Softw Syst Model (2013) 12:285–306
DOI 10.1007/s10270-012-0230-7

SPECIAL SECTION PAPER

Relational interprocedural verification of concurrent programs

Bertrand Jeannet

Received: 17 June 2010 / Revised: 14 December 2011 / Accepted: 12 January 2012 / Published online: 10 March 2012

Abstract We propose a general analysis method for recursive, concurrent programs that track effectively procedure calls and return in a concurrent context, even in the presence of unbounded recursion and infinite-state variables like integers. This method generalizes the relational interprocedural analysis of sequential programs to the concurrent case, and extends it to backward or coreachability analysis. We implemented it for programs with scalar variables and experimented with several classical synchronization protocols in order to illustrate the precision of our technique and also to analyze the approximations it performs.

Keywords Concurrent program analysis · Interprocedural analysis · Abstract interpretation · Numerical abstract domains · Forward and backward analysis

1 Introduction

Interprocedural analysis of sequential programs is well understood in its principles [5,27,38] and more recent contributions concern mainly algorithmic techniques and/or alternative views [8,26,35,36]. However, the interprocedural analysis of concurrent programs is much harder: it is known to be undecidable, even when all data variables are finite [34], unlike in the sequential case [3].

Communicated by Prof. Krishnan, Dr. Cerone, and Dr. Van Hung.

This work has been supported by the Conseil Général de l'Isère and the Région Rhône-Alpes as part of the OpenTLM project (pôle de compétitivité Minalogic).

B. Jeannet (✉)
INRIA, Grenoble, France
e-mail: bertrand.jeannet@inrialpes.fr

We consider in this article the reachable-state analysis of concurrent programs with a fixed number of threads, recursive procedures and shared memory. The applications of such an analysis are numerous: deadlock detection, detecting data races, …This considered program model is rather general; it includes or can encode several models addressed in the literature, such as those considered in [33,34,40]. The main challenges in the analysis of such programs is to model the procedure call and return semantics in each thread, and to take into account the modification of global variables made by the other threads during the execution of the procedure of the current thread.

It is indeed the combination of recursion and concurrency which is difficult to tackle: in the case where the other threads do not modify shared variables, classical interprocedural techniques apply [5,27]; in the case where no thread performs procedure calls, one can reduce the concurrent program to a sequential one by considering the product of the control flow graphs (CFG) of all threads, as done in model checking. But the combination of the two features makes the reachability problem undecidable [34].

Look for instance at the program of Fig. 1, in which two threads synchronize by calling the procedure barrier implementing a synchronization barrier algorithm found in [40]. We would like to prove that the fail instruction of thread T1 is unreachable, because this would require that the thread T0 calls at least 503 times procedure barrier, whereas it calls it at most 502 times: there are 501 iterations in its loop, and there is a last call just after the loop. The challenge we want to tackle is to prove this unreachability property without inlining the procedure barrier, which may be called from several call sites in the program. Observe that this requires the analysis of the possible values of numerical variables. We need to prove that when each thread is at its loop head, we have $p0 = p1$. For discovering such properties, we plan to resort to

```
                    /* Shared global variables and initial condition */
                    var go : bool, counter,p0,p1:int;
                    initial counter==0 and go;

                    /* Procedure (shared by all threads) */
                    proc barrier(lgo:bool) returns (nlgo:bool)
                    begin
                      lgo = not lgo; counter = counter+1;
                      if counter==2
                      then counter=0; go = lgo;
                      else assume(lgo==go); /* Equivalent to wait until (lgo==go) */
                      endif;
                      nlgo = lgo;
                    end
/* Threads */
thread T0:                                  thread T1:
var lgo0:bool;                              var lgo1:bool;
begin                                       begin
  p0 = 0; lgo0 = true;                        p1 = 0; lgo1 = true;
  while p0<=500 do                            while p1<=502 do
    lgo0 = barrier(lgo0); p0 = p0 + 1;          lgo1 = barrier(lgo1); p1 = p1 + 1;
  done;                                       done;
  lgo0 = barrier(lgo0);                       /* Unreachable point */
end                                           fail;
                                            end
```

Fig. 1 Two threads synchronizing through a synchronization barrier procedure called inside counting loops

symbolic abstract interpretation techniques like convex polyhedra [7] and not to state-space enumeration techniques, the complexity of which depends on the magnitude of numerical constants.

Various approaches have been recently explored. A first approach is thread-modular analysis, in which one considers a thread interacting with a context that abstracts the possible steps of other threads [12]. Another option is to be less general on the class of the considered program: in [33] the authors define a notion of transactional procedures for which they succeed in summarizing the procedures. Another recently explored approach consists in focusing only on executions with a bounded number of context switches [28]. This restriction basically allows reducing the concurrent program to a sequential one, but the inferred invariants are not sound for any execution: they allow to discover bugs but they cannot prove a property.

We propose a method that analyzes all threads in parallel and tracks effectively procedure calls and returns in a concurrent context, even in the presence of unbounded recursion and infinite-state variables like integers. It is based on a generalization of relational interprocedural analysis of sequential programs. Relational interprocedural analysis is a technique for analyzing recursive programs in which the semantics of procedures are approximated by input/output relations that are characterized by fixpoint equations that are solved iteratively. In [26] we revisited this interprocedural analysis as an abstraction of the operational semantics of sequential programs. Technically, this abstraction consists mainly in collapsing call stacks into sets, in order to get rid of the source of infinity due to unbounded stacks, but *only after having appropriately instrumented the original semantics*.

We generalize this method to concurrent programs, in which each thread has it own call stack. After a suitable instrumentation, which defines the call context used to match procedure calls and returns, we apply to call stacks an abstraction which collapses separately the stack tail of each thread, but which takes the product of their stack tops, to relate the local environments of the different threads. This method can also be applied to *backward or coreachability analysis*, which is seldom mentioned for concurrent or interprocedural analysis.

The advantage of our method compared to the above-mentioned approaches is *better precision w.r.t. thread-modular techniques* [12], insofar as we can represent properties like $p0 = p1$ relating local environments of different threads, *generality* (termination is guaranteed on any program with unbounded recursion, unlike [33]), and *soundness* (all possible executions are taken into account, unlike [28] which is sound only w.r.t. the considered bound on context switches). In our method, termination and soundness follows from the abstract interpretation approach we adopt.

Besides the theoretical motivation, an important application we have in mind is the verification of SystemC/TLM (transaction-level-modeling) models of systems-on-chips (SoCs) [13], which are multithreaded C++ programs using a cooperative scheduling policy. Such an application requires in addition the resolution of virtual method invocation. This is not discussed in this paper, where we assume that the called procedure at a call site is statically known.

Contributions Our first contribution is to show that it is possible to analyze concurrent, recursive programs using relational techniques in the sense of [5,27], and to efficiently

tackle unbounded recursion, unlike most other techniques. Our second contribution is methodological: we use instrumentation to define how procedure calls and returns are matched; we then collapse unbounded stacks into sets to make the control finite, and we resort to data abstraction to deal with the remaining source of infinity. We apply this approach not only to forward analysis, but also to backward analysis, which we define precisely for recursive and concurrent programs, for the first time to our knowledge.

Our third contribution is experimental: We implemented our technique for programs with finite-type and numerical variables, and we experimented with several classical synchronization protocols that allow us to illustrate the precision of our technique and also to illustrate the approximations it performs.

Compared to our previously published conference paper [21], we considerably improved our notations; we added running examples and generalized our technique to backward analysis in an unified way.[1]

Outline Section 2 defines the program model we consider and its semantics, and Sect. 3 reminds some basics about abstract interpretation that are needed in the sequel.

In Sect. 4 we instrument the standard semantics for forward analysis, with information that will be exploited in the stack abstraction. Section 5 motivates and defines our *concurrent stack abstraction*, describes the induced forward abstract semantics and discusses optimality results. Section 6 defines an instrumented semantics for backward analysis and the backward analysis induced by the stack abstraction applied on it. We pinpoint in this section the partial duality between the forward and backward analyses.

We combine in Sect. 7 the stack abstraction with a data abstraction, to obtain an implementable analysis, and discuss its complexity. We eventually describe in Sect. 8 the experiments that we performed with our implementation. In Sect. 9, we discuss two improvements of our abstraction. Section 10 concludes and discusses related work.

2 Program model and standard semantics

2.1 Program model

We consider a simple concurrent imperative programming language with the following features:

(i) a program is composed of a *fixed number* of threads, interacting by the mean of shared global variables, and a set of non-nested procedures with a value parameter passing policy (as in JAVA or ML).

(ii) each procedure has its own set of local variables, and formal input and output parameters.

[1] Concerning notations, we put inside environments "program counter" variables, which allowed us to simplify a lot of the notations (standard and instrumented semantics, abstract postconditions), to clarify the different points and to show the duality between forward and backward analysis.

T^t	: Thread t
P_0^t	: Main procedure of thread T^t
P_i	: Any procedure
pc	: Program counter variable
$\mathbf{fp}_i$	: Formal input parameters of P_i
$\mathbf{fr}_i$	: Formal output parameters of P_i
$LVar_i, \mathbf{l}_i$	: Local variables of procedure P_i
	: (including $\mathbf{fp}_i$, $\mathbf{fr}_j$ & pc)
$GVar, \mathbf{g}$	: Global variables (set/vector)
$G = \langle K, I \rangle$	: Control flow graph
$\mathsf{s}_i, \mathsf{e}_i \in K$	: Entry and exit points of P_i in a CFG
$c \in K$	: Any control point in a CFG

Fig. 2 Syntactic domains and notations

We rely on shared global variables for communication and synchronization between threads. Figures 1 and 16, 17 and 18 give examples of such programs. The main restrictions are thus the absence of exceptions or non-local jumps, variable aliasing on the stack (as it happens with reference parameter passing), pointers to procedures and procedural parameters.

The syntactic and semantic domains we use are summarized in Figs. 2 and 5. Each *thread* T^t is defined by its main procedure, denoted as P_0^t. Each *procedure* $P_i = \langle \mathbf{fp}_i, \mathbf{fr}_i, \mathbf{l}_i, G_i \rangle$ is defined by its vector of formal input parameters $\mathbf{fp}_i$, output parameters $\mathbf{fr}_i$ and local variables $\mathbf{l}_i$ (that include formal parameters), and by its intraprocedural CFG (control flow graph) G_i.

The *intraprocedural CFG* of a procedure P_i is a graph $G = \langle K, I \rangle$ where

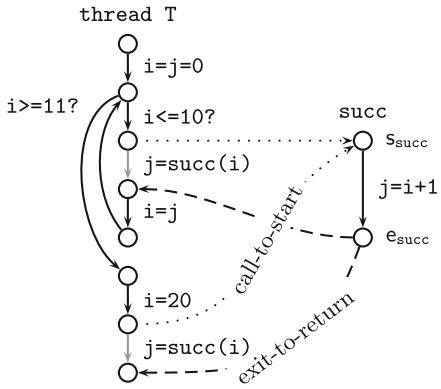
- K is the set of *control points* of P_i, containing unique entry and exit control points s_i and e_i;
- and $I : K \times K \to \mathsf{Inst}$ labels edges of the graph with two kinds of instructions:
  1. intraprocedural instructions $\langle R \rangle$, specified as a relation R between unprimed and primed variables that allow expressing tests and assignments;
  2. procedure calls $\langle \mathbf{y} := P_j(\mathbf{x}) \rangle$, where $\mathbf{x}$ and $\mathbf{y}$ are the vectors of actual input and output parameters; for the sake of simplicity and *w.l.o.g.*, we assume that $\mathbf{x}$ *and* $\mathbf{y}$ *are local variables* (unlike in [20,21]), so that these operations do not modify the value of global variables.

We assume that there are no two procedure call edges from the same point in G, that is, non-deterministic choices should be made *before* the call point. This allows us to define the functions $call$ and ret recording matching call and return-site nodes: if $I(c, c') = \langle \mathbf{y} := P_j(\mathbf{x}) \rangle$, then $call(c) = c'$ and $ret(c') = c$. $proc(c)$ denotes the index i of the procedure P_i that contains c.

Fig. 3 A single-thread program and its (global) CFG

```
proc succ(x:int) returns (y:int)
begin      (s_succ)
  y = x+1 (e_succ)
end

thread T:
var i,j:int;
begin
  i = j = 0;
  while i<=10 do (c1)
    j = succ(i); (c2)
    i = j;
  done;
  i=20;
  j=succ(i);
end
```



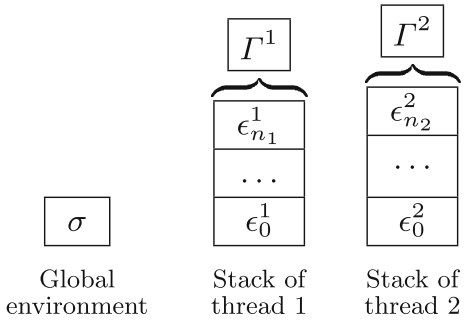


Fig. 4 Program state in standard semantics

The *global CFG G* (see Fig. 3) is constructed as the union of intraprocedural CFG G_is, further modified by replacing edges labeled by procedure calls by a *call-to-start* edge (connecting the call site to the entry point of the callee) and an *exit-to-return* edge (connecting the exit point of the callee to the return site). Thus there are three kinds of instructions in global CFGs: intraprocedural instructions $\langle R \rangle$, procedure calls $\langle \texttt{call}\ \mathbf{y} := P_j(\mathbf{x}) \rangle$ and procedure returns $\langle \texttt{ret}\ \mathbf{y} := P_j(\mathbf{x}) \rangle$.

2.2 Operational semantics

Without loss of generality, from now on we assume a program with only two threads (the general case being a bounded number N of threads). The semantic domains are summarized in Figs. 4 and 5.

A state $s = (\sigma, \Gamma^1, \Gamma^2)$ is defined by a global environment σ and the stacks Γ^t of *local environments* of the two threads. An environment maps variables to their values. We assume that local environments also contain the value of the *program counter* variable, denoted as *pc* and taking its values in the set of control points K.[2] In this respect, we merge the notions of environments and activation records. In the sequence $\Gamma \cdot \epsilon^t_{n_t}$, $\epsilon^t_{n_t}$ denotes the current or top local environment of the thread T^t. Environments can be updated with the notation $\epsilon[x \mapsto v]$. If $\mathbf{v}$, $\mathbf{v}'$ are vectors of variables, $\epsilon(\mathbf{v})$ denotes the corresponding vector of values, and $\mathbf{v} \setminus \mathbf{v}'$ denotes the sub-vector of $\mathbf{v}$ that does not contain any variable in $\mathbf{v}'$.

Figure 6 first defines the semantics of one thread in isolation (transition relation $\rightarrow^t$) for the three kind of instructions of a global CFG:

(Intra) An intraprocedural instruction, which is a guard or an assignment modeled by a relation R, acts only on the global environment and the top local environment.

(Call) A procedure call pushes a new local environment on the stack and initializes it according to the parameter passing policy. We use the auxiliary function R^+ defined by Eq. (R^+) to model this initialization, in which uninitialized variables are given arbitrary values.

(Ret) Similarly, a procedure return pops from the stack the top local environment and modifies the new top local environment according to the parameter passing convention, modeled by the function R^- defined by Eq. (R^-).

The transition relation $\rightarrow \subseteq S \times S$ induced by the full program is then defined by the rules (Conc1) and (Conc2) as a special asynchronous product of the two transition relations $\rightarrow^1$ and $\rightarrow^2$ that share the global environment. We define the initial set of states as $S_0 = \{\langle \sigma, \epsilon^1, \epsilon^2 \rangle \mid init(\sigma) \wedge \epsilon^1(pc) = \mathsf{s}^1_0 \wedge \epsilon^2(pc) = \mathsf{s}^2_0\}$ where s^t_0 denotes the start point of the main procedure of thread T^t, and *init* an initial condition on global variables.

2.3 Collecting semantics and analysis goal

The forward collecting semantics induced by a transition system $(S, \rightarrow)$ characterizes the set of reachable states of

[2] *pc* is effectively a *local* variable, as its value is pushed and popped from the stack during procedure calls and returns, unlike global variables.

Fig. 5 Semantic domains

v	$\in Value$	: values of expr. and variables
σ	$\in GEnv = GVar \to Value$	: global environments
ϵ, ε	$\in LEnv = LVar \to Value$	: local environments for a procedure
Γ, Υ	$\in LEnv^+$	: stacks of local environments
$\langle \sigma, \Gamma \rangle$	$\in S^t = GEnv \times LEnv^+$	: state of a thread in isolation
$\langle \sigma, \Gamma^1, \Gamma^2 \rangle$	$\in S = GEnv \times LEnv^+ \times LEnv^+$	: full program states

$$\frac{I(c,c') = \langle R \rangle \quad \epsilon(pc) = c \wedge \epsilon'(pc) = c' \ \wedge \ R(\sigma, \epsilon, \sigma', \epsilon')}{\langle \sigma,\ \Gamma \cdot \epsilon \rangle \to^t \langle \sigma',\ \Gamma \cdot \epsilon' \rangle} \ \text{(Intra)}$$

$$\frac{I(c, \mathtt{s}_j) = \langle \mathtt{call\ y} := P_j(\mathbf{x}) \rangle \quad R^+_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon, \epsilon_j)}{\langle \sigma,\ \Gamma \cdot \epsilon \rangle \to^t \langle \sigma,\ \Gamma \cdot \epsilon \cdot \epsilon_j \rangle} \ \text{(Call)}$$

$$\frac{I(\mathtt{e}_j, ret(c)) = \langle \mathtt{ret\ y} := P_j(\mathbf{x}) \rangle \quad R^-_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon, \epsilon_j,\ \epsilon')}{\langle \sigma,\ \Gamma \cdot \epsilon \cdot \epsilon_j \rangle \to^t \langle \sigma,\ \Gamma \cdot \epsilon' \rangle} \ \text{(Ret)}$$

$$\frac{\langle \sigma, \Gamma^1 \rangle \to^1 \langle \sigma', \Upsilon^1 \rangle}{\langle \sigma, \Gamma^1, \Gamma^2 \rangle \to \langle \sigma', \Upsilon^1, \Gamma^2 \rangle} \ \text{(Conc1)} \qquad \frac{\langle \sigma, \Gamma^2 \rangle \to^2 \langle \sigma', \Upsilon^2 \rangle}{\langle \sigma, \Gamma^1, \Gamma^2 \rangle \to \langle \sigma', \Gamma^1, \Upsilon^2 \rangle} \ \text{(Conc2)}$$

$$R^+_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon, \epsilon_j) = \begin{cases} \epsilon(pc) = c \\ \epsilon_j(pc) = \mathtt{s}_j \\ \epsilon_j(\mathbf{fp}) = \epsilon(\mathbf{x}) \end{cases} \ (\mathrm{R}^+)$$

$$R^-_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon, \epsilon_j,\ \epsilon') = \begin{cases} \epsilon(pc) = c \\ \epsilon_j(pc) = \mathtt{e}_j \\ \epsilon' = \epsilon \left[\begin{matrix} pc \mapsto ret(c) \\ \mathbf{y} \mapsto \epsilon_j(\mathbf{fr}) \end{matrix} \right] \end{cases} \ (\mathrm{R}^-)$$

Fig. 6 Standard Operational Semantics: transition relation $\to^t$ of the thread T^t and transition relation $\to$ of the full program. The relations R^+ and R^- on environments define parameter passing mechanisms

a program from a set of *initial states* $S_0 \subseteq S$. We first define the postcondition operator $post(X)$ which we decompose according to the interprocedural CFG and the thread T^t performing an instruction:

$$post(X) = \{s' \mid \exists s \in X : s \to s'\} = \bigcup_{(c,c') \in K \times K} \bigcup_t post^t(\underbrace{c \xrightarrow{I(c,c')} c'}_{\tau})(X) \tag{1}$$

$post^t(\tau)(X)$ is deduced from the semantic rules of Fig. 6. For instance:

$$post^1(c \xrightarrow{\langle \mathtt{call\ y}:=P_j(\mathbf{x}) \rangle} \mathtt{s}_j) = \left\{ \langle \sigma,\ \Gamma^1 \cdot \epsilon \cdot \epsilon_j, \Gamma^2 \rangle \ \middle|\ \begin{matrix} \langle \sigma,\ \Gamma^1 \cdot \epsilon, \Gamma^2 \rangle \in X \\ R^+_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon, \epsilon_j) \end{matrix} \right\}$$

The set $reach(S_0)$ of reachable states from initial states $S_0 \subseteq S$ corresponds to the smallest solution of $X = S_0 \cup post(X)$: a state is reachable if it is either initial or the successor of some reachable state. Since the forward transfer function $F[S_0](X) = S_0 \cup post(X)$ is monotone and continuous, according to Kleene's theorem the set of reachable states may be defined as

$$reach(S_0) = lfp(F[S_0]) = \bigcup_{n \geq 0} (F[S_0])^n(\emptyset) \tag{2}$$

By duality, the backward collecting semantics characterizes the set $coreach(S_1)$ of states coreachable from (i.e. leading to) a set of *final states* S_1. It is the natural choice for inferring or checking a necessary condition on a program state to reach a final, typically erroneous configuration. As for reachable states, a state is coreachable if it is either final or the predecessor of some coreachable state. We get thus a least fixpoint characterization similar to Eq. (2):

$$coreach(S_1) = lfp(G[S_1]) \tag{3}$$
$$\text{with } G[S_1](X) = S_1 \cup pre(X)$$
$$pre(X) = \{s \mid \exists s' \in X : s \to s'\}$$

3 Abstract interpretation

Our analysis will be formalized and proved sound using the abstract interpretation framework. We thus remind some definitions and properties about abstract interpretation and refer to [6] for a more detailed presentation. We reminded in Sect. 2.3 that reachability analysis reduces to solving the fixpoint Eq. (2) in the complete lattice $L = \wp(S)$ ordered by set inclusion. The idea of abstract interpretation is to transpose such an equation to a simpler abstract lattice $L^\sharp$, such that the two lattices forms a *Galois connection* denoted as $(L, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (L^\sharp, \sqsubseteq^\sharp)$, meaning that $\forall x \in L, \forall y \in L^\sharp : \alpha(x) \sqsubseteq^\sharp y \Leftrightarrow x \sqsubseteq \gamma(y)$. Such a Galois connection between two complete lattices induces the following properties on abstraction and concretization functions:

(i) α and γ are monotone (increasing);
(ii) α is distributive for the least upper bound
(iii) $\alpha \circ \gamma$ is retractive ($\alpha \circ \gamma \sqsubseteq id$);
(iv) $\gamma \circ \alpha$ is extensive ($\gamma \circ \alpha \sqsupseteq id$).

Given a continuous function $F : L \to L$, the standard fixpoint transfer theorem says that if one take a *correct approximation* $F^\sharp \sqsupseteq^\sharp \alpha \circ F \circ \gamma$ of F in $L^\sharp$, then

$$lfp(F^\sharp) \sqsupseteq \alpha(lfp(F)) \quad \text{or} \quad \text{equivalently } \gamma(lfp(F^\sharp)) \sqsupseteq lfp(F)$$

$$\begin{array}{ccc} L & \xrightarrow{F} & L \\ \gamma \uparrow & & \downarrow \alpha \\ L^\sharp & \xrightarrow[F^\sharp]{} & L^\sharp \end{array} \qquad \begin{array}{ccc} \bullet & \sqsupseteq & lfp(F) \\ \gamma \uparrow & & \downarrow \alpha \\ lfp(F^\sharp) & \sqsupseteq^\sharp & \bullet \end{array}$$

Considering for F the forward transfer function $F[S_0](X)$ introduced in Sect. 2.3 with $L = \wp(S)$, this theorem allows computing an overapproximation of reachable states in a simpler lattice $L^\sharp$, using a correct approximation of $F[S_0](X)$ in $L^\sharp$.

If we consider the stronger hypothesis $F^\sharp \circ \alpha = \alpha \circ F$, which implies the previous one, we get the stronger result

$$lfp(F^\sharp) = \alpha(lfp(F)) \tag{4}$$

When designing an abstract interpretation, it is highly desirable (but not always possible) to satisfy such a hypothesis for abstract transfer functions.

In practice, one does not abstract the "global" transfer function $F[S_0]$; instead, one exploits the decomposition shown in Eq. (1) to abstract separately the semantics of each instruction in the CFG of the thread T^t $post_t(c \xrightarrow{I(c,c')} c') : \wp(S) \to \wp(S)$ with a function $post_t^\sharp(c \xrightarrow{I(c,c')} c') : L^\sharp \to L^\sharp$.

4 Instrumenting the standard semantics for forward analysis

In this section, we instrument the operational semantics for forward analysis. The idea is to tag local environments of procedures with information about their call context, and to use such tags to match procedure calls and returns.

If we consider a procedure P in thread 1 of a two-thread program, its call context is defined by

1. The global variable and formal parameters at its start point in thread 1;
2. The full call stack of thread 2: During the execution of P in thread 1, thread 2 can indeed perform several procedure returns and then calls again new procedures, with execution steps modifying the global variables and interacting with P. This dependency on the full call stack of the other thread(s) is the intuitive reason why the combination of concurrency and recursion makes the reachability analysis undecidable even for Boolean programs.

We *choose* here to take into account only part (1) of the call context, and we delay alternative choices to Sect. 9. Because P may modify the global variables and formal parameters $\mathbf{g}$, $\mathbf{fp}$ during its execution, we introduce in local environments copies $\mathbf{g}_0$, $\mathbf{fp}_0$ that contain at any point of P the value of $\mathbf{g}$ and $\mathbf{fp}$ at its start point.

The second orthogonal point of our instrumentation is to push global variables into call stacks.

In the instrumented semantics, the new environments $\epsilon \in Env$ are thus defined on variables $\mathbf{g}_0$, $\mathbf{fp}_0$, $\mathbf{g}$, $\mathbf{l}$, where the values of $\mathbf{g}_0$, $\mathbf{fp}_0$ keep track of the call context at start point of the current procedure. The new state-space is

$$S_f = Env^+ \times Env^+ \tag{5}$$

Figure 7 defines the new semantic rules induced by the standard semantics modified by pushing global variables on stacks (relations (R), (R^+), (R^-)) and adding auxiliary variables (relations (R^f), (R^{f+}), (R^{f-})). Rules (IntraF), (CallF) and (RetF) have the same structure as in Fig. 6, whereas rules (Conc1F) and (Conc2F) have now to update the modification of global variables induced by one thread to the other thread, so that the top environments of the concurrent stacks always agree on the current value of global variables. *This is needed only after intraprocedural instructions*, because of the assumption that input and output parameters are local variables; see Sect. 2.1 and Fig. 6.

In this semantics, reachable call stacks are necessarily *well formed* in the following sense.

Definition 1 (*Well-formed stacks and states*) A stack $\Gamma = \epsilon_0 \dots \epsilon_n \in Env^+$ is *well formed* if, for any $i < n$:

(i) $c_i = \epsilon_i(pc)$ is a call site for the procedure P_j, with $j = proc(c_{i+1})$, $c_{i+1} = \epsilon_{i+1}(pc)$ and $I(c_i, \mathtt{s}_j) = \langle \mathtt{call}\ \mathbf{y} := P_j(\mathbf{x}) \rangle$;
(ii) equality between actual and formal input parameters holds at call sites: $\epsilon(c_{i+1}) = \mathtt{s_j} \Rightarrow \epsilon_i(\mathbf{g}, \mathbf{x}) = \epsilon_{i+1}(\mathbf{g}, \mathbf{fp}^j)$.
(iii) equality between actual and copies of formal input parameters holds at any point: $\epsilon_i(\mathbf{g}, \mathbf{x}) = \epsilon_{i+1}(\mathbf{g}_0, \mathbf{fp}_0^j)$.

A state $\langle \Gamma^1, \Gamma^2 \rangle \in S_f$ is *well formed* if Γ^1 and Γ^2 are well formed, and if the top environments ϵ^1, ϵ^2 of Γ^1, Γ^2 satisfies $\epsilon^1(\mathbf{g}) = \epsilon^2(\mathbf{g})$.

For instance, if one considers the prog. of Fig. 3, a stack of the form [4] $(pc = \mathtt{c_1} \wedge i = 30) \cdot (pc = \mathtt{s_{succ}} \wedge x_0 = x = 30)$ is well formed (although not reachable), but a stack of the

Fig. 7 Instrumented semantics for forward analysis: transition relation $\rightarrow_f^t$ of the thread T^t and transition relation $\rightarrow_f$ of the full program

Pushing global variables $\mathbf{g}$ on stacks

$$R(c,c')(\epsilon,\epsilon') = \epsilon(pc)=c \wedge \epsilon'(pc)=c' \wedge R(\epsilon,\epsilon') \quad (\mathrm{R})$$

$$R^{+}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon,\epsilon'_j) = \begin{cases} \epsilon(pc)=c \wedge \epsilon'_j(pc)=\mathsf{s}_j \\ \epsilon'_j(\mathbf{g},\mathbf{fp})=\epsilon(\mathbf{g},\mathbf{x}) \end{cases} \quad (\mathrm{R}^{+})$$

$$R^{-}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon,\epsilon_j,\ \epsilon') = \begin{cases} \epsilon(pc)=c \wedge \epsilon_j(pc)=\mathsf{e}_j \\ \epsilon'(pc)=ret(c) \\ \epsilon'(\mathbf{l}\backslash\mathbf{y})=\epsilon(\mathbf{l}\backslash\mathbf{y}) \ \wedge\ \epsilon'(\mathbf{g},\mathbf{y})=\epsilon_j(\mathbf{g},\mathbf{fr}) \end{cases} \quad (\mathrm{R}^{-})$$

Adding forward instrumentation variables $\mathbf{g}_0, \mathbf{fp}_0$

$$R^{f}(c,c')(\epsilon,\epsilon') = \begin{cases} R(c,c')(\epsilon,\epsilon') \\ \epsilon'(\mathbf{g}_0,\mathbf{fp}_0)=\epsilon(\mathbf{g}_0,\mathbf{fp}_0) \end{cases} \quad (\mathrm{R}^{\mathrm{f}})$$

$$R^{f+}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon,\epsilon'_j) = \begin{cases} R^{+}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon,\epsilon'_j) \\ \epsilon'_j(\mathbf{g}_0,\mathbf{fp}_0)=\epsilon(\mathbf{g},\mathbf{x}) \end{cases} \quad (\mathrm{R}^{\mathrm{f}+})$$

$$R^{f-}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon,\epsilon_j,\ \epsilon') = \begin{cases} R^{-}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon,\epsilon_j,\ \epsilon') \\ \epsilon'(\mathbf{g}_0,\mathbf{fp}_0)=\epsilon(\mathbf{g}_0,\mathbf{fp}_0) \end{cases} \quad (\mathrm{R}^{\mathrm{f}-})$$

Semantic rules

$$\frac{I(c,c')=\langle R\rangle \qquad R^{f}(c,c')(\epsilon,\epsilon')}{\Gamma\cdot\epsilon \rightarrow_f^t \Gamma\cdot\epsilon'} \quad (\mathrm{IntraF})$$

$$\frac{I(c,\mathsf{s}_j)=\langle \texttt{call}\ \mathbf{y}:=P_j(\mathbf{x})\rangle \qquad R^{f+}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon,\epsilon'_j)}{\Gamma\cdot\epsilon \rightarrow_f^t \Gamma\cdot\epsilon\cdot\epsilon'_j} \quad (\mathrm{CallF})$$

$$\frac{I(\mathsf{e}_j,ret(c))=\langle \texttt{ret}\ \mathbf{y}:=P_j(\mathbf{x})\rangle \qquad R^{f-}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon,\epsilon_j,\ \epsilon')}{\Gamma\cdot\epsilon\cdot\epsilon_j \rightarrow_f^t \Gamma\cdot\epsilon'} \quad (\mathrm{RetF})$$

$$\frac{\Gamma^1 \rightarrow_f^1 \Upsilon^1 \qquad \Upsilon^1=\Upsilon\cdot\varepsilon^1 \qquad \varepsilon^2=\epsilon^2[\mathbf{g}\mapsto\varepsilon^1(\mathbf{g})]}{\langle\Gamma^1,\ \Gamma^2\cdot\epsilon^2\rangle \rightarrow_f \langle\Upsilon^1,\ \Gamma^2\cdot\varepsilon^2\rangle} \quad (\mathrm{Conc1F})$$

$$\frac{\Gamma^2 \rightarrow_f^2 \Upsilon^2 \qquad \Upsilon^2=\Upsilon\cdot\varepsilon^2 \qquad \varepsilon^1=\epsilon^1[\mathbf{g}\mapsto\varepsilon^2(\mathbf{g})]}{\langle\Gamma^1\cdot\epsilon^1,\ \Gamma^2\rangle \rightarrow_f \langle\Gamma^1\cdot\varepsilon^1,\ \Upsilon^2\rangle} \quad (\mathrm{Conc2F})$$

form $(pc=\mathsf{c}_1 \wedge i=3)\cdot(pc=\mathsf{e}_{\mathtt{succ}} \wedge x_0=4)$ is not, because it violates condition (iii): actual parameter i does not match formal parameter x_0.

Proposition 1 *If $s \in S_f$ is a well-formed state, then any $s' \in S_f$ such that $s \rightarrow_f^* s'$ is a well-formed state.*

Proof (i) and (ii) are satisfied in the standard semantics. For (iii) the only rule defining the value of $\mathbf{g}_0, \mathbf{fp}_0$ in an environment is the rule (CallF), using the relation ($\mathrm{R}^{\mathrm{f}+}$). (iii) is thus satisfied just after a procedure call by the two top environments $\epsilon_{tl}\cdot\epsilon$. Later in the execution, as long as ϵ is not popped from the stack, neither the value of $\mathbf{x}$ in ϵ_{tl} nor the value of $\mathbf{g}_0, \mathbf{fp}_0$ in ϵ can change, so (iii) is preserved. □

With this notion of well-formed state, we get a strong conditions for an environment to lie below another environment in reachable call stacks. Indeed, initial states with stacks of height 1 are always well formed, thus Proposition 1 applies to reachability analysis from such initial states.

5 Forward analysis

In this section, we present our reachability analysis for concurrent and recursive programs. We start by discussing the classical techniques that inspired us for analyzing programs that are either recursive or concurrent. We then give in Sect. 5.2 an axiomatic presentation of our analysis, before formalizing it and proving its soundness by abstract interpretation in Sect. 5.3.

5.1 Two sources of inspiration

5.1.1 Relational interprocedural analysis of sequential programs

In interprocedural analysis, the operation that is difficult to model in the analysis is the procedure return operation. Consider the program in Fig. 8. Inferring the invariant at the start of the procedure `succ` from the invariants at the call sites, or propagating the invariant from its start to its exit

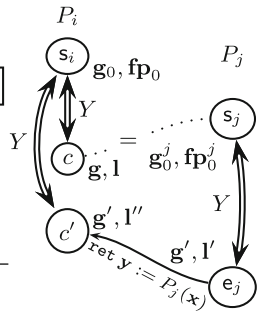
```
proc succ(x:int) returns (y:int)
begin   0≤x≤10 ∨ x=20
  y = x+1
end     (0≤x≤10 ∨ i=20) ∧ y=x+1
var i,j:int;
begin
  i = j = 0;
  while i<=10 do
    j = succ(i);0≤i≤10 ∧ j=i+1
    i = j;       1≤i≤11 ∧ j=i
  done;
  i=20; j=succ(i);    i=20 ∧ j=21
end
```

Fig. 8 Interprocedural analysis of the program of Fig. 3

point are rather easy. But for the procedure return, one has to combine two important information: $y = x + 1$ at the exit point of `succ`, and $0 \leq i \leq 10$ or $i = 20$ at the call sites. This combination is basically a relation composition, but it is complexified by details related to the parameter passing policy.

The first technique that inspired us is thus the functional or relational approach described in [5,27], in which one associates at each control point a relation between the input state and the current state of the enclosing procedure, so that at the exit point of a procedure P one obtains its input/output *summary* capturing the effect of a call to P. If we describe this technique with our notations, it manipulates predicates of the form $Y(\mathbf{g}_0, \mathbf{fp}_0, \mathbf{g}, \mathbf{l})$, relating the reachable input state of a procedure (variables $\mathbf{g}_0, \mathbf{fp}_0$ introduced in Sect. 4) and the reachable current state (variables $\mathbf{g}, \mathbf{l}$). The rule for the procedure return operation is:

$$\frac{\begin{array}{c} I(\mathtt{e}_j, ret(c)) = \langle \mathtt{ret\ y} := P_j(\mathbf{x}) \rangle \\ Y(\mathbf{g}_0, \mathbf{fp}_0,\ \mathbf{g}, \mathbf{l}) \quad \boxed{Y(\mathbf{g}_0^j, \mathbf{fp}_0^j,\ \mathbf{g}', \mathbf{l}')} \\ \text{(call-site)} \qquad \text{(exit-site)} \\ (\mathbf{g}, \mathbf{x}) = (\mathbf{g}_0^j, \mathbf{fp}_0^j) \\ \text{(unification of actual and formal parameters)} \\ R^{f-}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\mathbf{l}, \mathbf{l}',\ \mathbf{l}'') \\ \text{(output parameter passing)} \end{array}}{\boxed{Y(\mathbf{g}_0, \mathbf{fp}_0,\ \mathbf{g}', \mathbf{l}'')} \quad \text{(return-site)}} \qquad (6)$$



(Remember that the relation $R^{f-}_{\mathbf{y}:=P_j(\mathbf{x})}$ constrains the value of the *pc* variable contained in $\mathbf{l}$, $\mathbf{l}'$ and $\mathbf{l}''$, hence the different instances of the Y predicate in Eq. (6) are associated with the mentioned control points.)

This rule implements the above-mentioned relation composition between Y at the call site and Y at the exit site, by unifying the actual $(\mathbf{g}, \mathbf{x})$ and formal $(\mathbf{g}_0, \mathbf{fp}_0^j)$ parameters, and eliminating them in Y at the return site. Coming back to the example of Fig. 8, which does not contain global variables, for the first call to `succ`, we have

$$\frac{\begin{array}{c} 0 \leq i \leq 10 \wedge j = i \qquad (0 \leq x_0 \leq 10 \vee x_0 = 20) \wedge y' = x_0 + 1 \\ \text{(call site)} \qquad\qquad \text{(exit-site)} \\ i = x_0 \\ \text{(unification of actual and formal parameters)} \\ i'' = i \wedge j'' = y' \\ \text{(output parameter passing)} \end{array}}{0 \leq i'' \leq 10 \wedge j'' = i'' + 1 \quad \text{(return-site)}}$$

Jeannet and Serwe [26] formalize this approach by stack abstraction: starting from the instrumented semantics of Sect. 4, it defines the Galois connection $\wp(Env^+) \underset{\alpha_f}{\overset{\gamma_f}{\leftrightarrows}} \wp(Env)$ with [3]

$$\alpha_f : \{\Gamma = \epsilon_0 \ldots \epsilon_n\} \mapsto \{\epsilon_i | 0 \leq i \leq n\}$$
$$\gamma_f : Y \mapsto \left\{ \Gamma = \epsilon_0 \ldots \epsilon_n \;\middle|\; \begin{array}{ll} \forall & 0 \leq i \leq n : \epsilon_i \in Y \\ \Gamma & \text{is a well-formed stack} \end{array} \right\} \qquad (7)$$

In the induced abstract semantics, when computing the effect of a procedure return, rule (RetF) of Fig. 7, the well-formedness condition in the definition of γ_f allows to match pairs of tail and top environments with the condition (iii) of Definition 1, so as to implement the relation composition of Eq. (6).

5.1.2 *Analysis of concurrent systems with an interleaving semantics*

The second technique which inspired us is the classical method used for the analysis of non-recursive concurrent systems. Such systems have a state-space of the form $S = GEnv \times LEnv^1 \times LEnv^2 \simeq (K^1 \times K^2) \times Env$, if one partitions the state-space according to the value of the *pc* variables. The usual technique for verifying such systems is to consider the product of the CFGs by observing that

$$\wp(S) \simeq K^1 \times K^2 \to \wp(Env) \qquad (8)$$

The ability to relate the local environments of concurrent threads is fundamental:

1. There is a technical reason. Consider the program of Fig. 9 in which the threads synchronize their parallel execution by *rendez-vous* on a channel `a` (this can be implemented using global shared variables). We want to prove that the `fail` instruction in thread `T2` is not reachable. Assume that we maintain *separate* predicates $Y^1(g, l^1)$ and $Y^2(g, l^2)$ for each pair of control points. Just after the first synchronization, we have $Y^1(g, l^1) = (g = l^1)$ and $Y^2(g, l^2) = (g = l^2 - 1)$. Now the only possible step is that thread 1 executes the instruction $g := g + l^1$.
   It is easy to compute its effect on the predicate Y^1 (one obtains $g = 2l^1$), but less so on the predicate Y^2. The only way to perform this is actually to build $Y = Y^1 \wedge Y^2 =$

[3] Jeannet and Serwe [26] actually distinguishes stack tops and stack tails, whereas we do not here.

```
var g:int;
thread T1:    thread T2:
var l1:int;   var l2:int;
begin         begin
  l1 = g;       l2 = g+1;
  sync a;       sync a;
  g = g+l1;
  sync a;       sync a;
end             if g>=2*l2 then fail;
              end
```

Fig. 9 Example illustrating the problem of maintaining relations between global and local variables in each thread

$(g = l^1 \wedge l^1 = l^2 - 1)$, to compute the effect of the instruction on Y, and then to forget the variable l^1: One obtains $g = 2l^2 - 2$ before the second synchronization, hence fail will not be executed.

The conclusion is that when a global variable is assigned in one thread, one needs to relate the local environments of different threads, at least temporarily, to maintain the relation between global and local variables in the other threads.

2. There is also a precision reason. Consider now the program of Fig. 10. In order to establish that the loop of the thread `T2` does not terminate (the *rendez-vous* induces a deadlock when $j = 11$), we need to infer the invariant $i = j$ when each thread is at the control point just after the synchronization instruction. If the possible environments of each thread are inferred separately, the non-termination of the thread `T2` cannot be proved.

5.2 Forward analysis: an axiomatic presentation and analysis example

The technique we propose in this paper for reachability analysis combines the two techniques presented above. It starts from the *instrumented semantics* of Sect. 4 and can be defined in an axiomatic way similar to Eq. (6) using three predicates:

- $Y_{hd}(\epsilon^1, \epsilon^2)$ that represents the set of reachable valuations for the product of the two threads. We point out that we will always have $\epsilon^1(\mathbf{g}) = \epsilon^2(\mathbf{g})$, because in the instrumented semantics any update of a global variable by one thread is propagated to the other thread (see Fig. 7), and because our abstraction will maintain this property.
  If $\epsilon^1(pc) = \mathsf{e}_j$ is the exit point of procedure P_j, $Y_{hd}(\epsilon^1, \epsilon^2) \simeq Y_{hd}(\mathbf{g}_0^1, \mathbf{fp}_0^1, \mathbf{l}^1, \ \mathbf{g}_0^2, \mathbf{fp}_0^2, \mathbf{l}^2, \ \mathbf{g})$ can be seen as a relation between the input $(\mathbf{g}_0^1, \mathbf{fp}_0^1)$ and the output $(\mathbf{g}, \mathbf{l}^1)$ of P_j in thread 1, which depends also on the local state $\mathbf{g}_0^2, \mathbf{fp}_0^2, \mathbf{l}^2$ of thread 2. This relation takes into account the moves that thread 2 may have performed since thread 1 started the execution of P_j.
- For $t = 1, 2$, an auxiliary predicate $Y_{tl}^t(\epsilon^t)$, which gives set of reachable valuations for thread t. We will have in this section

```
thread T1:       thread T2:
var i:int;       var j:int;
begin            begin
  i = 0;           j = 0;
  while i<=10 do   while j<=11 do
    sync a;          sync a;
    i = i+1;         j = j+1;
  done             done
end              end
```

Fig. 10 Example illustrating the need of relating local variables of different threads

$$Y_{tl}^1(\epsilon^1) = \exists \epsilon^2 : Y_{hd}(\epsilon^1, \epsilon^2)$$

and conversely for Y_{tl}^2 (this will not hold any more for backward analysis in Sect. 6).

The explanation of the indices $_{hd}$ and $_{tl}$ will be made explicit in Sect. 5.3.

If we focus on the transitions performed by thread 1, the abstract procedure return corresponding to Eq. (6) becomes:

$$\frac{\begin{array}{c} I(\mathsf{e}_j, ret(c)) = \langle \mathtt{ret}\ \mathbf{y} := P_j(\mathbf{x}) \rangle \\ \underset{\text{(call-site)}}{Y_{tl}^1(\epsilon_{tl})} \quad \underset{\text{(exit-site)}}{\boxed{Y_{hd}(\epsilon, \epsilon^2)}} \quad \underset{\text{(unification of parameters)}}{\epsilon_{tl}(\mathbf{g}, \mathbf{x}) = \epsilon(\mathbf{g}_0, \mathbf{fp}_0)} \\ \underset{\text{(output parameter passing)}}{R^{f-}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon_{tl}, \epsilon, \epsilon')} \end{array}}{\boxed{Y_{hd}(\epsilon', \epsilon^2)} \quad \text{(return-site for thread 1)}} \tag{9}$$

The rules for procedure calls and intraprocedural instructions are simpler and reflect the instrumented semantics of Fig. 7.

$$\frac{I(c, \mathsf{s}_j) = \langle \mathtt{call}\ \mathbf{y} := P_j(\mathbf{x}) \rangle \quad \boxed{Y_{hd}(\epsilon, \epsilon^2)} \quad R^{f+}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon, \epsilon')}{\boxed{Y_{tl}^1(\epsilon)} \quad \boxed{Y_{hd}(\epsilon', \epsilon^2)}} \qquad \frac{I(c, c') = \langle R \rangle \quad \boxed{Y_{hd}(\epsilon, \epsilon^2)} \quad R^f(c, c')(\epsilon, \epsilon') \quad (\epsilon^2)' = \epsilon^2[\mathbf{g} \mapsto \epsilon'(\mathbf{g})]}{\boxed{Y_{hd}\left(\epsilon', (\epsilon^2)'\right)}} \tag{10}$$

It should be clear that this analysis extends both the relational interprocedural analysis described in Sect. 5.1.1, by the way it defines the effect of a procedure return, and the analysis of concurrent systems described in Sect. 5.1.2, because it manipulates the product Y_{hd} of the two threads.

As an example, we shall analyze the program of Fig. 11. For the sake of simplicity, we assume that the execution starts in thread 1, and that context switches can occur only in control point marked with the symbol "*". For readability purpose, we will partition Y according to the value of local program counter variables pc^1 and pc^2. The induced dependency graph between the $Y(pc^1, pc^2)$'s is depicted on Fig. 11. Dashed lines indicate call-to-start edges, and dotted lines exit-to-return edges.

```
var g:int;
initial g==0;
proc succ(x:int) returns (y:int)
begin      (s)*
  y = x+1;
  g = g+1; (e)
end

thread T1:                    thread T2:
var i,j:int;
begin                         begin
  assume(0<=i<=10); (s1)*              (s2)
  j = succ(i);      (c1)
  j = succ(i);      (e1)*     g = g+1;(e2)*
end                           end
```

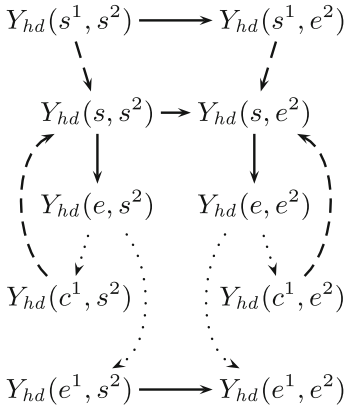


Fig. 11 Program analyzed with concurrent relational analysis and the dependency graph between abstract invariants. *Dashed and dotted arrows* correspond resp. to call-to-start and exit-to-return edges. *Vertical arrows* correspond to steps performed by thread 1, *horizontal arrows* to steps performed by thread 2

We detail below the iterative solving of the induced fixpoint equations. We write $Y(\ldots) = Y(\ldots)_k$ when the value $Y(\ldots)_k$ at the kth iteration is equal to the least fixpoint value $Y(\ldots)$.

The first iteration produces the following results:

$$
\begin{aligned}
Y_{hd}(s^1,s^2) = Y_{hd}(s^1,s^2)_1 &= g=0 && \wedge\ 0\le i^1\le 10\\
Y_{hd}(s,s^2)_1 &= g_0^1=0 \wedge g=g_0^1 && \wedge\ 0\le x_0^1\le 10\\
Y_{hd}(e,s^2)_1 &= g_0^1=0 \wedge g=g_0+1 && \wedge\ 0\le x_0^1\le 10 \wedge y^1=x_0^1+1\\
Y_{hd}(s^1,e^2) = Y_{hd}(s^1,e^2)_1 &= g=1 && \wedge\ 0\le i^1\le 10\\
Y_{hd}(s,e^2)_1 &= g_0^1=0 \wedge g=g_0^1+1 && \wedge\ 0\le x_0^1\le 10 \quad \text{(image of } Y_{hd}(s,s^2)_1)\\
&\vee\ g_0^1=1 \wedge g=g_0^1 && \wedge\ 0\le x_0^1\le 10 \quad \text{(image of } Y_{hd}(s^1,e^2))\\
&= 0\le g_0^1\le 1 \wedge g=1 && \wedge\ 0\le x_0^1\le 10\\
Y_{hd}(e,e^2)_1 &= 0\le g_0^1\le 1 \wedge g=2 && \wedge\ 0\le x_0^1\le 10 \wedge y^1=x_0^1+1\\
Y_{tl}^1(s^1) &= Y_{hd}(s^1,s^2) \vee Y_{hd}(s^1,e^2) = 0\le g\le 1 \wedge 0\le i^1\le 10
\end{aligned}
$$

For $Y_{hd}(c^1, s^2)_1$, we instantiate Eq. (9) as follows:

$$
\frac{\overbrace{0\le i_{tl}^1\le 10\wedge 0\le g_{tl}\le 1}^{Y_{tl}^1(s^1)} \quad \overbrace{g_0^1=0\wedge g=g_0+1\wedge 0\le x_0^1\le 10\wedge y^1=x_0^1+1}^{Y_{hd}(e,s^2)_1}}{Y_{hd}(c^1,s^2)_1 \;=\; g=1\wedge 0\le i^1\le 10\wedge j^1=i^1+1}
$$

(call-site) (exit-site)

$g_{tl}=g_0^1\wedge i_{tl}^1=x_0^1$ (unification of input parameters)

$g'=g\wedge (i^1)'=i_{tl}^1\wedge (j^1)'=y^1$ (output parameter passing)

(return-site)

For $Y_{hd}(c^1, e^2)_1$, we also instantiate Eq. (9):

$$
\frac{\overbrace{0\le i_{tl}^1\le 10\wedge 0\le g_{tl}\le 1}^{Y_{tl}^1(s^1)} \quad \overbrace{0\le g_0^1\le 1\wedge g=2\wedge 0\le x_0^1\le 10\wedge y^1=x_0^1+1}^{Y_{hd}(e,e^2)_1}}{Y_{hd}(c^1,e^2)_1 \;\supseteq\; g=2\wedge 0\le i^1\le 10\wedge j^1=i^1+1}
$$

(call-site) (exit-site)

$g_{tl}=g_0^1\wedge i_{tl}^1=x_0^1$ (unification of input parameters)

$g'=g\wedge (i^1)'=i_{tl}^1\wedge (j^1)'=y^1$ (output parameter passing)

(return-site)

Taking the union with the image of $Y_{hd}(c^1, s^2)_1$, we obtain $Y_{hd}(c^1, e^2)_1 = g=2\wedge 0\le i^1\le 10\wedge j^1=i^1+1$.

The second iteration produces the following results:

$$
\begin{array}{rll}
Y_{hd}(s,s^2)_2 &= 0\le g_0^1\le 1 \wedge g=g_0^1 & \wedge\ 0\le x_0^1\le 10\\
Y_{hd}(e,s^2)_2 &= 0\le g_0^1\le 1 \wedge g=g_0+1 & \wedge\ 0\le x_0^1\le 10 \wedge y^1=x_0^1+1\\
Y_{hd}(s,e^2)_2 &= 0\le g_0^1\le 1 \wedge g=g_0^1+1 & \wedge\ 0\le x_0^1\le 10 \quad \text{(image of } Y_{hd}(s,s^2)_2\text{)}\\
&\vee\ g_0^1=1 \wedge g=g_0^1 & \wedge\ 0\le x_0^1\le 10 \quad \text{(image of } Y_{hd}(s^1,e^2)\text{)}\\
&\vee\ g_0^1=2 \wedge g=g_0^1 & \wedge\ 0\le x_0^1\le 10 \quad \text{(image of } Y_{hd}(c^1,e^2)_1\text{)}\\
&= g_0^1\le g\le g_0^1+1 \wedge 1\le g\le 2 & \wedge\ 0\le x_0^1\le 10\\
Y_{hd}(e,e^2)_2 &= g_0^1+1\le g\le g_0^1+2 \wedge 2\le g\le 3 & \wedge\ 0\le x_0^1\le 10 \wedge y^1=x_0^1+1
\end{array}
$$

For $Y_{hd}(c^1,s^2)_1$, we instantiate Eq. (9) as follows:

$$
\begin{array}{c}
\overbrace{0\le i_{tl}^1\le 10 \wedge 0\le g_{tl}\le 1}^{Y_{tl}^1(s^1)} \quad \overbrace{0\le g_0^1\le 1 \wedge g=g_0+1 \wedge 0\le x_0^1\le 10 \wedge y^1=x_0^1+1}^{Y_{hd}(e,s^2)_2}\\
\text{(call-site)} \qquad\qquad \text{(exit-site)}\\
g_{tl}=g_0^1 \wedge i_{tl}^1=x_0^1 \qquad g'=g \wedge (i^1)'=i_{tl}^1 \wedge (j^1)'=y^1\\
\text{(unification of input parameters)} \qquad \text{(output parameter passing)}\\
\hline
Y_{hd}(c^1,s^2)=Y_{hd}(c^1,s^2)_2 \;=\; 1\le g\le 2 \wedge 0\le i^1\le 10 \wedge j^1=i^1+1 \qquad \text{(return-site)}
\end{array}
$$

Similarly, $Y_{hd}(c^1,e^2)=Y_{hd}(c^1,e^2)_2 = 2\le g\le 3 \wedge 0\le i^1\le 10 \wedge j^1=i^1+1$.

Observe that $Y_{hd}(c^1,s^2)$ is an overapproximation of the exact result, for which $g=1$ (procedure succ has been called once, and thread T2 has not moved yet). The explanation is as follows:

- Observe first that in $Y(s,s^2)$ and thus in $Y(e,s^2)$ we have $0\le g_0^1\le 1$: This is because we have merged at the start point of succ the possible values of effective parameters at the two call sites (dashed line in Fig. 11). These two predicates are, however, still exact.
- Now $Y(e,s^2)$, which defines the summary function of succ when T2 is located at point s^2, is combined with $Y_{tl}^1(s^1)$, which does not contain any information about T2. In particular, $Y_{tl}^1(s^1)$ includes the case where T2 is located at e^2 and $g=1$.

Hence the approximation corresponds precisely to the choice made in Sect. 4: when a thread performs a procedure call, we do not save in the variables memorizing the call context of the procedure the local state of the other threads, and in particular their *pc* variable.

Another related observation is that the value of $Y_{hd}(e,e^2)_2$ satisfies only $g_0^1+1\le g\le g_0^1+2$. This corresponds to the join of two different situations: the partial summary $Y_{hd}(e,e^2)_2$ does not record whether

1. thread T2 was already located at e^2 upon procedure call (in which case g has been incremented only once during the execution of succ), or if
2. it was located at point s^2 and a context switch has occurred during the execution of succ, so that g has been incremented as second time by T2 during the execution of succ.

The third and last iteration gives:

$$
\begin{array}{rll}
Y_{hd}(s,s^2)=Y_{hd}(s,s^2)_3 &= 0\le g_0^1\le 2 \wedge g=g_0^1 & \wedge\ 0\le x_0^1\le 10\\
Y_{hd}(e,s^2)=Y_{hd}(e,s^2)_3 &= 0\le g_0^1\le 2 \wedge g=g_0+1 & \wedge\ 0\le x_0^1\le 10 \wedge y^1=x_0^1+1\\
Y_{hd}(s,e^2)=Y_{hd}(s,e^2)_3 &= 0\le g_0^1\le 2 \wedge g=g_0^1+1 & \wedge\ 0\le x_0^1\le 10\\
& & \quad \text{(image of } Y_{hd}(s,s^2)\text{)}\\
&\vee\ g_0^1=1 \wedge g=g_0^1 & \wedge\ 0\le x_0^1\le 10\\
& & \quad \text{(image of } Y_{hd}(s^1,e^2)\text{)}\\
&\vee\ 2\le g_0^1\le 3 \wedge g=g_0^1 & \wedge\ 0\le x_0^1\le 10\\
& & \quad \text{(image of } Y_{hd}(c^1,e^2)\text{)}\\
&= g_0^1\le g\le g_0^1+1 \wedge 1\le g\le 3 & \wedge\ 0\le x_0^1\le 10\\
Y_{hd}(e,e^2)=Y_{hd}(e,e^2)_3 &= g_0^1+1\le g\le g_0^1+2 \wedge 2\le g\le 4 & \wedge\ 0\le x_0^1\le 10 \wedge y^1=x_0^1+1\\
Y_t l^1(c^1) &= 1\le g\le 3 & \wedge\ 0\le i^1\le 10 \wedge j^1=i^1+1
\end{array}
$$

At last, by applying rule (9) at call site c^1 of thread T1, we obtain

$$Y_{hd}(e^1, s^2) = 2 \le g \le 3 \wedge 0 \le i^1 \le 10 \wedge j^1 = i^1 + 1$$
$$Y_{hd}(e^1, e^2) = 2 \le g \le 4 \wedge 0 \le i^1 \le 10 \wedge j^1 = i^1 + 1$$

The invariant $Y_{hd}(e^1, e^2)$ is to be compared with the exact result, in which $g = 3$. Observe however that the value of j is exact, because in this example the local variables (including the pc variables) do not interact with the global variable g. Hence, the approximation performed on g is not propagated.

This example illustrates the kind of approximation our forward analysis may perform. We will show in Sect. 8 that it is still able to analyze with enough precision subtle synchronization protocols. Another concern is the soundness of this analysis. In order to prove it, we formalize it by abstract interpretation of the instrumented semantics, by generalizing the Galois connection of Eq. (7).

5.3 Formalization as a concurrent stack abstraction

We prove now the soundness of the analysis method defined by Eqs. (9)–(10) using an abstract interpretation approach.

We use the following functions on stacks: for any stack $\Gamma = \epsilon_0 \dots \epsilon_n \in Env^+$ and set of stacks $X \in \wp(Env^+)$:

$$\begin{array}{ll} hd(\Gamma) = \{\epsilon_n\} & hd(X) = \bigcup_{\Gamma \in X} \\ tl(\Gamma) = \{\epsilon_i \mid 0 \le i < n\} & tl(X) = \bigcup_{\Gamma \in X} tl(\Gamma) \\ elts(\Gamma) = hd(\Gamma) \cup tl(\Gamma) & elts(X) = \bigcup_{\Gamma \in X} elts(\Gamma) \end{array}$$

Starting from the instrumented semantics of Sect. 4, we define the Galois connection

$$\wp(S_f) = \wp(Env^+ \times Env^+) \underset{\alpha_c}{\overset{\gamma_c}{\leftrightarrows}} A_c = \wp(Env \times Env) \times \wp(Env) \times \wp(Env) \tag{11}$$

$$\alpha_c\Big(\Big\{\big(\overbrace{\langle\epsilon_0^1 \dots \epsilon_{n_1}^1}^{\Gamma^1}, \overbrace{\epsilon_0^2 \dots \epsilon_{n_2}^2\rangle}^{\Gamma^2}\big)\Big\}\Big) = \left\langle \begin{array}{c} hd(\Gamma^1, \Gamma^2), \\ tl(\Gamma^1), \\ tl(\Gamma^2) \end{array} \right\rangle = \left\langle \begin{array}{c} \big\{\langle \epsilon_{n_1}^1, \epsilon_{n_2}^2 \rangle\big\}, \\ \big\{\epsilon_{i_1}^1 \mid 0 \le i_1 < n_1\big\}, \\ \big\{\epsilon_{i_2}^2 \mid 0 \le i_2 < n_2\big\} \end{array} \right\rangle$$

$$\alpha_c(X) = \bigsqcup_{\langle \Gamma^1, \Gamma^2 \rangle \in X} \alpha_c(\langle \Gamma^1, \Gamma^2 \rangle)$$

$$\gamma_c\big(\langle Y_{hd}, Y_{tl}^1, Y_{tl}^2 \rangle\big) = \left\{ \overbrace{\langle \epsilon_0^1 \dots \epsilon_{n_1}^1}^{\Gamma^1}, \overbrace{\epsilon_0^2 \dots \epsilon_{n_2}^2 \rangle}^{\Gamma^2} \;\middle|\; \begin{array}{l} \langle \epsilon_{n_1}^1, \epsilon_{n_2}^2 \rangle \in Y_{hd} \\ \forall 0 \le i_1 < n_1 \;:\; \epsilon_{i_1}^1 \in Y_{tl}^1 \\ \forall 0 \le i_2 < n_2 \;:\; \epsilon_{i_2}^2 \in Y_{tl}^2 \\ \langle \Gamma^1, \Gamma^2 \rangle \\ \quad \text{is a well-formed state} \end{array} \right\}$$

The indices hd and tl we used in the previous sections are now more understandable: they represent resp. products of stacks heads and stack tails.

As an example, consider the set X of two reachable states of the prog. of Fig. 11 previously analyzed:

$$X = \left\{ \left\langle \begin{array}{l} (pc^1 = s^1 \wedge g^1 = 1 \wedge i^1 = 0) \cdot \\ \quad (pc^1 = e \wedge g_0^1 = 1 \wedge x_0^1 = 0 \wedge y^1 = 1 \wedge g^1 = 2), \\ \quad (pc^2 = e^2 \wedge g^2 = 2) \end{array} \right\rangle, \left\langle \begin{array}{l} (pc^1 = c^1 \wedge g^1 = 1 \wedge i^1 = 0) \cdot \\ \quad (pc^1 = e \wedge g_0^1 = 1 \wedge x_0^1 = 0 \wedge y^1 = 1 \wedge g^1 = 3), \\ \quad (pc^2 = e^2 \wedge g^2 = 3) \end{array} \right\rangle \right\}$$

The first state is reachable by an execution where thread 2 moves before the first call to succ, and the second by an execution where thread 2 moves during the second call to succ. We compute $\gamma \circ \alpha(X)$, which captures the loss of information performed by the abstraction of X.

$$\alpha(X) = \left\langle \left\{ \Big(pc^1 = e \wedge pc^2 = e^2 \wedge g_0^1 = 1 \wedge x_0^1 = 0 \wedge y^1 = 1 \wedge 2 \le g^1 = g^2 \le 3\Big) \right\}, \left\{ \Big(pc^1 \in \{s^1, c^1\} \wedge g^1 = 1 \wedge i^1 = 0\Big) \right\}, \emptyset \right\rangle$$

$$\gamma \circ \alpha(X) = \left\{ \left\langle \begin{array}{l} \underline{(pc^1 \in \{s^1, c^1\}} \wedge g^1 = 1 \wedge i^1 = 0) \cdot \\ \quad (pc^1 = e \wedge g_0^1 = 1 \wedge x_0^1 = 0 \wedge y^1 = 1 \wedge \underline{2 \le g^1 \le 3}), \\ \quad (pc^2 = e^2 \wedge g^2 = g^1) \end{array} \right\rangle \right\}$$

Compared to X that contains two states, $\gamma \circ \alpha(X)$ contains two additional states, in particular one which is not reachable: $\langle (pc^1 = s^1 \dots) \cdot (pc^1 = e \dots g^1 = 3), (pc^2 = e^2 \wedge g^2 = 3) \rangle$. This approximation will be propagated by the procedure return operation to the pair of control point (c^1, e^2), where g may be equal to 3 in the abstract semantics, unlike in the concrete one (see the complete analysis in Sect. 5.2).

The abstract domain A_c defines an abstract semantics that is simpler than the concrete semantics (the control is now finite) and that can be seen as an analysis method that can be further combined with a data abstraction as shown in Sect. 7. The abstraction function α is never applied, but allows to relate the abstract transfer functions to the concrete ones.

Figure 12 defines the abstract postcondition operator $apost_c^1 : A_c \to A_c$ induced by the concrete postcondition $post^1 : S_f \to S_f$, which just reformulates Eqs. (9), (10) with a set-theoretic notation. We detail only the steps performed by thread 1.

The case of intraprocedural instruction (Eq. (12)) is simple: The top environment of thread 1 is modified according to the relation R, and the top environment of thread 2 is modified to reflect the new values of global variables. The sets of tail environments are not modified. For procedure call, Eq. (13), the new top environment of thread 1 is initialized using the relation R^{f+} defined in Fig. 7. The set of tail activation records of thread 1 is extended by projecting the former top environment on thread 1. The case of procedure return, Eq. (14) has already been discussed in Sect. 5.2. We select a global top environment in $\langle \epsilon, \epsilon^2 \rangle \in Y_{hd}$ and a tail environment $\epsilon_{tl} \in Y_{tl}^1$ for thread 1, so that actual parameters in ϵ_{tl}

Fig. 12 Abstract postcondition $apost^1_c : A_c \to A_c$, using the relations $R^f_\bullet$ defined in Fig. 7

$$apost^1_c(c \xrightarrow{\langle R\rangle} c')(Y_{hd}, Y^1_{tl}, Y^2_{tl}) = (Z_{hd}, Z^1_{tl}, Z^2_{tl}) \text{ with}$$
$$Z_{hd} = \left\{ \langle \epsilon', (\epsilon^2)' \rangle \;\middle|\; \begin{array}{c} \langle \epsilon,\ \epsilon^2 \rangle \in Y_{hd} \\ R^f(c,c')(\epsilon,\epsilon') \\ (\epsilon^2)' = \epsilon^2[\mathbf{g} \mapsto \epsilon'(\mathbf{g})] \end{array} \right\} \quad \begin{array}{l} Z^1_{tl} = Y^1_{tl} \\ Z^2_{tl} = Y^2_{tl} \end{array} \tag{12}$$

$$apost^1_c(c \xrightarrow{\langle \texttt{call}\ \mathbf{y}:=P_j(\mathbf{x})\rangle} \mathtt{s}_j)(Y_{hd}, Y^1_{tl}, Y^2_{tl}) = (Z_{hd}, Z^1_{tl}, Z^2_{tl}) \text{ with}$$
$$Z_{hd} = \left\{ \langle \epsilon', \epsilon^2 \rangle \;\middle|\; \begin{array}{c} \langle \epsilon, \epsilon^2 \rangle \in Y_{hd} \\ R^{f+}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon,\epsilon') \end{array} \right\} \quad \begin{array}{l} Z^1_{tl} = Y^1_{tl} \cup \{\epsilon | \langle \epsilon, \epsilon^2 \rangle \in Y_{hd}\} \\ Z^2_{tl} = Y^2_{tl} \end{array} \tag{13}$$

$$apost^1_c(\mathtt{e}_j \xrightarrow{\langle \texttt{ret}\ \mathbf{y}:=P_j(\mathbf{x})\rangle} ret(c))(Y_{hd}, Y^1_{tl}, Y^2_{tl}) = (Z_{hd}, Z^1_{tl}, Z^2_{tl}) \text{ with}$$
$$Z_{hd} = \left\{ \langle \epsilon', \epsilon^2 \rangle \;\middle|\; \begin{array}{c} \langle \epsilon, \epsilon^2 \rangle \in Y_{hd} \wedge \epsilon_{tl} \in Y^1_{tl} \\ \epsilon_{tl}(\mathbf{g},\mathbf{x}) = \epsilon(\mathbf{g}_0, \mathbf{fp}_0) \\ R^{f-}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon_{tl},\epsilon,\epsilon') \end{array} \right\} \quad \begin{array}{l} Z^1_{tl} = Y^1_{tl} \\ Z^2_{tl} = Y^2_{tl} \end{array} \tag{14}$$

match frozen copy of formal parameters in ϵ. The new top environment is then obtained using the relation R^{f-} defined in Fig. 7.

Proposition 2 ($apost^1_c$ is a correct approximation of $post^1$) *For any set $X \subseteq S_f$ of well-formed states, $apost^1_c \circ \alpha_c(X) \sqsupseteq \alpha_c \circ post^1(X)$. More precisely*:

(i) *if τ is an intraprocedural or a call instruction, $apost^1_c(\tau) \circ \alpha_c(X) = \alpha_c \circ post^1(\tau)(X)$;*
(ii) *if τ is a return instruction, $apost^1_c(\tau) \circ \alpha_c(X) \sqsupseteq \alpha_c \circ post^1(\tau)(X)$*

$apost^2_c$ is defined in a symmetric way and is similarly a correct approximation of $post^2$ ($apost$ is decomposed as $post$ is in Eq. (1)). As a corollary, for any set $S_0 \in S_f$ of well-formed states, $lfp(F_c[S_0]) \sqsupseteq \alpha_c(lfp(F[S_0]))$, where $F_c[S_0](Y) = \alpha_c(S_0) \sqcup apost(Y)$ is the abstract transfer function and $F[S_0]$ has been defined in Sect. 2.3.

Not surprisingly, the abstract semantics is less precise for return instructions, because they implicitly need to rebuild the stacks. The example of Sect. 5.2 illustrated this point.

Proof First, observe that for any τ, the function $\alpha_c \circ post(\tau) : \wp(S_f) \to A_c$ is distributive, as a composition of distributive functions.

1. Let $\tau = c \xrightarrow{\langle R\rangle} c'$. Observe that the function $apost_c(\tau)$ defined by Eq. (12) is distributive, hence $apost_c(\tau) \circ \alpha_c$ is distributive. We can thus compare the two distributive functions $apost_c(\tau) \circ \alpha_c$ and $\alpha_c \circ post(\tau)$ on singleton sets.
From Eq. (IntraF) of Fig. 7, we have

$$\begin{aligned} &\alpha_c \circ post(\tau)\left(\left\{\left\langle \Gamma^1 \cdot \epsilon,\ \Gamma^2 \cdot \epsilon^2 \right\rangle\right\}\right) \\ &= \alpha_c\left(\left\{\left\langle \Gamma^1 \cdot \epsilon',\ \Gamma^2 \cdot (\epsilon^2)' \right\rangle \;\middle|\; \begin{array}{l} R^f(c,c')(\epsilon,\epsilon') \wedge \\ (\epsilon^2)' = \epsilon^2[\mathbf{g} \mapsto \epsilon'(\mathbf{g})] \end{array}\right\}\right) \\ &= \left\langle \left\{ \langle \epsilon', (\epsilon^2)' \rangle \;\middle|\; \begin{array}{l} R^f(c,c')(\epsilon,\epsilon') \wedge \\ (\epsilon^2)' = \epsilon^2[\mathbf{g} \mapsto \epsilon'(\mathbf{g})] \end{array} \right\},\ \begin{array}{c} \\ elts(\Gamma^1),\ elts(\Gamma^2) \end{array} \right\rangle \end{aligned}$$

On the other hand, from Eq. (12),

$$\begin{aligned} &apost_c(\tau) \circ \alpha_c\left(\left\{\left\langle \Gamma^1 \cdot \epsilon,\ \Gamma^2 \cdot \epsilon^2 \right\rangle\right\}\right) \\ &= apost_c(\tau)\left(\left\{\left\langle \left\{ \langle \epsilon, \epsilon^2 \rangle \right\},\ elts(\Gamma^1),\ elts(\Gamma^2) \right\rangle\right\}\right) \\ &= \left\langle \left\{ \langle \epsilon', (\epsilon^2)' \rangle \;\middle|\; \begin{array}{l} R^f(c,c')(\epsilon,\epsilon') \wedge \\ (\epsilon^2)' = \epsilon^2[\mathbf{g} \mapsto \epsilon'(\mathbf{g})] \end{array} \right\},\ \begin{array}{c} \\ elts(\Gamma^1),\ elts(\Gamma^2) \end{array} \right\rangle \end{aligned}$$

The two expressions are equal.

2. Let $\tau = c \xrightarrow{\texttt{call}\ \mathbf{y}:=P_j(\mathbf{x})} \mathtt{s}_j$. Observe that the function $apost_c(\tau)$ defined by Eq. (12) is distributive, hence $apost_c(\tau) \circ \alpha_c$ is distributive. We can thus compare the two distributive functions $apost_c(\tau) \circ \alpha_c$ and $\alpha_c \circ post(\tau)$ on singleton sets.
From Eq. (IntraF) of Fig. 7, we have

$$\begin{aligned} &\alpha_c \circ post(\tau)\left(\left\{\left\langle \Gamma^1 \cdot \epsilon,\ \Gamma^2 \cdot \epsilon^2 \right\rangle\right\}\right) \\ &= \alpha_c\left(\left\{\left\langle \Gamma^1 \cdot \epsilon \cdot \epsilon',\ \Gamma^2 \cdot \langle c^2, \epsilon^2 \rangle \right\rangle \mid R^{f+}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon,\epsilon')\right\}\right) \\ &= \left\langle \left\{ \langle \epsilon', \epsilon^2 \rangle \mid R^{f+}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon,\epsilon') \right\},\ elts(\Gamma^1) \cup \{\epsilon\},\ elts(\Gamma^2) \right\rangle \end{aligned}$$

On the other hand, from Eq. (13),

$$\begin{aligned} &apost_c(\tau) \circ \alpha_c\left(\left\{\left\langle \Gamma^1 \cdot \epsilon,\ \Gamma^2 \cdot \epsilon^2 \right\rangle\right\}\right) \\ &= apost_c(\tau)\left(\left\{\left\langle \{\langle \epsilon, \epsilon^2 \rangle\},\ elts(\Gamma^1),\ elts(\Gamma^2) \right\rangle\right\}\right) \\ &= \left\langle \left\{ \langle \epsilon', \epsilon^2 \rangle \mid R^{f+}_{\mathbf{y}:=P_j(\mathbf{x})}(c,c')(\epsilon,\epsilon') \right\},\ elts(\Gamma^1) \cup \{\langle c, \epsilon \rangle\},\ elts(\Gamma^2) \right\rangle \end{aligned}$$

The two expressions are equal.

3. Let $\tau = \mathsf{e}_j \xrightarrow{\texttt{ret}\ \mathbf{y}:=P_j(\mathbf{x})} ret(c)$. Here $apost_{hd}(\tau)$, hence $apost(\tau)$, is not distributive, unlike $\alpha_c \circ post(\tau)$. Thus we cannot obtain an equality, but just a soundness inclusion. From Eq. (RetF) of Fig. 7 we have for Y a set of well-formed states:

$$post(\tau)(Y) = \left\{ \left\langle \Gamma^1 \cdot \epsilon', \Gamma^2 \cdot \epsilon^2 \right\rangle \;\middle|\; \begin{array}{l} \left\langle \Gamma^1 \cdot \epsilon_{tl} \cdot \epsilon,\ \Gamma^2 \cdot \epsilon^2 \right\rangle \in Y \\ R^{f-}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon_{tl}, \epsilon, \epsilon') \end{array} \right\}$$

Thus, exploiting the well-formedness of stacks in Y and decomposing α_c as $\langle \alpha_{hd}, \alpha^1_{tl}, \alpha^2_{tl} \rangle$,

$$\alpha_{hd} \circ post(\tau)(Y) = \left\{ \langle \epsilon', \epsilon^2 \rangle \;\middle|\; \begin{array}{l} \left\langle \Gamma^1 \cdot \epsilon_{tl} \cdot \epsilon,\ \Gamma^2 \cdot \epsilon^2 \right\rangle \in Y \\ \epsilon_{tl}(\mathbf{g}, \mathbf{x}) = \epsilon(\mathbf{g}_0, \mathbf{fp}_0) \\ R^{f-}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon_{tl}, \epsilon, \epsilon') \end{array} \right\}$$

$$\alpha^1_{tl} \circ post(\tau)(Y) = \bigcup \left\{ elts(\Gamma^1) \mid \langle \Gamma^1 \cdot \epsilon_{tl} \cdot \epsilon,\ \Gamma^2 \rangle \in Y \right\}$$

$$\alpha^2_{tl} \circ post(\tau)(Y) = \alpha^2_{tl}(Y)$$

On the other hand, from Eq. (14):

$$apost_{hd}(\tau) \circ \alpha_c(Y) = apost_{hd}(\tau)(\langle Y_{hd}, Y^1_{tl}, Y^2_{tl} \rangle)$$

$$= \left\{ \langle \epsilon', \epsilon^2 \rangle \;\middle|\; \begin{array}{l} \langle \epsilon, \epsilon^2 \rangle \in Y_{hd} \ \wedge\ \epsilon_{tl} \in Y^1_{tl} \\ \epsilon_{tl}(\mathbf{g}, \mathbf{x}) = \epsilon(\mathbf{g}_0, \mathbf{fp}_0) \\ R^{f-}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon_{tl}, \epsilon, \epsilon') \end{array} \right\}$$

$$= \left\{ \langle \epsilon', \epsilon^2 \rangle \;\middle|\; \begin{array}{c} \langle \Gamma^1 \cdot \epsilon,\ \Gamma^2 \cdot \epsilon^2 \rangle \in Y \ \wedge\ \langle \Omega^1_0 \cdot \epsilon_{tl} \cdot \Omega^1_1,\ \Omega^2 \rangle \in Y \\ \epsilon(\mathbf{g}, \mathbf{x}) = \epsilon(\mathbf{g}_0, \mathbf{fp}_0) \\ R^{f-}_{\mathbf{y}:=P_j(\mathbf{x})}(\epsilon_{tl}, \epsilon, \epsilon') \end{array} \right\}$$

$$\sqsupseteq \alpha_{hd} \circ post(\tau)(Y)$$

$$apost^1_{tl}(\tau) \circ \alpha_c(Y) = \alpha^1_{tl}(Y) \sqsupseteq \alpha^1_{tl} \circ post(\tau)(Y)$$

$$apost^2_{tl}(\tau) \circ \alpha_c(Y) = \alpha^2_{tl}(Y) = \alpha^2_{tl} \circ post(\tau)(Y)$$

□

Completeness results The first trivial result we have is that in the case of multithreaded programs without procedure calls, our technique is exact, as the abstraction function becomes the identity. Observing now that the stack abstraction falls back to the *functional* abstraction defined in [26] for single-thread program, we inherit from the following theorem:

Theorem 1 ([26]) *For single-thread programs, and for initial sets of states X_0 composed only of one-element stacks, the abstract reachability analysis is optimal: $a reach_c(X_0) = \alpha_c(reach(X_0))$*

This implies that the set of *top* environments of reachable stacks is computed exactly, so that the invariants inferred at each control point are the exact ones.

6 Backward analysis

6.1 Instrumenting the standard semantics for backward analysis

Whereas forward analysis aims at inferring invariants, backward analysis aims at inferring necessary conditions to reach a final, typically erroneous configuration. Figure 15 gives an example of backward analysis from control point (e), which shows for instance that $x = 0$ at point (s') is a necessary condition to reach (e).

The key ideas of our forward analysis is located in the instrumentation of the standard semantics described in Sect. 4, which puts the global store in the stacks and augments resulting environments with information about the call context of procedures, and in the concurrent stack abstraction of Sect. 5.3. The same principle can be applied to backward analysis. However, the duality between the two analyses is not total:

1. Assignments and substitutions of a variable by an expression are inverse of each other only when the assignment is invertible (like `x=2x+1` and unlike `x=y`). The assignments involved in the (forward) semantics of procedure returns are *not* invertible.
2. In the forward semantics, a new element is pushed on the stack when executing a procedure call; in this case the former top element remains unchanged (see Eq. (Call) of Fig. 6). In the backward semantics, the dual operation is the procedure return. However, when a procedure return is executed in a backward way, not only is a new element pushed on the stack, but one has to consider the predecessor of the former top element, for which the previous point (1) applies.

This explains that the instrumentation for backward analysis is only partially symmetric to the instrumentation for forward analysis. In this instrumented semantics:

- Global variables are pushed on stacks as in Sect. 4, so that the state-space becomes $S_b = Env^+ \times Env^+$.
- Environments are defined on variables $\mathbf{g}, \mathbf{l}, \mathbf{g}_1, \mathbf{fr}_1$, where the values of the auxiliary variables $\mathbf{g}_1, \mathbf{fr}_1$ keep track of the "return" context at exit point of the current procedure.
- Environments ϵ associated with a call site with the associated instruction $\mathbf{y} := P_j(\mathbf{x})$ define in addition the values of auxiliary variables $\mathbf{g}_t, \mathbf{y}_t$, that store the values of $\mathbf{g}$ and $\mathbf{y}$ at the return site, which are made undefined by the relation $R^{-}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon, \epsilon_j,\ \epsilon')$ defined in Fig. 7 when taken backward.

The semantic rules are given on Fig. 13. $s \leftarrow s'$ means that s is a predecessor (in the standard semantics) of s'.

Adding backward instrumentation variables

$$R^b(c,c')(\epsilon,\epsilon') = \begin{cases} R(c,c')(\epsilon,\epsilon') \\ \epsilon(\mathbf{g}_1,\mathbf{fr}_1) = \epsilon'(\mathbf{g}_1,\mathbf{fr}_1) \end{cases} \quad (\mathrm{R^b})$$

$$R^{b+}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon,\ \epsilon',\epsilon'_j) = \begin{cases} R^{+}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon',\epsilon'_j) \\ \epsilon = \epsilon' \ominus \{\mathbf{g}_t,\mathbf{y}_t\} \end{cases} \quad (\mathrm{R^{b+}})$$

$$R^{b-}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon,\epsilon_j,\ \epsilon') = \begin{cases} R^{-}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon,\epsilon_j,\ \epsilon') \\ \epsilon(\mathbf{g}_t,\mathbf{y}_t,\mathbf{g}_1,\mathbf{fr}_1) = \epsilon'(\mathbf{g},\mathbf{y},\mathbf{g}_1,\mathbf{fr}_1) \\ \epsilon_j(\mathbf{g}_1,\mathbf{fr}_1) \quad\quad = \epsilon'(\mathbf{g},\mathbf{y}) \end{cases} \quad (\mathrm{R^{b-}})$$

Semantic rules

$$\frac{I(c,c') = \langle R\rangle \quad R^b(c,c')(\epsilon,\epsilon')}{\Gamma\cdot\epsilon \leftarrow^t_b \Gamma\cdot\epsilon'} \quad (\text{IntraB})$$

$$\frac{I(c,\mathtt{s}_j) = \langle \mathtt{call}\ \mathbf{y} := P_j(\mathbf{x})\rangle \quad R^{b+}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon,\ \epsilon',\epsilon'_j)}{\Gamma\cdot\epsilon \leftarrow^t_b \Gamma\cdot\epsilon'\cdot\epsilon'_j} \quad (\text{CallB})$$

$$\frac{I(\mathtt{e}_j, ret(c)) = \langle \mathtt{ret}\ \mathbf{y} := P_j(\mathbf{x})\rangle \quad R^{b-}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon,\epsilon_j,\ \epsilon')}{\Gamma\cdot\epsilon\cdot\epsilon_j \leftarrow^t_b \Gamma\cdot\epsilon'} \quad (\text{RetB})$$

$$\frac{\Gamma^1 \leftarrow^1_b \Upsilon^1 \qquad \Upsilon^1 = \Upsilon\cdot\varepsilon^1 \qquad \epsilon^2 = \varepsilon^2[\mathbf{g}\mapsto\epsilon^1(\mathbf{g})]}{\langle\Gamma^1,\ \Gamma^2\cdot\epsilon^2\rangle \leftarrow_b \langle\Upsilon^1,\ \Gamma^2\cdot\varepsilon^2\rangle} \quad (\text{Conc1B})$$

$$\frac{\Gamma^2 \leftarrow^2_b \Upsilon^2 \qquad \Upsilon^2 = \Upsilon\cdot\varepsilon^2 \qquad \epsilon^1 = \varepsilon^1[\mathbf{g}\mapsto\epsilon^2(\mathbf{g})]}{\langle\Gamma^1\cdot\epsilon^1,\ \Gamma^2\rangle \leftarrow_b \langle\Gamma^1\cdot\varepsilon^1,\ \Upsilon^2\rangle} \quad (\text{Conc2B})$$

Fig. 13 Instrumented semantics for backward analysis: transition relation $\leftarrow^t_b$ of the thread T^t and transition relation $\leftarrow_b$ of the full program, using the relations $R_\bullet$ defined in Fig. 7

For procedure calls, as mentioned in the point (2) above, the rules (CallB)–($\mathrm{R^{b+}}$) consist basically in popping the top environment from the stack, whereas the rules (RetF)–($\mathrm{R^{f-}}$) for procedure returns in the instrumented forward semantics of Fig. 7 is more complex. Notice also that equality between formal and actual input parameters is checked only at this point by the relation $R^{+}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon',\epsilon'_j)$, which acts as a filter. For instance in Fig. 15, at point (e') of procedure `succ`, the value of formal input parameter `x` is unknown, although the corresponding actual parameter `i` satisfies $i \geq 0$ at the two return sites.[4]

For procedure returns, notice that in rules (RetB)–($\mathrm{R^{b-}}$) the values of $\mathbf{g}$ and $\mathbf{y}$ is undefined in ϵ, because in the standard forward semantics these values are defined in ϵ' by non-invertible assignments; see rule ($\mathrm{R^-}$) of Fig. 6.

In this semantics, coreachable call stacks are necessarily *backward well formed* in the following sense.

Definition 2 (*Backward well-formed stacks and states*) A stack $\Gamma = \epsilon_0 \ldots \epsilon_n \in Env^+$ is *backward well formed*, if for any $i < n$:

(i) $c_i = \epsilon_i(pc)$ is a call site for the procedure P_j, with $j = proc(c_{i+1}), c_{i+1} = \epsilon_{i+1}(pc)$ and $I(c_i, \mathtt{s}_j) = \langle \mathtt{call}\ \mathbf{y} := P_j(\mathbf{x})\rangle$;

(ii) equality between copies of actual and formal output parameters holds: $\epsilon_i(\mathbf{g}_t, \mathbf{y}_t) = \epsilon_{i+1}(\mathbf{g}_1, \mathbf{fr}_1^j)$.

A state $\langle\Gamma^1, \Gamma^2\rangle \in S_b$ is *backward well formed* if Γ^1 and Γ^2 are well formed, and if top environments agree on the current value of global variables.

Proposition 3 *If $s' \in S_b$ is a well-formed state, then any $s \in S_f$ such that $s \leftarrow^*_b s'$ is a backward well-formed state.*

Proof It is similar to the proof of Proposition 1. (i) is trivial, (ii) follows from the fact that the only rule defining $\mathbf{g}_t, \mathbf{y}_t$ in the "tail" environment and $\mathbf{g}_1, \mathbf{fr}_1$ in the top environment is the rule (RetB), relation ($\mathrm{R^{b-}}$). □

Final states with stacks of height 1 are always backward well formed. However, if the set of final states is specified as an invariant on the top environments in some callee procedure, one has to rebuild all the tail environments that makes the resulting stacks backward well formed.

6.2 Backward analysis

In this section, we define a method for coreachability analysis for concurrent and recursive programs, following the same ideas as those for the forward analysis.

Axiomatic presentation We define in Fig. 14 our backward analysis in an axiomatic way as in Sect. 5.2, using the three same predicates $Y_{hd}(\epsilon^1, \epsilon^2)$, $Y^1_{tl}(\epsilon^1)$ and $Y^2_{tl}(\epsilon^2)$, which now

[4] If one detects that `x` is not modified in `succ`, one could match the effective parameter `i` at return site with the formal parameter `x` at exit site.

Fig. 14 Backward analysis based on the instrumented semantics of Fig. 13

$$\frac{\begin{array}{c} I(c,\mathtt{s}_j) = \langle \mathtt{call}\ \mathbf{y} := P_j(\mathbf{x})\rangle \\ \begin{array}{ccc} Y^1_{tl}(\epsilon'_{tl}) & \boxed{Y_{hd}(\epsilon',\epsilon^2)} & \epsilon'_{tl}(\mathbf{g}_t,\mathbf{y}_t) = \epsilon'(\mathbf{g}_1,\mathbf{fr}_1) \\ \text{(call-site)} & \text{(start point)} & \text{(output parameters, Prop. 3)} \end{array} \\ R^{b+}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon,\ \epsilon'_{tl},\epsilon') \\ \text{(matches } \epsilon'_{tl} \text{ and } \epsilon' \text{ on input parameters \& forgets } \mathbf{g}_t,\mathbf{y}_t) \end{array}}{\boxed{Y_{hd}(\epsilon,\epsilon^2)} \quad \text{(call-site)}} \tag{15}$$

$$\frac{I(\mathtt{e}_j, ret(c)) = \langle \mathtt{ret}\ \mathbf{y} := P_j(\mathbf{x})\rangle \quad \boxed{Y_{hd}(\epsilon',\epsilon^2)} \quad R^{-}_{\mathbf{y}:=P_j(\mathbf{x})}(c)(\epsilon_{tl},\epsilon,\ \epsilon')}{\boxed{Y_{tl}(\epsilon_{tl})} \quad \boxed{Y_{hd}(\epsilon,\epsilon^2)}} \tag{16}$$

$$\frac{I(c,c') = \langle R\rangle \quad \boxed{Y_{hd}\ (\epsilon',(\epsilon^2)')} \quad R(c,c')(\epsilon,\epsilon') \quad \epsilon^2 = (\epsilon^2)'[\mathbf{g} \mapsto \epsilon(\mathbf{g})]}{\boxed{Y_{hd}(\epsilon,\epsilon^2)}} \tag{17}$$

Fig. 15 Backward interprocedural analysis of a sequential program, starting from the exit point e

```
   thread T: var i,j:int;
   begin                                  proc succ(x:int) returns (y:int)
(c) ⊥                                     begin
    j=succ(i); j=0                   (s') y1=1 ∧ x=0
    i=j;       i=0                        if (x<0) then halt; endif;
    j=succ(i); i≥0 ∧ j=1                  0≤y1≤1 ∧ x=y1 − 1
    if j!=1 or i<0 then halt; endif;      y=x+1;
(e) ⊤                                (e') 0≤y1≤1 ∧ y=y1
   end                                    end
```

The coreachable value at exit point e' is the union of the result of the two return-to-exit edges induced by the two calls to procedure succ.

represent resp. coreachable stack tops and coreachable stack tails.

The main difference is that in backward analysis, a stack $\Gamma\cdot\epsilon\cdot\epsilon'$ can be coreachable while $\Gamma\cdot\epsilon$ is *not* coreachable. This is the case in the program of Fig. 15, in which the set of stacks $(pc = c \wedge j_t = 0)\cdot(pc = e' \wedge y_1 = 1 \wedge y = y_t)$ is coreachable from control point (e), but no stack of height 1 satisfying $pc = c$ is coreachable. This motivates the clear separation between stacks tails (predicates Y^t_{tl}) and stack tops (predicate Y_{hd}) in the stack abstraction, whereas in the case of forward analysis it can be derived by projection of stack tops.

Formalization as a concurrent stack abstraction We can formalize the backward analysis using the Galois connection defined in Sect. 5.3 with Eq. (5.3), by substituting

(i) S_f by S_b;
(ii) the notion of well-formed state of Definition 1 by the notion of backward well-formed states of Definition 3.

The abstract precondition operator $apre_c : A_c \to A_c$ induced by the concrete precondition

$$pre : S_b \to S_b \qquad X \mapsto \left\{ s \mid \exists s' \left\{ \begin{array}{l} s \leftarrow_b s' \wedge s' \in X \\ s \text{ and } s' \text{ are backward well formed} \end{array} \right. \right\}$$

is defined by reformulating the rules of Fig. 14 with a set-theoretic notation, as we did from the Eqs. (9), (10) to the definitions of Fig. 12. We have the following result.

Proposition 4 ($apre_c$ is a correct approximation of pre) *For any set $X \subseteq S_b$ of well-formed states, $apre_c \circ \alpha(X) \sqsupseteq \alpha_c \circ post(X)$. More precisely:*

(i) *if τ is an intraprocedural or a return instruction, $apost_c(\tau)\circ\alpha_c(X) = \alpha_c \circ post(\tau)(X)$;*
(ii) *if τ is a call instruction, $apost_c(\tau)\circ\alpha_c(X) \sqsupseteq \alpha_c \circ post(\tau)(X)$*

As a corollary, for any set $S_1 \in S_b$ of well-formed states, $lfp(G_c[S_1]) \sqsupseteq \alpha_c(lfp(G[S_1]))$, where $G_c[S_1](Y) = \alpha_c(S_1) \sqcup apre_c(Y)$ is the abstract transfer function and $G[S_0]$ has been defined in Sect. 2.3.

Proof Similar to the proof of Proposition 2. We obtain only an inclusion for procedure calls because the abstract precondition is not distributive. □

Combining forward and backward analysis Backward analysis alone does not deliver precise information, because the standard semantics taken backward has a high degree of non-determinism. It is really useful when it is intersected with the

result of a forward analysis. It allows to select among reachable states those leading to a final states. As our abstract precondition operator is not distributive, focusing on reachable states *during the backward analysis* also enables a better precision.

As the two analyses are based on different instrumentation, two solutions are possible for intersecting a backward analysis with the results of a forward analysis:

- either one first projects the results of the forward analysis by existentially quantifying the instrumentation variables $\mathbf{g}_0, \mathbf{fp}_0$;
- or one can add the variables $\mathbf{g}_0, \mathbf{fp}_0$ in the instrumented backward semantics; this enables a matching between actual and (copies of) formal input parameters in the procedure return operation; for instance in point (e') of Fig. 15, we would add the invariant $x_0 \geq 0$, see the discussion referencing the footnote 4.

7 Combining stack and data abstractions

In the previous sections, we embedded the control encoded by the *pc* variables in the environments in order to simplify the notations, but in practice we manipulate explicit CFGs. If we do this, the concurrent stack abstract domain A_c of Eq. (5.3) can be rewritten as

$$\Big(K^1 \times K^2 \to \wp(Env^1 \times Env^2)\Big) \times \Big((K^1 \to \wp(Env^1)\Big) \times \Big((K^2 \to \wp(Env^2)\Big)$$

in which environments define the values of ordinary data variables. Any abstraction for environments $\wp(Env) \leftrightarrows Env^\sharp$ can now be applied to A_c in order to obtain an implementable domain

$$A_c^\sharp = (K^1 \times K^2 \to Env^\sharp) \times (K^1 \to Env^\sharp) \times (K^2 \to Env^\sharp)$$

Provided that the lattice $Env^\sharp$ is equipped with meet and join operators, an abstract equality constraint between variables/dimensions, an abstract existential quantification, and an abstract operator $R^\sharp$ for intraprocedural instructions R, the predicate formulation of $apost$ and $apre$ given in Eqs. (9)–(10) and in Fig. 14 can be implemented. Observe that the ability to represent accurately equality constraints implies that the abstract domain is a relational one, in the sense that it can represent properties linking the values of different variables, unlike, for instance, the interval abstract domain [4].

Literature offers several example of suitable abstractions for environments, as shown by the following three examples:

1. **when all variables are Booleans**, $Env \simeq \mathbb{B}^n$, A_c is finite, and properties can be represented exactly with BDDs [2];
2. **when all variables are of numerical types**, $Env \simeq \mathbb{R}^n$, and properties in $\wp(Env)$ can be abstracted by octagons [31], convex polyhedra [7], etc… In this case, only intraprocedural instructions R and logical disjunction will induce a further approximation in $apost_c^\sharp : A_c^\sharp \to A_c^\sharp$ w.r.t. $apost_c : A_c \to A_c$.
   The example of Sect. 5.2 was analyzed without abstracting numerical variables. Applying on it the octagon or convex polyhedra abstract domain would actually lead to the same result (unless widening is applied).
3. **when variables are either Boolean or pointers to memory cells**, [37] proposes an abstraction in which Boolean variables, pointers and memory configurations are represented and abstracted using 3-valued logical structures: $\wp(Env) \simeq \wp(2-STRUCT) \leftrightarrows \wp(3-BSTRUCT)$. All the needed operations for our concurrent analysis method have actually already been implemented in the TVLA tool [29], in the context of interprocedural shape analysis of *sequential* programs, as described in [23,24].
   These papers show indeed how to represent *input/output relations* between shape graphs seen as 2-valued logical structures, by using 2-valued logical structure on a duplicated vocabulary, and resorts to the method of [37] to abstract such relations with 3-valued logical structures. This approach enabled the relational interprocedural shape analysis of sequential programs reminded in Sect. 5.1.1: The memory configuration is seen there as a (rather special) global variable, and parameter passing is encoded with 3-valued logical formula.
   Thus, in the same way that [24] implements the sequential procedure return operation defined by rule (6) in the 3-valued logic abstract domain, it is also possible to implement the concurrent procedure return operation defined by rule (9).

Complexity analysis We analyze here the space (resp. computational) worst-case complexity of the domain $A_c^\sharp$, by considering the size of the representation of abstract values (resp. the cost of operations on them). We assume that $\varphi(d)$ denotes the space/computational (worst-case) complexity of abstract environments $Env^\sharp$ of dimension d. For instance, the space complexity $\varphi(d)$ is 2^d for BDDs and d^3 for octagons.

Table 1 (left part) gives the complexity results in function of recursion and concurrency features. If we assume that $\varphi(d)$ is bounded by 2^d (case of Boolean programs without data abstraction) the complexity is

1. polynomial in the size k of the CFGs,
2. exponential in the number n of threads,
3. in $\mathcal{O}(\varphi(nd))$ if $d = g + l$ is the number of visible variables in each thread: we inherit the complexity of the data abstraction modulo a factor n.

Table 1 Complexity comparison

Program	single-thread	concurrent
single procedure	$k\cdot\varphi(g+l)$	$k^n\cdot\varphi(g+nl)$ (model-checking)
recursion	$2k\cdot\varphi(2g+l)$ (relational interproc. analysis)	$k^n\cdot\varphi\big((n+1)g+nl\big)$ $+\ nk\cdot\varphi(2g+l)$

n : number of threads
k : number of control points
g : number of global variables
l : number of local variables (max. per thread)
$\varphi(d)$: complexity of d-dimensional environments

(1) and (2) correspond to the complexity of model-checking, which is not surprising as our technique reduces to it in the single-procedure case. (3) shows that the complexity of our method is higher than for the concurrent, non-recursive case due to the (expensive) duplication of variables performed by the instrumentation of Sect. 4.

We discussed here the worst-case complexity. It is important to observe that most techniques aimed at reducing the practical complexity can be reused, like Cartesian product and/or variables packing for the number of variables [16], which applies on the data abstraction regardless on the context in which it is used, or partial order reduction for concurrency [15], which explores dynamically the product CFGs of the different threads and prunes useless transitions. In our case, the addition by our interprocedural technique of the Y^1_{tl} and Y^2_{tl} components (that are not products) should not break the pruning heuristics.

8 Implementation and experiments

We implemented our analysis for programs manipulating finite-type and numerical variables. The applied data abstraction abstracts $\wp(Env) = \wp(\mathbb{B}^n \times \mathbb{R}^p)$ with functions $\mathbb{B}^n \to \mathsf{Pol}(\mathbb{R}^p)$ associating to Boolean variables convex polyhedra. These functions are implemented as MTBDDs [2] in the BddApron logico-numerical domain library[17], which relies on the Cudd BDD library [39] and the Apron numerical abstract domain library [25].

Our ConcurInterproc analyzer [18,21], which can be tried online, takes as input a concurrent program, performs forward and/or backward analysis by solving Eqs. (2) or (3) on the above-described abstract domain using Kleene iteration and possibly widening, and then displays the results using various options. It follows the architecture we discuss in [22]. During the fixpoint analysis, the global equation system is actually built dynamically from the product of initial control points, using the CFG of each thread, in order to avoid building a huge product CFG with only a small reachable part. This is done in the Fixpoint library [19] by alternating propagation phases that discover newly reachable control points, and fixpoint computation phases, in the spirit of the guided analysis technique of [14].

We experimented a number of synchronization algorithms to illustrate the precision of our method, but also in order to analyze some of the approximations it induces. These programs can be analyzed online [18].

Mutual exclusion algorithms We first analyzed a few mutual exclusion algorithms, in which code to acquire and to release the critical section is delegated to two procedures `acquire` and `release`, as done for the Peterson algorithm depicted on Fig. 16. A forward analysis (3.5 s on a 2 GHz Pentium M laptop) succeeds in showing that at most one thread can be in a critical section C1 or C2. Notice that this simple example already contains unbounded recursion (without correlation between threads), and several return sites for most procedures. We also tried the program of Fig. 17, on which the analysis of [33] does not terminate, whereas ours terminates (in 8s) and proves that the mutual exclusion is ensured at the two sites and that the `fail` instruction is not reachable in any thread.

Notice that these two small examples are demanding, in the sense that synchronization algorithms are very subtle and ask for precise analysis. Concerning running times, the size of the reachable part of the equation graph remains quite high: (217, 486) and (438, 800) for the two examples (in terms of nb. of locations and transitions).

Barrier synchronization algorithms We now experiment a synchronization barrier algorithm from [40], Fig. 1. Our method proves (in 4 s) that thread T1 cannot reach the `fail` instruction (provided that we use several descending iteration steps to recover the loss of information due to the widening on convex polyhedra).

But if we make the counters `p0`, `p1` local to the main procedure of each thread, our method fails to infer that $p_0 = p_1$ when the control is at the head of the two loops, because they become uncorrelated when both threads are in the procedure `barrier`. In this case, neither the tail environments of thread t nor the top environments contain both counters: the relation between these counters is lost and cannot be recovered on procedure return. This is a typical case where the call context taken into account, as discussed in Sect. 4, is not sufficient; here, one should add the stack top of the other thread.

This phenomenon can be limited if local variables are related to global variables. In the example of Fig. 18, which is the skeleton of a timed SystemC/TLM model with cooperative scheduling, the counters `p0` and `p1` are local, but remain correlated by the two global clocks `T0` and `T1`. Thus, we can prove that the writer cannot terminate its loop (it is too slow). This example also illustrate the usefulness of reduction techniques. Here, because context switches can occur only in the `wait` procedure, only 32 locations are explored (in 0.4 s).

```
var b0,b1,turn:bool;
initial not b0 and not b1;
proc acquire(tid:bool) returns ()
begin
  if not tid then
    b0 = true; turn = tid;
    assume (b1==false or turn==not tid);
  else
    b1 = true; turn = tid;
    assume (b0==false or turn==not tid);
  endif;
end
proc release(tid:bool) returns ()
begin
  if not tid then b0 = false;
     else b1 = false; endif;
end
proc main(tid:bool) returns ()
begin
  while random do
    acquire(tid); /* C1 */ release(tid);
  done;
  if random then main(tid); endif;
  acquire(tid); /* C2 */ release(tid);
end

thread T0:
var tid:bool;
begin tid = false; main(tid); end
thread T1:
var tid:bool;
begin tid = true; main(tid); end
```

Fig. 16 The Peterson algorithm

```
var g:uint[3],
    x,y:bool;
initial g==uint[3](0) and not x and not y;

proc foo(tid:bool,q:bool) returns ()
begin
  if not q then
    x=true; y=true; foo(tid,q);
  else
    acquire(tid); /* C1 */
    g = g + uint[3](1);
    release(tid);
  endif;
end
proc main(tid:bool) returns ()
var q:bool;
begin
  q = random;
  foo(tid,q);
  acquire(tid); /* C2 */
  if g==uint[3](0) then fail; endif;
  release(tid);
end

thread T0:
var tid:bool;
begin tid = false; main(tid); end
thread T1:
var tid:bool;
begin tid = true; main(tid); end
```

Fig. 17 The example of [33], on which our analysis terminates

9 Variations around the stack abstraction

Reducing the complexity by projection In the abstract domain A_c defined by Eq. (5.3), an abstract value is a triplet $\langle Y_{hd}, Y^1_{tl}, Y^2_{tl}\rangle$, the complexity of which is dominated by Y_{hd} and which can be viewed as a predicate $Y_{hd}(\mathbf{g}^1_0, \mathbf{fp}^1_0, \mathbf{g}^2_0, \mathbf{fp}^2_0, \mathbf{g}, \mathbf{l}^1, \mathbf{l}^2)$. Now, for the analysis to be relational, it is necessary:

1. in terms of concurrency, to relate the variables $\mathbf{g}, \mathbf{l}^1, \mathbf{l}^2$, as discussed in Sect. 5.1.2;
2. in terms of procedure call/return, to relate the call context $\mathbf{g}_0, \mathbf{fp}_0$ to the current value of variables, in order to perform the relation composition of Eq. (9).

```
var T0,T1: int;
initial T0==0 and T0==T1;
proc wait(pid:uint[1], time:int) returns ()
begin
  if pid == uint[1]0
  then T0 = T0 + time; yield; assume(T0 <= T1);
  else T1 = T1 + time; yield; assume(T1 <= T0);
  endif;
end
thread reader:
var p0,time0:int, pid:uint[1];
begin
  pid = uint[1]0; time0 = 10; p0 = 0;
  while (p0 < 100) do
    wait(pid,time0); p0 = p0 + 1;
  done;
  yield;
end

thread writer:
var p1,time1:int, pid:uint[1];
begin
  pid = uint[1]1; time1 = 20; p1 = 0;
  while (p1 < 100) do
    wait(pid,time1); p1 = p1 + 1;
  done;
  yield;
end
```

Fig. 18 Producer and consumer with wrong time synchronization, analyzed with a cooperative scheduling policy (use of `yield` instructions)

Table 2 Complexity comparison for concurrent and recursive programs

standard	variation 1	variation 2
$k^n \cdot \varphi\big((n+1)g + nl\big)$ $+nk \cdot \varphi(2g + l)$	$nk^n \cdot \varphi(2g + nl)$ $+nk \cdot \varphi(2g + l)$	$nk^{2n-1} \cdot \varphi\big(2g + (2n-1)l\big)$ $+nk^{2n-1} \cdot \varphi\big(2g + (2n-1)l\big)$

n : number of threads g : number of global variables
k : number of control points l : number of local variables (max. per thread)
$\varphi(d)$: complexity of d-dimensional environments

However, there is no strong intuition for correlating the variables $\mathbf{g}_0^1, \mathbf{fp}_0^1$ and $\mathbf{g}_0^2, \mathbf{fp}_0^2$. We could thus approximate Y_{hd} with the conjunction

$$\underbrace{Y^1_{hd}(\mathbf{g}_0^1, \mathbf{fp}_0^1,\ \mathbf{g}, \mathbf{l}^1, \mathbf{l}^2)}_{=\exists(\mathbf{g}_0^2, \mathbf{fp}_0^2)\ Y_{hd}} \ \wedge\ \underbrace{Y^2_{hd}(\mathbf{g}_0^2, \mathbf{fp}_0^2,\ \mathbf{g}, \mathbf{l}^1, \mathbf{l}^2)}_{=\exists(\mathbf{g}_0^1, \mathbf{fp}_0^1)\ Y_{hd}}$$

The new complexity of abstract values, which is given in Table 2, col. "variation 1", is lower due to the reduction of the number of (global) variables to be related in the same predicate. It is all the more interesting that the global store is likely to be more complex than the local store (for instance when it includes a model of the memory as in shape analysis [23]). We have not implemented this technique yet, but we conjecture that the negative impact on precision should be very minor in practice.

Improving the precision by extending the call context In Sect. 4, we explained that the call context of a procedure in a thread includes the full call stacks of the other threads, and we made the explicit choice to abstract away this aspect in the analysis. A refinement would be to consider the top environments of the other threads, which is a less rough abstraction of their call stacks. Combined with the previous technique, for the thread 1 we would have tail and head environments of the form:

$$\begin{array}{l} Y^1_{tl}(\mathbf{g}_0^1, \mathbf{fp}_0^1, \mathbf{g}, \mathbf{l}^1, \boxed{\mathbf{l}_0^2}, \boxed{\mathbf{l}^2}) \\ Y^1_{hd}(\mathbf{g}_0^1, \mathbf{fp}_0^1, \mathbf{g}, \mathbf{l}^1, \mathbf{l}^2, \boxed{\mathbf{l}_0^2}) \end{array}$$

where the framed variables are the additional auxiliary variables, and the (solid) arrows indicate the additional matching performed when unifying tail and head environments during procedure returns. The complexity of the resulting abstract values is given in Table 1, column "variation 2", is of course higher. Intuitively, extending the call context mechanically makes the analysis less modular and more precise.

In particular, this solution solves the precision problems raised by the example of Fig. 1 when the counters `p0` and `p1` are local variables.

10 Related work and conclusion

Our first contribution is an existence proof that it is possible to analyze concurrent, recursive programs using relational techniques. Our approach unifies the relational approach to interprocedural analysis of sequential programs, and the analysis technique for concurrent, non-recursive systems based on the product of their CFGs.

We also think that our method is conceptually elegant, based on a simple instrumentation of the concrete semantics, followed by a control abstraction that collapses stacks into sets and from which we derive mechanically an abstract semantics. Sections 6 and 9 shows that the approach is general enough to define a precise backward analysis or various alternatives to the abstraction of Sect. 5.

We showed that this method can be implemented using a non-trivial combination of BDDs and convex polyhedra, which allowed us to experiment with small (but demanding) examples combining concurrency, unbounded recursion and infinite-state variables, and to illustrate its practical relevance. More experimental results are available at [18].

We did not address here the well-known efficiency problem raised by concurrency and interleaving semantics. However most techniques attacking this problem, like identification of atomic blocks [10] and partial order reduction [15], are fully applicable in our context and can be very efficient. Moreover, as mentioned in introduction, our target application is the analysis of SystemC/TLM models of SoCs [13], which follows a cooperative scheduling policy, thus making this problem less severe.

Our plan for the future is to apply our ConcurInterproc tool to SystemC/TLM models and also to analyze concurrent data-structure algorithms using a suitable shape abstraction.

Related work We focus on general techniques dealing with a combination of recursion and concurrency. The SPADE tool [32] analyzes concurrent programs with dynamic threads and recursion by representing the program state by terms and by using rewriting techniques on sets of terms. Their running times are much higher than ours. [9] was a first step in this direction, but considers only unsynchronized concurrency. Works like [1] exploits the principles of regular model-checking, with each thread being represented with a

pushdown system communicating by *rendez-vous*. Compared to our method, those techniques cannot be combined easily with infinite data-abstractions such as convex polyhedra, but most of them can handle dynamic thread creation.

Thread-modular techniques like [12] are more efficient but inherently less precise than our method w.r.t. concurrency: they never relate the local store of the different threads and they do not track the order of the updates of the global store performed by the environment of a thread (i.e., the other threads). Malkis et al. [30] shows in the non-recursive case that such a thread-modular approach is an abstraction of the interleaving semantics. Flanagan and Qadeer [12] use explicit stacks and cannot tackle unbounded recursion (they can thus be more precise than us w.r.t. recursion). The advantage of this approach is of course efficiency and the ability to handle dynamic thread creation as in [11].

Qadeer et al. [33] is close to us in the ambition of extending relational analysis to concurrent programs. However their method is based on the notion of transactional procedures and requires the accesses to global variables to be protected by mutex, which makes it less general than ours. It is also guaranteed to terminate only for an identified class of programs, but in this case it seems that the analysis is exact, which is the good side of this approach.

According to our first experiments and our intuition, our approach should be especially efficient for the cases where the local environments of the different threads must be related (as in timed TLM models), because our analysis effectively relates them, but where synchronization mechanisms do not involve several *local* stores at different recursion depths. Otherwise an accurate analysis requires to put in the call context of procedures several stack elements of the concurrent threads.

Acknowledgments We thank the anonymous referees for their constructive suggestions that helped to largely improve the initial version of the paper.

References

1. Bouajjani, A., Müller-Olm, M., Touili, T.: Regular symbolic analysis of dynamic networks of pushdown systems. In: Concurrency Theory, CONCUR'05. LNCS, vol. 3653 (2005)
2. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **35**(8), 377 (1986)
3. Caucal, D.: On the regular structure of prefix rewriting. Theor. Comput. Sci. **106**(1), 61 (1992)
4. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: 2nd Int. Symp. on Programming, Dunod, Paris (1976)
5. Cousot, P., Cousot, R.: Static determination of dynamic properties of recursive procedures. In: IFIP Conf. on Formal Description of Programming Concepts (1977)
6. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. J. Logic Program. **13**(2–3), 103 (1992)
7. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Principles of Prog. Languages, POPL'78. ACM, New York (1978)
8. Esparza, J., Knoop, J.: An automata-theoretic approach to interprocedural data-flow analysis. In: Foundations of Software Science and Computation Structure, FoSSaCS '99. LNCS, vol. 1578 (1999)
9. Esparza, J., Podelski, A.: Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In: Principles of Prog. Languages, POPL'00. ACM, New York (2000)
10. Flanagan, C., Freund, S.N., Lifshin, M., Qadeer, S.: Types for atomicity: static checking and inference for java. ACM Trans. Program. Lang. Syst. **30**(4) (2008)
11. Flanagan, C., Freund, S.N., Qadeer, S., Seshia, S.A.: Modular verification of multithreaded programs. Theor. Comput. Sci. **338**(1–3), 153–183 (2005)
12. Flanagan, C., Qadeer, S.: Thread-modular model checking. In: SPIN'03: Workshop on Model Checking Software. LNCS, vol. 2648 (2003)
13. Ghenassia, F. (ed.): Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems. Springer, Berlin (2005)
14. Gopan, D., Reps, T.W.: Guided static analysis. In: Static Analysis Symposium, SAS'07. LNCS, vol. 4634 (Aug 2007)
15. Gueta, G., Flanagan, C., Yahav, E., Sagiv, M.: Cartesian partial-order reduction. In: SPIN'07: Model Checking Software. LNCS, vol. 4595 (2007)
16. Halbwachs, N., Merchat, D., Gonnord, L.: Some ways to reduce the space dimension in polyhedra computations. Formal Methods Syst. Des. **29**(1), 79–95 (2006)
17. Jeannet, B.: The BDDAPRON logico-numerical abstract domains library. http://www.inrialpes.fr/pop-art/people/bjeannet/bjeannet-forge/bddapron/
18. Jeannet, B.: The CONCURINTERPROC interprocedural analyzer for concurrent programs. http://pop-art.inrialpes.fr/interproc/concurinterprocweb.cgi
19. Jeannet, B.: The FIXPOINT equation solver http://www.inrialpes.fr/pop-art/people/bjeannet/bjeannet-forge/fixpoint/
20. Jeannet, B.: Relational interprocedural analysis of concurrent programs. Technical Report 6671, INRIA (Oct 2008)
21. Jeannet, B.: Relational interprocedural verification of concurrent programs. In: Software Engineering and Formal Methods, SEFM'09. IEEE (Nov 2009)
22. Jeannet, B.: Some experience on the software engineering of abstract interpretation tools. In: Int. Workshop on Tools for Automatic Program AnalysiS, TAPAS'2010. ENTCS, vol. 267, pp. 29–42. Elsevier, Amsterdam (2010)
23. Jeannet, B., Loginov, A., Reps, T., Sagiv, M.: A relational approach to interprocedural shape analysis. In: Static Analysis Symposium, SAS'04. LNCS, vol. 3148 (2004)
24. Jeannet, B., Loginov, A., Reps, T., Sagiv, M.: A relational approach to interprocedural shape analysis. ACM Trans. Program. Lang. Syst. (TOPLAS), **32**(2), Article 5 (2010)
25. Jeannet, B., Miné, A.: APRON: A library of numerical abstract domains for static analysis. In: Computer Aided Verification, CAV'2009. LNCS, vol. 5643, pp. 661–667 (2009). http://apron.cri.ensmp.fr/library/
26. Jeannet, B., Serwe, W.: Abstracting call stacks for interprocedural verification of imperative programs. In: Int. Conf. on Algebraic Methodology and Software Technology, AMAST'04. LNCS, vol. 3116 (2004)
27. Knoop, J., Steffen, B.: The interprocedural coincidence theorem. In: Compiler Construction, CC'92. LNCS, vol. 641 (1992)
28. Lal, A., Touili, T., Kidd, N., Reps, T.W.: Interprocedural analysis of concurrent programs under a context bound. In: Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08. LNCS (2008)

29. Lev-Ami, T., Sagiv, M.: TVLA: A system for implementing static analyses. In: Static Analysis Symposium, SAS'00, pp. 280–301 (2000)
30. Malkis, A., Podelski, A., Rybalchenko, A.: Thread-modular verification is cartesian abstract interpretation. In: Int. Colloquium on Theoretical Aspects of Computing (ICTAC'06). LNCS, vol. 4281 (2006)
31. Miné, A.: The octagon abstract domain. Higher-Order Symb. Comput. **19**(1), 31–100 (2006)
32. Patin, G., Sighireanu, M., Touili, T.: Spade: Verification of multithreaded dynamic and recursive programs. In: Computer Aided Verification, CAV'07. LNCS, vol. 4590 (2007)
33. Qadeer, S., Rajamani, S.K., Rehof, J.: Summarizing procedures in concurrent programs. In: Principles of Programming Languages, POPL'04. ACM, New York (2004)
34. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans. Program. Lang. Syst. **22**(2), 416–430 (2000)
35. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Principles of Prog. Languages, POPL'95. ACM, New York (1995)
36. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Sci. Comput. Program. **58**(1–2), 206–263 (2005)
37. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Prog. Lang. Syst. **24**(3), 217–298 (2002)
38. Sharir, M., Pnueli, A. : Semantic foundations of program analysis. In: Muchnick, S., Jones, N. (eds.) Program Flow Analysis: Theory and Applications, chap. 7, pp. . Prentice Hall, Upper Saddle River (1981)
39. Somenzi, F.: Cudd: Colorado University Decision Diagram Package. ftp://vlsi.colorado.edu/pub
40. Taubenfeld, G.: Synchronization Algorithms and Concurrent Programming. Prentice Hall, Upper Saddle River (2006)

Author Biography

Bertrand Jeannet is a research scientist at INRIA. He received his PhD in computer science from Institut National Polytechnique de Grenoble (INPG) in 2000. He worked as a post-doctoral researcher at University of Aalborg (Denmark), before joining INRIA-Rennes in 2001 and moving to INRIA-Grenoble in 2007. His research interests include program verification, abtsract interpretation, in particular of numerical variables, shape analysis, interprocedural and concurrent program analysis, and applications to program testing. He is the co-author of the APRON library for numerical abstract domain, as well as other tools and libraries dedicated to static analysis.