sharir and Pnueli 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 [5], the constraint variables for this approach 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 unknown $[v,d]$, where v belongs to a function g, thus represents the abstract value attained at program point v when g is called in context 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. We obtain the following constraint system for the variables $[v,d]$:

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

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 $d_2 = \mathsf{get}\,[r_g, d']$.

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 $\mathsf{get}\,[x,\ \mathsf{get}\,[y, d]]$, 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 expressions is, however, monotonic for variable assignments of which at least one is a *monotonic assignment*. 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 [10]. The least solution describes in some sense 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 $\mathsf{enter}^\sharp$ 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 unknowns depends on the number of elements in $\mathbb{D}$, which might be large. Often procedures are only called in few distinct abstract calling-contexts. In this case, *local* fixpoint iteration may succeed by starting from a set X of interesting variables, such as $X = \{[r_{\mathsf{main}}, d_0]\}$, and return a *partial* solution which contains the return values of the procedure summary for the required abstract calling-contexts only. 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 [9, 10].

Thus, the least 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 prefer to 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

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

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 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., joining the new value of an known with the one from 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 unknowns. 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 system (1').

4 Partial Context-Sensitivity

Local solving has difficulties when reachability is explicitly encoded in the constraint. We now consider analyses where tracking reachability explicitly is necessary not just for termination, but also for the result 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 for handling function calls then take the following form:

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

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 $\mathbf{1} = \{\bullet\}$, constraint system (2) generalizes a constraint system for callstring 0. 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, every solution of (2) provides a *sound* analysis information [2].

Assume for a moment that the complete lattice $\mathbb{D}_2$ of partial contexts is infinite. Then the same argument as in the last section 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 $\mathbb{D}_2 = \mathbf{1}$) 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 $\mathbf{1}$). 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.

5 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 [1, 24, 26] or concurrently running threads [29].

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 then is by modifying the abstract semantics $[\![s]\!]^\sharp$ to a function:

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

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:

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

In absence of procedures, this constraint system can be solved with Round-Robin iteration or some kind of worklist algorithm. 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:

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

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* unknowns $[u,b]$ 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.

6 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 by allowing multiple writes and also to 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. In side-effecting constraint systems, a constraint is given by a pair (x, f) where the right-hand side f now is a function $f \in (V \to \mathbb{D}) \to (V \to \mathbb{D} \to \mathsf{unit}) \to \mathbb{D}$. A call $\mathsf{get}\ y$ of the first argument function of f to some unknown $y \in V$ 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. A variable assignment $\sigma \in V \to \mathbb{D}$ is a *solution* to the 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 $\sigma\, y \sqsupseteq d$. If f is given by $\mathbf{fun}\ \mathsf{get} \to \mathbf{fun}\ \mathsf{set} \to e$ for an expression e, we again represent the constraint (x, f) by "$x \sqsupseteq e$".

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 4, 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:

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

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 unknown. Moreover, we have:

Theorem 1. *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).

Since every solution of constraint system (2) is a sound abstraction of the concrete semantics, Theorem 1 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 5 — 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:

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

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

7 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

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

of the abstract functions $\mathsf{enter}_e^\sharp$ 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:

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

8 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 the initial call to main, i.e. $[r_{\mathsf{main}}, d_0]$, it is based on *forward propagation*: whenever the abstract state at an unknown $[u, b]$ 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 [11]. We show that this forward propagation can be achieved if the following variant of the side-effecting constraint system (5) is used:

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

Theorem 2. *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)*.*

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 be 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/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.

9 Experimental Evaluation

Side-effecting constraint systems are at the heart of Goblint — a 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. For this paper, we considered a lockset analysis, 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 [8].

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 Smtp relay checker is 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 Sharir/Pnueli's forward propagating algorithm. 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.

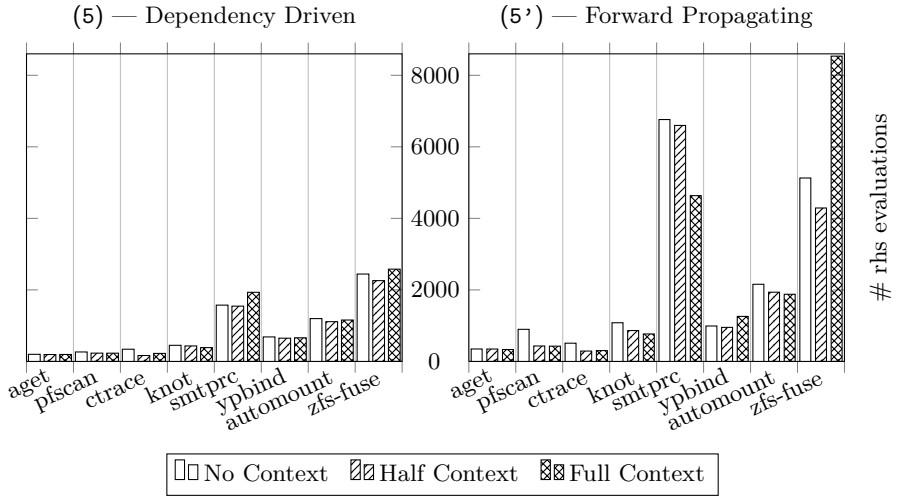

Fig. 1. Timing results.

For benchmarking we used an Ubuntu 11.04 machine with an Intel Core 2 Quad Q9650 3GHz processor, of which the analyzer currently utilizes a single core, and 4.0GB DDR2(800 MHz) memory. For all benchmarks and all configurations, the analyzer performs reasonably well — the full context analysis of the 24kloc program zfs-fuse takes 3.7 seconds only. For a fair comparison between the different configurations, we counted the number of evaluations of right-hand sides. Figure 1 displays these numbers for the given list of benchmarks, sorted according to their sizes. The Table 1 shows, for each program, the number of lines where a data race could not be ruled out. 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 the most efficient — for larger programs and forward propagation, sometimes more right-hand sides had to be evaluated. This shows that trading the number of updates against the number of constraint variables does not necessarily pay off. Secondly, the half context configuration turned out to be the most efficient (with the notable exception of benchmark smt_proc and forward propagation), while sacrificing no precision. 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.

Table 1. Precision Results.

		Lines with Warnings		
Name	Size(LoC)	No Ctx.	Half Ctx.	Full Ctx.
aget	1280	162	162	162
pfscan	1295	72	72	72
ctrace	1407	87	79	79
knot	2255	140	62	62
smtprc	5787	1068	636	636
ypbind	6596	251	244	244
automount	20624	505	480	480
zfs-fuse	24097	2319	2318	2318

For reproducibility of our results, the benchmarks, the system Goblint together with scripts to run the system in the various configurations can be downloaded from `http://goblint.github.com/side_effect` .

10 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* [8], which we 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.

Surprisingly, the constraint systems arising from interprocedural analysis with tabulation of partial contexts are not monotonic. Still, we plan to extend the widening/narrowing approach of Cousot and Cousot [4], that expects monotonicity, to our setting in order to allow analyses also to build upon very expressive lattices where ascending chains of elements may not be ultimately stable.

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

References

1. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. Ph.D. thesis, DIKU, University of Copenhagen (1994)
2. Apinis, K., Seidl, H., Vojdani, V.: Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis. Tech. Rep. TUM-I1213, Fakultät für Informatik, Tecnichse Universitet München, Germany (2012)
3. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL'09. pp. 289–300. ACM Press (2009)
4. Cousot, P., Cousot, R.: Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL'77. pp. 238–252. ACM Press (1977)
5. Cousot, P., Cousot, R.: Static Determination of Dynamic Properties of Recursive Procedures. In: IFIP Conf. on Formal Description of Programming Concepts. pp. 237–277. North-Holland (1977)
6. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE analyzer. In: ESOP'05. LNCS, vol. 3444, pp. 21–30. Springer (2005)
7. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL'78. pp. 84–96. ACM Press (1978)
8. Das, M., Lerner, S., Seigle, M.: ESP: Path-sensitive program verification in polynomial time. In: PLDI'02. pp. 57–68. ACM Press (2002)

9. Fecht, C.: Abstrakte Interpretation logischer Programme: Theorie, Implementierung, Generierung. Ph.D. thesis, Universität des Saarlandes (1997)
10. Fecht, C., Seidl, H.: A Faster Solver for General Systems of Equations. Sci. Comput. Program. 35(2), 137–161 (1999)
11. Flexeder, A., Mihaila, B., Petter, M., Seidl, H.: Interprocedural control flow reconstruction. In: APLAS'10. LNCS, vol. 6461, pp. 188–203. Springer (2010)
12. Guyer, S.Z., Lin, C.: Client-Driven pointer analysis. In: SAS'03. LNCS, vol. 2694, pp. 214–236. Springer (2003)
13. Hofmann, M., Karbyshev, A., Seidl, H.: Verifying a local generic solver in Coq. In: SAS'10, LNCS, vol. 6337, pp. 340–355. Springer (2010)
14. Jørgensen, B.: Finding fixpoints in finite function spaces using neededness analysis and chaotic iteration. In: SAS'94. LNCS, vol. 864, pp. 329–345. Springer (1994)
15. Kildall, G.: A unified approach to global program optimization. In: POPL'73. pp. 194–206. ACM Press (1973)
16. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO'04. pp. 75–88. IEEE Press (2004)
17. Lhoták, O., Chung, K.C.A.: Points-to analysis with efficient strong updates. In: POPL'11. pp. 3–16. ACM Press (2011)
18. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Jjava. ACM Transactions on Software Engineering and Methodology 14, 1–41 (Jan 2005)
19. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: Cil: An infrastructure for C program analysis and transformation. In: CC'02. LNCS, vol. 2304, pp. 213–228. Springer (2002)
20. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: POPL'95. pp. 49–61. ACM Press (1995)
21. Rinetzky, N., Bauer, J., Reps, T.W., Sagiv, S., Wilhelm, R.: A semantics for procedure local heaps and its abstractions. In: POPL'05. pp. 296–309 (2005)
22. Seidl, H., Vene, V., Müller-Olm, M.: Global invariants for analyzing multithreaded applications. Proc. of the Estonian Academy of Sciences: Phys., Math. 52(4), 413–436 (2003)
23. Seidl, H., Vojdani, V.: Region analysis for race detection. In: SAS'09. LNCS, vol. 5673, pp. 171–187. Springer (2009)
24. Shapiro, M., Horwitz, S.: Fast and accurate flow-insensitive points-to analysis. In: POPL'97. pp. 1–14. ACM Press (1997)
25. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Muchnick, S., Jones, N. (eds.) Program Flow Analysis: Theory and Application. pp. 189–233. Prentice-Hall (1981)
26. Steensgaard, B.: Points-to analysis in almost linear time. In: POPL'96. pp. 32–41. ACM Press (1996)
27. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot: A java bytecode optimization framework. In: CASCON'99. IBM Press (1999)
28. Vergauwen, B., Wauman, J., Lewi, J.: Efficient fixpoint computation. In: SAS'94. LNCS, vol. 864, pp. 314–328. Springer (1994)
29. Vojdani, V.: Static Data Race Analysis of Heap-Manipulating C Programs. Ph.D. thesis, University of Tartu (2010)
30. Wilson, R.P., French, R.S., Wilson, C.S., Amarasinghe, S.P., Anderson, J.M., Tjiang, S.W.K., Liao, S.W., Tseng, C.W., Hall, M.W., Lam, M.S., Hennessy, J.L.: SUIF: An infrastructure for research on parallelizing and optimizing compilers. SIGPLAN Not. 29, 31–37 (1994)