HAL open science

# Relational interprocedural analysis of concurrent programs

Bertrand Jeannet

## To cite this version:

Bertrand Jeannet. Relational interprocedural analysis of concurrent programs. [Research Report] RR-6671, INRIA. 2008, pp.36. inria-00328045

## HAL Id: inria-00328045
## https://inria.hal.science/inria-00328045v1

Submitted on 9 Oct 2008

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Relational interprocedural analysis of concurrent programs*

Bertrand Jeannet

**N° 6671**

Octobre 2008

Thème COM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--6671--FR+ENG

# Relational interprocedural analysis of concurrent programs

Bertrand Jeannet

Thème COM — Systèmes communicants
Équipe-Projet Pop Art

Rapport de recherche n° 6671 — Octobre 2008 — 33 pages

**Abstract:** We extend the relational approach to interprocedural analysis of sequential programs to concurrent programs composed of a fixed number of threads. In the relational approach, a sequential program is analyzed by computing summaries of procedures, and by propagating reachability information using these summaries. We generalize this approach to concurrent programs by computing for each thread procedure summaries that takes into account the parallel execution of the other threads.

Technically, we define our analysis method by instrumenting the operational semantics of programs, then by abstracting the call-stacks of the concurrent threads, and last by abstracting the program environments in order to lead to an effective analysis that always terminates.

This method allows to extend to concurrent programs existing relational interprocedural analysis (e.g., numerical variables analysis, shape analysis). We implemented it for programs with scalar variables, and we experiment several classical synchronisation protocols in order to illustrate the precision of our technique, but also to analyze the approximations it performs.

**Key-words:** verification of concurrent programs, interprocedural analysis, abstract interpretation

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

# Analyse relationelle interprocédurale de programmes concurrents

**Résumé :** Nous étendons aux programmes concurrents l'approche relationnelle de l'analyse interprocédurale de programmes séquentiels. Dans l'approche relationelle, un programme séquentiel est analysé en calculant des résumés de procédure, et en propageant ensuite les informations d'accessibilité au moyen de ces résumés. Nous généralisons cette approche aux programmes concurrents en calculant pour chaque tâche des résumés de procédure qui prennent en compte l'exécution parallèle des autres tâches du programme concurrent.

Techniquement, nous définissons notre méthode d'analyse en instrumentant la sémantique opérationnelle standard, puis en effectuant une interprétation abstraite des piles d'appels des tâches concurrentes. Une interprétation abstraite portant les données permet finalement l'obtention d'une analyse qui termine.

Cette méthode permet d'étendre aux programmes concurrents les analyses interprocédurales existantes (analyse des variables numériques, analyse de forme). Nous l'avons implantée pour des programmes à variables scalaires, et nous avons expérimenté plusieurs protocoles classiques de synchronisation, afin d'illustrer la précision de notre méthode, mais aussi les approximations qu'elle effectue.

**Mots-clés :** vérification de programmes concurrents, analyse interprocédurale, interprétation abstraite

# 1 Introduction

Interprocedural analysis of sequential programs is well-understood in its principles [CC77b, SP81, KS92] and more recent contributions concerns mainly algorithmical techniques and/or alternative views [RHS95, EK99, RSJM05, JS04]. However, the interprocedural analysis of concurrent programs is much harder: it is known to be undecidable, even when all data variables are finite [Ram00], unlike in the sequential case [Cau92].

We consider in this paper the reachability analysis of concurrent programs with a fixed number of threads, recursive procedures, shared memory and interleaving semantics. Such an analysis has $(i)$ to model the procedure call and return semantics in each thread, and $(ii)$ to take into account the modification of global variables made by the other threads during the execution of the procedure of the thread being active. It is precisely this combination which is difficult to tackle: in the case where the other threads does not modify global variables, classical interprocedural techniques apply; in the case where no thread perform procedure calls, one can reduce the concurrent program to a sequential one, by computing the interleaved product of the control-flow-graphs of all threads, as done in model-checking.

Recent works follow various approaches. A first approach is thread-modular analysis, in which one consider a thread interacting with its environment. [FQ03] proposes a method taking into account the procedure calls of a thread while inferring in parallel environment assumptions that abstracts the possible steps of other threads. The advantage (and the motivation) of this approach is to limit the combinatorial explosion typical of concurrency with an interleaving semantics. However the obtained analysis is not very accurate because of the abstraction of the environment threads. Another approach is to be less general on the class of considered program: [QRR04] defines a notion of transaction and transactional procedures, for which they succeed to summarize procedures. Another recently explored approach consists in focusing only on executions with a bounded number of context switches [QR05, LTKR08]. This restriction allows to reuse traditional interprocedural analysis method and to obtain completeness results, but of course modulo the considered subset of executions: the inferred invariants are not sound for any execution. In some way, this approach is reminiscent of symbolic execution, as it allows to discover bugs but it cannot be used for proving a property.

We propose here an approach that analyzes all threads in parallel and tracks effectively procedure calls and returns in a concurrent context, even in the case of unbounded recursion. In particular it is able to generate for a procedure $P$ in a thread $t$ a procedure summary $P^t$ that takes into account the possible moves performed by the other threads during the execution of $P$.

Technically, our method relies on a previous work [JS04] that reformulates classical interprocedural analysis as an abstract interpretation [CC77a] of the operational semantics of (sequential) programs. Basically, by a suitable abstraction of the program's call-stacks, one eliminates the source of infinity due to stacks. This abstraction exploits the equality between actual and formal parameters in call-stacks in order to match accurately procedure calls and returns. Then, if some data variables are infinite, a data abstraction can be applied (such as convex polyhedra for numerical variables) to lead to an effective analysis.

In this paper we generalize this stack abstraction approach to concurrent programs, in which each thread has it own call-stack. The abstraction we describe abstracts separately the stack tail of each thread, but it pairs their stack tops, in order to relate the local environments of the different threads. This abstraction results in an interprocedural analysis method for concurrent programs that is as general as corresponding methods for sequential programs; it deals only with the control part of programs, so that all all the known abstraction for data can be reused. Moreover, this method can also lead to a backward analysis, that infers necessary conditions for reaching some (typically erroneous) configurations, and which is very useful in combination with forward analysis: (i) either to improve forward analysis: sometimes a forward analysis fails to prove a property, that becomes successful when intersected with a backward analysis, because the approximations performed by the two analysis are different; (ii) or to perform a "semantic" slicing of a program, that isolates the executions that are possible counter-examples to an invariance property.

**Contributions.** Our contribution can be summarized as follows:

- we define an interprocedural analysis method for concurrent programs, which lead to both forward and backward analysis;
- we prove its soundness w.r.t. the concrete operational semantics of the considered programs by using abstract interpretation techniques;
- we show how to combine it with a data abstraction and we study the complexity of the resulting analysis;
- we implemented our technique for programs with finite-type and numerical variables, and we experimented several classical synchronisation protocols, that allows to illustrate the precision of our technique, but also to analyze the approximations it performs.

**Outline.** Section 2 defines the model of the programs we consider and gives their standard semantics. In Section 3 we define two instrumented semantics (resp. dedicated to forward and backward analysis), which equip the standard semantics with additional information that will be exploited in the stack abstraction. Section 4 motivates and defines our *concurrent stack abstraction*, describes the induced forward and backward abstract semantics together with correction proofs, and discusses optimality results. We discuss in Section 5 its practical implementation and we analyze its complexity. We eventually describe in Section 6 the experiments that we performed with our implementation, and we analyze practically the strength and weaknesses of our analysis in term of precision. Sections 7 and 8 resp. discusses the related work and concludes.

## 2 Program model and standard semantics

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

| | |
|---|---|
| $T$, $P$ : | Threads and procedures |
| $P_O^t$ : | Main procedure of thread $T^t$ |
| $GVar, \mathbf{g}$ : | Global variables |
| $FP_i, \mathbf{fp}_i$ : | Formal input parameters of $P_i$ |
| $FR_i, \mathbf{fr}_i$ : | Formal output parameters of $P_i$ |
| $LVar_i, \mathbf{l}_i$ : | Local variables of procedure $P_i$ |
| : | (including $\mathbf{fp}_i$ and $\mathbf{fr}_j$) |
| $\mathsf{s}_i, \mathsf{e}_i$ : | Entry and exit points of $P_i$ |
| $G = \langle K, I \rangle$ : | Global flow graph |

Table 1: Syntactic domains.

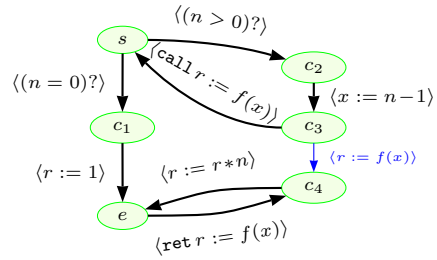


Table 2: cfg for the Factorial (single-thread) program

procedure has its own set of local variables, formal input and output parameters. We rely on shared global variables for communication and synchronisation between threads. Figs. 3-6 give examples of our program model.

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.

## 2.1 Program Syntax

The syntactic domains we use are summarised in Table 1. $\mathbf{g}$, $\mathbf{l}$ denote vectors of variables, whereas $GVar, LVar$ denote sets of variables.

A *program* is defined by a set of global variables, a set of procedures, and a set of concurrent threads (see Figs. 3-6). A *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 includes formal parameters), and by its intraprocedural CFG (control flow graph) $G_i$.

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

- $K$ is the set of *control points* of $P$, containing unique entry and exit control points $\mathsf{s}$ and $\mathsf{e}$;
- $I : K \times K \to \mathsf{Inst}$ labels edges of the graph with two kinds of instructions: intraprocedural instructions $\langle R \rangle$ and 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.

  Intraprocedural instructions are specified as a relation $R \subseteq (GEnv \times LEnv)^2$ allowing to express both a guard on global and local variables, and a transformation of those variables.

  We require the $G$ to be deterministic for procedure calls, i.e. if $I(c, c')$ is a call then there exists no $c''$ such that $I(c, c'')$ or $I(c'', c')$ is a call. The functions *call* and *ret* record matching call and return-site nodes: $call(c) = c'$ and $ret(c') = c$.

The *global CFG* $G$ of the program is constructed as the union of intraprocedural CFG $G_i$'s, further modified by replacing edges labelled 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

| | | |
|---|---|---|
| $v$ | $\in$ *Value* | : values of expressions and variables |
| $\sigma$ | $\in GEnv = GVar \to Value$ | : global environments |
| $\epsilon$ | $\in LEnv_i = LVar_i \to Value$ | : local environments for procedure $P_i$ |
| $\epsilon$ | $\in LEnv = \bigcup_i LEnv_i$ | : local environments for any procedure |
| $r = \langle c, \epsilon \rangle$ | $\in Act = K \times LEnv$ | : activation record (standard semantics) |
| $\Gamma$ | $\in Act^+$ | : stacks (sequences) of activation records[1] |
| $\langle \sigma, \Gamma \rangle$ | $\in S^t = GEnv \times Act^+$ | : state of a thread in isolation |
| $\langle \sigma, \Gamma^1, \Gamma^2 \rangle$ | $\in S = GEnv \times Act^+ \times Act^+$ | : full program states |

Table 3: Semantic domains

the return-site), see Fig. 1. Thus there are three kinds of instructions labelling edges of global CFGs: intraprocedural instructions $\langle R \rangle$, procedure calls $\langle \texttt{call}\,\mathbf{y} := P_j(\mathbf{x}) \rangle$ and procedure returns $\langle \texttt{ret}\,\mathbf{y} := P_j(\mathbf{x}) \rangle$. Entry nodes has no incoming edges except call-to-start edges. $proc(c)$ denotes the (index of the) procedure that contains $c$.

## 2.2 Operational Semantics

For the sake of simplicity, from now on we assume a program with only two threads. The semantic domains are summarised in Table 3.

A state $s = (\sigma, \Gamma_1, \Gamma_2)$ is defined by a global environment $\sigma$ and the stacks $\Gamma^t$ of *activation records* of the 2 threads, see Fig. 1. An activation record is a pair of a control point $c$ and an local environment $\epsilon$. $\langle c^t_{n_t}, \epsilon^t_{n_t} \rangle$ is the current or top activation record of the thread $T^t$.

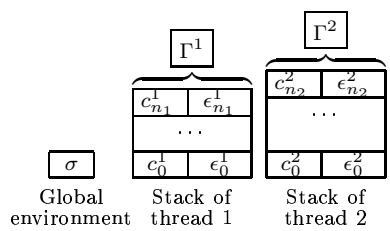


Figure 1: Program state in standard semantics

Environments map variables to values. They can be concatenated with the $\oplus$ operator, and updated with the notation $\sigma[x \mapsto v]$. 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 subvector of $\mathbf{v}$ that does not contain any variable in $\mathbf{v}'$.

Tab. 4 first defines (in SOS-style) the semantics of one thread in isolation (transition relation $\to^t$). The transition relation $\to \subseteq S \times S$ induced by the full program is then defined as a special asynchronous product of the two transition relations $\to^1$ and $\to^2$, in which the global environment is shared. We define the initial set of states as $S^0 = \{\langle \sigma, \langle s^1_0, \epsilon^1 \rangle, \langle s^2_0, \epsilon^2 \rangle \rangle \mid init(\sigma)\}$ where $s^1_0$ and $s^2_0$ denote the start point of the main procedure of each thread, and *init* an initial condition on global variables.

## 2.3 Collecting Semantics.

The forward collecting semantics induced by a transition system $(S, \to)$ characterizes the set of reachable states of a program, from a set of *initial states* $X_0 \subseteq S$:

$$reach(X_0) \stackrel{\text{def}}{=} \{s \mid \exists s_0 \in X_0 \ , \ s_0 \to^* s\} \tag{1}$$

[1] For any set $E$, $E^+ \stackrel{\text{def}}{=} \bigcup_{i>0} E^i$.

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

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

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

$$\frac{\langle \sigma,\Gamma_1\rangle \rightarrow^1 \langle \sigma',\Gamma_1'\rangle}{\langle \sigma,\Gamma_1,\Gamma_2\rangle \rightarrow \langle \sigma',\Gamma_1',\Gamma_2\rangle} \quad \text{(Conc1)} \qquad \frac{\langle \sigma,\Gamma_2\rangle \rightarrow^2 \langle \sigma',\Gamma_2'\rangle}{\langle \sigma,\Gamma_1,\Gamma_2\rangle \rightarrow \langle \sigma',\Gamma_1,\Gamma_2'\rangle} \quad \text{(Conc2)}$$

$$R^+_{\mathbf{y}:=P_j(\mathbf{x})}(\sigma,\epsilon,\epsilon_j) \stackrel{\text{def}}{=} \epsilon_j(\mathbf{fp}_j) = (\sigma\oplus\epsilon)(\mathbf{x}) \quad \text{(R+)}$$

$$R^-_{\mathbf{y}:=P_j(\mathbf{x})}(\sigma,\epsilon,\epsilon_j,\sigma',\epsilon') \stackrel{\text{def}}{=} \begin{cases} \sigma' = \sigma[\mathbf{y}\mapsto \epsilon_j(\mathbf{fr}_j) \mid \mathbf{y}\in GVar] \\ \epsilon' = \epsilon[\mathbf{y}\mapsto \epsilon_j(\mathbf{fr}_j) \mid \mathbf{y}\in LVar] \end{cases} \quad \text{(R-)}$$

Table 4: Standard Operational Semantics: transition relation $\rightarrow^t$ of the thread $T^t$ and transition relation $\rightarrow$ of the full program. The relations on environments $R^+$ and $R^-$ define parameter passing mechanisms.

It is the standard semantics for inferring invariants. For $X \subseteq S$, we define the concrete postcondition operator $post(X) \stackrel{\text{def}}{=} \{s' \mid \exists s \in X : s \rightarrow s'\}$, which we will actually decompose into operators associated to the transitions of the interprocedural flow graph $G$:

$$post(X) = \bigcup_{(c,c')\in K\times K} post(c \xrightarrow{I(c,c')} c')(X)$$

$post(c \xrightarrow{I(c,c')} c')(X)$ is easily deduced from the semantic rules of Tab. 4. For $X_0, X \subseteq S$, we define the forward transfer function $F[X_0](X) \stackrel{\text{def}}{=} X_0 \cup post(X)$. Since $F[X_0]$ is monotone and continuous, according to Kleene's theorem we have

$$reach(X_0) \ = \ lfp(F[X_0]) \ = \ \bigcup_{n\geq 0} (F[X_0])^n(\emptyset)$$

By duality, the backward collecting semantics characterizes the set of states coreachable from (i.e. leading to) a set of *final states*. It is the natural choice for inferring or checking a necessary condition on a program state to reach a final, typically erroneous configuration. We get the following definitions:

$$\begin{aligned} pre(X) &= \{s \mid \exists s' \in X : s \rightarrow s'\} \\ coreach(X_0) &= lfp(G[X_0]) \ \text{ with } \ G[X_0](X) = X_0 \cup pre(X) \end{aligned}$$

# 3 Instrumenting the standard semantics

We introduce now a modified semantics, in which

1. global variables will be passed forth and back on procedure calls;

2. procedures will keep a frozen copy of formal parameters (including the those holding the value of global variables).

This technique is classical in relational interprocedural analysis, for relating the possible states of a procedure at its start point with its possible states at its exit point. The only difference in the concurrent case is that we have to ensure that the threads agrees on the current value of global variables, as each of them will have its own copy of them.

We will actually define two instrumented semantics, one dedicated to forward analysis and the other one dedicated to backward analysis. The goal of an instrumented semantics is typically to transform properties on executions into properties on (instrumented) states, that can be later exploited in a more abstract semantics. In our case, we will obtain strong properties on possible call-stacks. In particular, it will provide a necessary conditions for an activation record to lie below another activation record in call-stacks, that will be exploited by the stack abstraction that will be introduced in Section 4.

## 3.1 A forward instrumented semantics

In the forward instrumented semantics an activation record is of the form $\langle c, \varsigma, \sigma, \epsilon \rangle$, where

- $\varsigma \in GFPEnv = GVar \cup FP \rightarrow Value$ keeps track of the values of global variables and formal parameters at the start point of the procedure $P_{proc(c)}$;
- $\sigma \in GEnv$ and $\epsilon \in LEnv$ are the global and local environment at point $c$.

Such an activation record links the values of formal parameters and global variables at start point to the current values of global and local variables at point $c$.

We now have $Act_i = K \times (GFPEnv \times GEnv \times LEnv)$ and $S_i = Act_i \times Act_i$ (with the constraint that the top activation records agree on the values of global variables). To lighten notations, we will denote simply $\epsilon \in Env$ the new environments associated to control points, with the convention that $\mathbf{g}_0$ and $\mathbf{fp}_0$ denotes the copy of global variables and formal parameters holding their value at start point of the current procedure.

Tab. 5 defines the semantic rules. Rules (Conc1F) and (Conc2F) takes care of propagating the update of global variables induced by one thread to the concurrent thread. As already precised, the top activation records of the concurrent stacks always agree on the current value of global variables. However it is not necessarily the case for tail activation records, that memorize the value of global variables at call-sites. The new set of initial states is defined as

$$S_i^0 = \{\langle \langle s_0^1 , \varsigma, \sigma, \epsilon^1 \rangle, \langle s_0^2 , \varsigma, \sigma, \epsilon^2 \rangle \rangle \mid init(\sigma) \wedge \varsigma = \sigma\}$$

**Properties of states in the forward instrumented semantics.** In this semantics, reachable call-stacks are necessarily *well-formed* in the following sense.

**Definition 1 (Well-formed stacks and states)**
*A stack* $\Gamma = \langle c_0, \epsilon_0 \rangle \ldots \langle c_n, \epsilon_n \rangle \in Act_i^+$ *is* well-formed *if, for any* $i < n$*:*

$$\frac{\begin{array}{c} I^t(c,c') = \langle R \rangle \\ R(\epsilon,\epsilon') \wedge \epsilon(\mathbf{g}_0,\mathbf{fp}_0) = \epsilon'(\mathbf{g}_0,\mathbf{fp}_0) \end{array}}{\Gamma \cdot \langle c,\epsilon\rangle \rightarrow_i^t \Gamma \cdot \langle c',\epsilon'\rangle} \quad \text{(IntraF)}$$

$$\frac{\begin{array}{c} I^t(c,s_j) = \langle \texttt{call}\,\mathbf{y} := P_j(\mathbf{x})\rangle \\ R^+_{\mathbf{y}:=P_j(\mathbf{x})}(\epsilon,\epsilon_j) \end{array}}{\Gamma \cdot \langle c,\epsilon\rangle \rightarrow_i^t \Gamma \cdot \langle c,\epsilon\rangle \cdot \langle s_j,\epsilon_j\rangle} \quad \text{(CallF)}$$

$$\frac{\begin{array}{c} I^t(e_j,c) = \langle \texttt{ret}\,\mathbf{y} := P_j(\mathbf{x})\rangle \\ R^-_{\mathbf{y}:=P_j(\mathbf{x})}(\epsilon,\epsilon_j,\epsilon') \end{array}}{\Gamma \cdot \langle call(c),\epsilon\rangle \cdot \langle e_j,\epsilon_j\rangle \rightarrow_i^t \Gamma \cdot \langle c,\epsilon'\rangle} \quad \text{(RetF)}$$

$$\frac{\begin{array}{c} \Gamma_1 \rightarrow^1 \Gamma_1' \qquad \Gamma_1' = \Gamma_1'' \cdot \langle c_1',\epsilon_1'\rangle \\ \epsilon_2' = \epsilon_2[\mathbf{g} \mapsto \epsilon_1'(\mathbf{g})] \end{array}}{\langle \Gamma_1,\ \Gamma_2 \cdot \langle c_2,\epsilon_2\rangle\rangle \rightarrow_i \langle \Gamma_1',\ \Gamma_2 \cdot \langle c_2,\epsilon_2'\rangle\rangle} \quad \text{(Conc1F)}$$

$$\frac{\begin{array}{c} \Gamma_2 \rightarrow^2 \Gamma_2' \qquad \Gamma_2' = \Gamma_2'' \cdot \langle c_2',\epsilon_2'\rangle \\ \epsilon_1' = \epsilon_1[\mathbf{g} \mapsto \epsilon_2'(\mathbf{g})] \end{array}}{\langle \Gamma_1 \cdot \langle c_1,\epsilon_1\rangle,\ \Gamma_2\rangle \rightarrow_i \langle \Gamma_1 \cdot \langle c_1,\epsilon_1'\rangle,\ \Gamma_2'\rangle} \quad \text{(Conc2F)}$$

$$R^+_{\mathbf{y}:=P_j(\mathbf{x})}(\epsilon,\epsilon_j) \stackrel{\text{def}}{=} \epsilon_j(\mathbf{g}_0,\mathbf{fp}_0^j,\mathbf{g},\mathbf{fp}^j) = \epsilon(\mathbf{g},\mathbf{x},\mathbf{g},\mathbf{x}) \quad \text{(R+)}$$

$$R^-_{\mathbf{y}:=P_j(\mathbf{x})}(\epsilon,\epsilon_j,\epsilon') \stackrel{\text{def}}{=} \left\{ \begin{array}{lcl} \epsilon'(\mathbf{g}_0,\mathbf{fp}_0,\mathbf{l}\setminus\mathbf{y}) & = & \epsilon(\mathbf{g}_0,\mathbf{fp}_0,\mathbf{l}\setminus\mathbf{y}) \\ \epsilon'(\mathbf{g}\setminus\mathbf{y},\mathbf{y}) & = & \epsilon_j(\mathbf{g}\setminus\mathbf{y},\mathbf{fr}) \end{array} \right. \quad \text{(R-)}$$

Table 5: Instrumented semantics: transition relation $\rightarrow_i^t$ of the thread $T^t$ and transition relation $\rightarrow_i$ of the full program.

*(i)* *$c_i$ is a call site for the procedure $P_j$, with $j = proc(c_{i+1})$:*
$I(c_i,s_j) = \langle \texttt{call}\,\mathbf{y} := P_j(\mathbf{x})\rangle$
*(ii)* *equality between actual and formal input parameters holds:*
$\epsilon_i(\mathbf{g},\mathbf{x}) = \epsilon_{i+1}(\mathbf{g}_0,\mathbf{fp}_0^j)$.

*A state $\langle \Gamma^1,\Gamma^2\rangle \in S_i$ is* well-formed *if $\Gamma^1$ and $\Gamma^2$ are well-formed, and if top activation records agree on the current value of global variables.*

In a well-formed state, conditions $(i)-(ii)$ above are a necessary condition for an activation record to lie below another activation record in reachable call-stacks.

**Proposition 1** *Any initial state $s \in S_0^i$ is a well-formed state. If $s \in S_i$ is a well-formed state, then any $s' \in S_i$ such that $s \rightarrow_i^* s'$ is a well-formed state.*

**Soundness of the forward instrumented semantics.** It should be clear that this new semantics does not modify the behaviour for programs: one can always fall back to the standard semantics. More formally, we define the projection function $\pi : S_i \rightarrow S$ as

$$\pi\left(\left\langle \begin{array}{c} \langle c_0^1,\ \varsigma_0^1,\sigma_0^1,\epsilon_0^1\rangle \ldots \langle c_{n_1}^1,\ \varsigma_{n_1}^1,\sigma_{n_1}^1,\epsilon_{n_1}^1\rangle, \\ \langle c_0^2,\ \varsigma_0^2,\sigma_0^2,\epsilon_0^2\rangle \ldots \langle c_{n_2}^2,\ \varsigma_{n_2}^2,\sigma_{n_2}^2,\epsilon_{n_2}^2\rangle \end{array} \right\rangle\right) = \left\langle \begin{array}{c} \sigma_{n_1}^1 (= \sigma_{n_2}^2), \\ \langle c_0^1,\epsilon_0^1\rangle \ldots \langle c_{n_1}^1,\epsilon_{n_1}^1\rangle, \\ \langle c_0^2,\epsilon_0^2\rangle \ldots \langle c_{n_2}^2,\epsilon_{n_2}^2\rangle \end{array} \right\rangle$$

In the sequel, we implicitly restrict $S_i$ to its subset of well-formed states.We can now formulate the correction of the instrumented semantics w.r.t. the standard semantics.

**Proposition 2 (Soundness of the forward instrumented semantics)**
*For any $s_i, s'_i \in S_i$ and $s, s' \in S$:*
1. $s_i \in S_i^0 \implies \pi(s_i) \in S^0$ *and* $s \in S_0 \implies \exists s_i \in \pi^{-1}(S_0^i)$;
2. $s_i \rightarrow_i^* s'_i \implies \pi(s_i) \rightarrow^* \pi(s'_i)$ *and*
   $s \rightarrow^* s' \implies \exists s_i \in \pi^{-1}(s),\ \exists s'_i \in \pi^{-1}(s') : s_i \rightarrow_i^* s'_i$.

## 3.2 An instrumented semantics for backward analysis

The instrumented semantics of the previous section has nice properties for states belonging to executions going forward, but it is not the case if one consider inverse executions, that is, if we are interested in states leading to a set of final states.

In the backward instrumented semantics, top activation records will be of the form $\langle c,\ \varsigma, \sigma, \epsilon\rangle$, where

- $\varsigma \in GFREnv = GVar \cup FR \rightarrow Value$ keeps track of the values of global variables and formal output parameters at the exit point of the enclosing procedure;
- $\sigma \in GEnv$ and $\epsilon \in LEnv$ are the global and local environment at point $c$.

Tail activation records, attached to a call-site $c$ with $c \xrightarrow{\langle \mathbf{y}:=P_j(\mathbf{x})\rangle} c'$, will record in addition the value $\varepsilon(\mathbf{y})$ of actual output parameters $\mathbf{y}$ at the return-site $c'$, as they are forgotten when going backward because of their assignment.

To lighten notations, we will denote simply $\epsilon$ the new environments associated to control points, with the convention that $\mathbf{g}_0$ and $\mathbf{fr}_0$ denotes the copy of global variables and formal output parameters holding their value at exit point of the current procedure. For tail environments attached to a call-site $c$ with $c \xrightarrow{\langle \mathbf{y}:=P_j(\mathbf{x})\rangle} c'$, $\mathbf{y}_1$ denotes the copy of actual output parameters at return site $c' = ret(c)$.

Tab. 6 defines the semantic rules. We define first the inverse (standard) transition relation as: $s' \leftarrow s$ iff $s \rightarrow s'$, and we then instrument the transition system $(S, \leftarrow)$ to obtain a transition system $(S_i, \leftarrow_i)$. Notice that the additional information carried by the instrumented semantics is propagated from left to right, although we use a left arrow to remind the reader that the execution of the program is going backward.

**Properties of states in the backward instrumented semantics.** As for the forward instrumented semantics, coreachable call-stacks and states in the backward instrumented semantics are *well-formed* in the following sense.

**Definition 2 (Well-formed stacks and states)**
*A stack $\Gamma = \langle c_0, \epsilon_0\rangle \ldots \langle c_n, \epsilon_n\rangle \in Act_i^+$ is* well-formed *if, for any $i < n$:*
*(i) $c_i$ is a call site for the procedure $P_j$, with $j = proc(c_{i+1})$:*
$I(c_i, s_j) = \langle \texttt{call}\, \mathbf{y} := P_j(\mathbf{x})\rangle$
*(ii) equality between the copy of actual output parameters and formal output parameters holds:*
$\epsilon_i(\mathbf{g}, \mathbf{y}_1) = \epsilon_{i+1}(\mathbf{g}_0, \mathbf{fr}_0^j)$.

$$\frac{\begin{array}{c} I^t(c,c') = \langle R \rangle \\ R(\epsilon,\epsilon') \wedge \epsilon(\mathbf{g}_0,\mathbf{fr}_0) = \epsilon'(\mathbf{g}_0,\mathbf{fr}_0) \end{array}}{\Gamma \cdot \langle c',\epsilon' \rangle \leftarrow_i^t \Gamma \cdot \langle c,\epsilon \rangle} \quad \text{(IntraB)}$$

$$\frac{\begin{array}{c} I^t(c,s_j) = \langle \texttt{call}\,\mathbf{y} := P_j(\mathbf{x}) \rangle \\ R^+_{\mathbf{y}:=P_j(\mathbf{x})}(\epsilon,\epsilon_j,\epsilon') \end{array}}{\Gamma \cdot \langle c,\epsilon \rangle \cdot \langle s_j,\epsilon_j \rangle \leftarrow_i^t \Gamma \cdot \langle c,\epsilon' \rangle} \quad \text{(CallB)}$$

$$\frac{\begin{array}{c} I^t(e_j,c) = \langle \texttt{ret}\,\mathbf{y} := P_j(\mathbf{x}) \rangle \\ R^-_{\mathbf{y}:=P_j(\mathbf{x})}(\epsilon,\epsilon',\epsilon_j) \end{array}}{\Gamma \cdot \langle c,\epsilon \rangle \leftarrow_i^t \Gamma \cdot \langle call(c),\epsilon' \rangle \cdot \langle e_j,\epsilon_j \rangle} \quad \text{(RetB)}$$

$$\frac{\begin{array}{c} \Gamma_1 \leftarrow^1 \Gamma'_1 \qquad \Gamma'_1 = \Gamma''_1 \cdot \langle c'_1,\epsilon'_1 \rangle \\ \epsilon'_2 = \epsilon_2[\mathbf{g} \mapsto \epsilon'_1(\mathbf{g})] \end{array}}{\langle \Gamma_1,\ \Gamma_2 \cdot \langle c_2,\epsilon_2 \rangle \rangle \leftarrow_i \langle \Gamma'_1,\ \Gamma_2 \cdot \langle c_2,\epsilon'_2 \rangle \rangle} \quad \text{(Conc1B)}$$

$$\frac{\begin{array}{c} \Gamma_2 \leftarrow^2 \Gamma'_2 \qquad \Gamma'_2 = \Gamma''_2 \cdot \langle c'_2,\epsilon'_2 \rangle \\ \epsilon'_1 = \epsilon_1[\mathbf{g} \mapsto \epsilon'_2(\mathbf{g})] \end{array}}{\langle \Gamma_1 \cdot \langle c_1,\epsilon_1 \rangle,\ \Gamma_2 \rangle \leftarrow_i \langle \Gamma_1 \cdot \langle c_1,\epsilon'_1 \rangle,\ \Gamma'_2 \rangle} \quad \text{(Conc2B)}$$

$$R^+_{\mathbf{y}:=P_j(\mathbf{x})}(\epsilon,\epsilon_j,\epsilon') \stackrel{\text{def}}{=} \left\{ \begin{array}{l} \epsilon'(\mathbf{g},\mathbf{x}) = \epsilon_j(\mathbf{g},\mathbf{fp}_j) \\ \epsilon'(\mathbf{g}_0,\mathbf{fr}^j_0,\mathbf{l} \setminus \mathbf{y}) = \epsilon(\mathbf{g}_0,\mathbf{fr}^j_0,\mathbf{l} \setminus \mathbf{y}) \end{array} \right. \quad \text{(R+)}$$

$$R^-_{\mathbf{y}:=P_j(\mathbf{x})}(\epsilon,\epsilon',\epsilon_j) \stackrel{\text{def}}{=} \left\{ \begin{array}{lcl} \epsilon'(\mathbf{g}_0,\mathbf{fr}^j_0,\mathbf{l} \setminus \mathbf{y},\mathbf{y}_1) & = & \epsilon(\mathbf{g}_0,\mathbf{fr}^j_0,\mathbf{l} \setminus \mathbf{y},\mathbf{y}) \\ \epsilon_j(\mathbf{g}_0,\mathbf{fr}^j_0,\mathbf{g},\mathbf{fr}^j) & = & \epsilon(\mathbf{g},\mathbf{y},\mathbf{g},\mathbf{y}) \end{array} \right. \quad \text{(R-)}$$

Table 6: Backward instrumented semantics: inverse transition relation $\leftarrow_i^t$ of the thread $T^t$ and inverse transition relation $\leftarrow_i$ of the full program.

*A state $\langle \Gamma^1,\Gamma^2 \rangle \in S_i$ is* well-formed *if $\Gamma^1$ and $\Gamma^2$ are well-formed, and if top activation records agree on the current value of global variables.*

As for the forward case, conditions $(i) - (ii)$) above give a necessary condition for an activation record to lie below another activation record in coreachable call-stacks.

**Proposition 3** *If $s_0 \in Act_i \times Act_i$ is a well-formed state, then any $s \in S_i$ such that $s_0 \leftarrow_i^* s$ is a well-formed state.*

**Soundness of the backward instrumented semantics.** We formalize the soundness of the backward instrumented semantics w.r.t. the standard semantics as for the forward instrumented semantics. We have the projection function $\pi : S_i \to S$ with

$$\pi\left(\left\langle \begin{array}{c} \langle c^1_0,\ \varepsilon^1_0,\varsigma^1_0,\sigma^1_0,\epsilon^1_0 \rangle \ldots \langle c^1_{n_1},\ \varsigma^1_{n_1},\sigma^1_{n_1},\epsilon^1_{n_1} \rangle, \\ \langle c^2_0,\ \varepsilon^2_0,\varsigma^2_0,\sigma^2_0,\epsilon^2_0 \rangle \ldots \langle c^2_{n_2},\ \varsigma^2_{n_2},\sigma^2_{n_2},\epsilon^2_{n_2} \rangle \end{array} \right\rangle\right) = \left\langle \begin{array}{c} \sigma^1_{n_1}(=\sigma^2_{n_2}), \\ \langle c^1_0,\epsilon^1_0 \rangle \ldots \langle c^1_{n_1},\epsilon^1_{n_1} \rangle, \\ \langle c^2_0,\epsilon^2_0 \rangle \ldots \langle c^2_{n_2},\epsilon^2_{n_2} \rangle \end{array} \right\rangle$$

In the sequel, we implicitly restrict $S_i$ to its subset of well-formed states.

**Proposition 4 (Soundness of the backward instrumented semantics)**
*For any* $s_i, s'_i \in S_i$ *and* $s, s' \in S$*:*
$s_i \leftarrow_i^* s'_i \implies \pi(s_i) \leftarrow^* \pi(s'_i)$ *and*
$s \leftarrow^* s' \implies \exists s_i \in \pi^{-1}(s),\ \exists s'_i \in \pi^{-1}(s') : s_i \leftarrow_i^* s'_i$.

### 3.3 Some facts about Abstract Interpretation

We just remind some definitions and properties of abstract interpretation framework, that will be discussed in the sequel of the paper. We refer to [CC92] for a detailed presentation of abstract interpretation.

A Galois connection $(L, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (L^\sharp, \sqsubseteq^\sharp)$ between two complete lattices induces the following properties on abstraction and concretization functions: (i) $\alpha$ and $\gamma$ are monotone (increasing); (ii) $\alpha$ 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

$$\mathit{lfp}(F^\sharp) \sqsupseteq \alpha(\mathit{lfp}(F))$$

$\mathit{lfp}(F^\sharp)$ is then an overapproximation of $\mathit{lfp}(F)$ in the sense: $\gamma(\mathit{lfp}(F^\sharp)) \sqsupseteq \mathit{lfp}(F)$. Considering for $F$ the forward transfer function $F[X_0](X)$ defined above, with $L = \wp(S)$, this theorem allows to compute an overapproximation of reachable states in a simpler lattice $L^\sharp$, using a correct approximation of $F[X_0](X)$ in $L^\sharp$.

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

$$\mathit{lfp}(F^\sharp) = \alpha(\mathit{lfp}(F)) \tag{2}$$

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

## 4 Stack abstraction and derived semantics

We will use the following functions on stacks: for any stack $\Gamma = r_0 \ldots r_n \in E^*$, $hd(\Gamma) = \{r_n\}$, $tl(\Gamma) = \{r_i \mid 0 \leq i < n\}$ and $elts(\Gamma) = hd(\Gamma) \cup tl(\Gamma)$. These functions are extended to sets of stacks.

### 4.1 Two sources of inspiration

The abstract domain we propose for concurrent and recursive programs is inspired by two techniques.

The first one is the functional or relational approach described in [SP81, KS92]. In this approach, 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 behaviour or *summary* $R_P(x, x')$. The effect of a call to $P$ on a set of reachable states $Q(x)$ is obtained by considering $Q'(x') = \exists x : Q(x) \wedge R_P(x)$.

The call-stack abstraction of [JS04] formalizes this approach within abstract interpretation framework. Starting from the instrumented semantics of Section 3.1, in which environments relate the input state of a procedure (variables

$\mathbf{g}_0, \mathbf{fp}_0$) and their current state (variables $\mathbf{g}, \mathbf{l}$), we define the Galois connection $\wp(Act_i^+) \underset{\alpha_f}{\overset{\gamma_f}{\leftrightarrows}} \wp(Act_i) \times \wp(Act_i)$ with

$$\alpha_f \ : \ \overbrace{\{r_0 \ldots r_n\}}^{\Gamma} \quad \mapsto \quad \left\langle \begin{array}{c} hd(\Gamma), \\ tl(\Gamma) \end{array} \right\rangle = \left\langle \begin{array}{c} \{r_n\}, \\ \{r_i \mid 0 \leq i < n\} \end{array} \right\rangle \tag{3}$$

$$\gamma_f \ : \ \langle Y_{hd}, Y_{tl} \rangle \quad \mapsto \quad \left\{ s = r_0 \ldots r_n \left| \begin{array}{l} r_n \in Y_{hd} \\ \forall 0 \leq i < n : r_i \in Y_{tl} \\ s \text{ is a well-formed stack} \end{array} \right. \right\} \tag{4}$$

In the induced abstract semantics, when computing the effect of a procedure return (rule (RetF) of Tab. 5), the well-formedness condition in the definition of $\gamma_f$ allows to match suitable pairs of tail and top environments with the condition $(ii)$ of Definition 1, so as to implement the relation composition of [KS92] (modulo the details due to the parameter passing mechanism). [JS04] gives an alternative proof of the interprocedural coincidence theorem of [KS92] in this framework.

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

$$S = GEnv \times (K^1 \times LEnv^1) \times (K^2 \times LEnv^2)$$

The usual technique for analyzing such systems is to observe that

$$\wp(S) \simeq K^1 \times K^2 \rightarrow \wp(GEnv \times LEnv^1 \times LEnv^2) \tag{5}$$

and to possibly abstract sets of (tuple of) environments $\wp(GEnv \times LEnv^1 \times LEnv^2)$ by an abstract lattice $Env^\sharp$. It is important to observe that the ability to relate the local environments of concurrent threads is important.

- There is first a technical reason. Assume that we maintain separate predicates $Y^1(g, l^1) = (g = l^1)$ and $Y^2(g, l^2) = (g = l^2 - 1)$ for two threads, and that the thread 1 modifies the value of the global variable $g$ with an 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 $Y = Y^1 \wedge Y^2 = (g = l^1 \wedge l^1 = l^2 - 1)$, to compute the effect of the instruction on $Y$, and then to forget the variable $l^1$ (one obtains $(g = 2l^2 - 2)$).

  The conclusion is that one need to relate the local environments of different threads, at least temporarily, when a global variable is assigned in one thread.

- There is also a precision reason. Consider the program of Fig. 2, in which two threads synchronize their parallel execution by *rendez-vous* on a channel `a` (this synchronization mechanism can be implemented using global shared variables). 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 synchronisation instruction. If the possible environments of each thread are inferred separately, the non-termination of the thread `T2` cannot be proved.

```
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
```

Figure 2: Two single-procedure concurrent threads, synchronizing on a channel `a`

## 4.2 Concurrent stack abstraction

**Intuition.** The discussion of the previous section lead us to generalize the stack abstraction of [JS04] reminded above. Assuming two threads, an abstract value will be defined by three functions $Y_{hd}$, $Y^1_{tl}$ and $Y^2_{tl}$, where

- $Y_{hd} : K^1 \times K^2 \to \wp(Env^1 \times Env^2)$ associates to pairs of control points $(c^1, c^2)$ sets of environments $Y_{hd}(c^1, c^2)(\mathbf{g},\ \mathbf{g}^1_0, \mathbf{fp}^1_0, \mathbf{l}^1,\ \mathbf{g}^2_0, \mathbf{fp}^2_0, \mathbf{l}^2)$ relating the global variables and the local variables of both threads (as in Eqn. (5)), as well as auxiliary variables introduced by the instrumented semantics.
- $Y^t_{tl} : K^t \to \wp(Env^t)$, $t = 1, 2$, associate to call-sites $c$ the set of tail environments $Y^t_{tl}(\mathbf{g},\ \mathbf{g}^t_0, \mathbf{fp}^t_0, \mathbf{l}^t)$ of thread $t$. Hence, the tail activation records of the different threads are not *directly* correlated, but *indirectly* they are linked by the means of global variables $\mathbf{g}$. This limits the loss of information due to the abstraction, as illustrated in Section 6.

It is interesting to understand the kind of procedure summaries we obtain with the function $Y_{hd}$. If we consider a procedure $P_j$ called by the thread 1 (with $c \xrightarrow{\mathbf{y}:=P_j(x)} c'$), $Y_{hd}(e_j, c^2)(\mathbf{g},\ \mathbf{g}_0, \mathbf{fp}_0, \mathbf{l},\ \mathbf{g}^2_0, \mathbf{fp}^2_0, \mathbf{l}^2)$ can be seen as a relation between the input $(\mathbf{g}_0, \mathbf{fp}_0)$ and the output $(\mathbf{g}, \mathbf{l})$ of $P_j$ in thread 1, that depends on the current state of the thread 2, given by control point $c^2$ and local variables $\mathbf{l}^2$ (the remaining variables $\mathbf{g}^2_0, \mathbf{fp}^2_0, \mathbf{l}^2$ does not play a direct role in this context). This relation takes into account the moves that thread 2 has performed since thread 1 started the execution of $P_j$.

Now if we consider the tail environments at the call site $Y^1_{tl}(c)(\mathbf{g}',\ \mathbf{g}'_0, \mathbf{fp}'_0, \mathbf{l}')$, we can match $Y_{hd}(e_j, c^2)$ with $Y^1_{tl}(c)$ with the constraint $\mathbf{g}' = \mathbf{g}_0 \wedge \mathbf{x}' = \mathbf{fp}_0$ (equating actual and formal parameters) and then perform the suitable operations to obtain a top environment $Y_{hd}(c', c^2)$ at the return site.

**Formal definition.** We define the following Galois connection

$$\wp(S_i) = \wp(Act^+_i \times Act^+_i) \underset{\alpha_c}{\overset{\gamma_c}{\leftrightarrows}} A_c = \wp(Act_i \times Act_i) \times \wp(Act_i) \times \wp(Act_i) \quad (6)$$

with

$$\alpha_c\Big(\big\{\langle \overbrace{r^1_0 \ldots r^1_{n_1}}^{\Gamma^1},\ \overbrace{r^2_0 \ldots r^2_{n_2}}^{\Gamma^2}\rangle\big\}\Big) = \left\langle \begin{array}{c} hd(\Gamma_1, \Gamma_2), \\ tl(\Gamma_1), \\ tl(\Gamma_2) \end{array} \right\rangle = \left\langle \begin{array}{c} \{\langle r^1_{n_1}, r^2_{n_2}\rangle\}, \\ \{r^1_{i_1} \mid 0 \leq i_1 < n_1\}, \\ \{r^2_{i_2} \mid 0 \leq i_2 < n_2\} \end{array} \right\rangle \quad (7)$$

$$\gamma_c\big(\langle Y_{hd}, Y^1_{tl}, Y^2_{tl}\rangle\big) = \left\{ \langle r^1_0 \dots r^1_{n_1}, r^2_0 \dots r^2_{n_2}\rangle \;\middle|\; \begin{array}{c} \langle r^1_{n_1}, r^2_{n_2}\rangle \in Y_{hd} \;\wedge\; \epsilon^1_{n_1}(\mathbf{g}) = \epsilon^2_{n_2}(\mathbf{g}) \\ \forall 0 \leq i_1 < n_1 \;:\; r^1_{i_1} \in Y^1_{tl} \\ \forall 0 \leq i_2 < n_2 \;:\; r^2_{i_2} \in Y^2_{tl} \\ r^1_0 \dots r^1_{n_1} \text{ and } r^2_0 \dots r^2_{n_2} \text{ are well-formed stacks} \end{array} \right\} \tag{8}$$

Observing that $\wp(Act_i) = \wp(K \times Env) \simeq K \to \wp(Env)$, we have the isomorphism

$$A_c \simeq \big(K^1 \times K^2 \to \wp(Env^1 \times Env^2)\big) \times \big(K^1 \to \wp(Env^1)\big) \times \big(K^2 \to \wp(Env^2)\big) \tag{9}$$

used at the beginning of this section. Elements of $A_c$ may not be representable. However the abstract domain $A_c$ still defines an abstract semantics that it is simpler than the concrete collecting semantics, and that can be seen as an analysis method. In particular, this stack abstraction can be further combined with a data abstraction, as discussed in Section 5, in order to lead to an effective analysis.

Observe in particular that for Boolean programs (that have only finite-state variables) $A_c$ is a finite lattice that can be implemented without further abstraction.

**Remark 1 ($\gamma_c$ is not injective)** *The Galois connection $\wp(S_i) \underset{\alpha_c}{\overset{\gamma_c}{\leftrightarrows}} A_c$ is not a Galois injection, i.e. $\gamma_c$ is not injective, as shown by the following counter-example, on the single-thread program of Tab. 2. We consider the forward instrumented semantics, but the same phenomenon happens for the backward instrumented semantics.*

$$\begin{aligned} \gamma^f_c\left(\left\langle \begin{array}{c} \{\langle s, n_0 = n{=}1\rangle\}, \\ \{\langle c_3, n_0 = n{=}1 \wedge x{=}0\rangle,\ \langle c_3, n_0 = n{=}2 \wedge x{=}1\rangle\} \end{array} \right\rangle\right) \\ = \gamma^f_c\left(\left\langle \begin{array}{c} \{\langle s, n_0 = n{=}1\rangle\}, \\ \{\langle c_3, n_0 = n{=}2 \wedge x{=}1\rangle\} \end{array} \right\rangle\right) \\ = \{\langle s, n_0 = n{=}1\rangle, \langle c_3, n_0 = n{=}2 \wedge x{=}1\rangle \cdot \langle s, n_0 = n{=}1\rangle\} \end{aligned}$$

*What happens is that the element $\langle c_3, n_0 = n = 1 \wedge x = 0\rangle$ cannot belong to a well-formed stack. Hence, we have $\alpha^f_c \circ \gamma^f_c(Y) \sqsubset Y$, because $\alpha^f_c \circ \gamma^f_c$ filters out elements that cannot belong to a well-formed stack.*

## 4.3 Forward abstract semantics

In this section, we abstract the transfer functions of the forward instrumented semantics (Section 3.1) in the abstract domain $A_c$.

Tab. 7 defines an abstract postcondition operator $apost_c : A_c \to A_c$ induced by the concrete postcondition $post : S_i \to S_i$. We decompose $apost_c$ according to each edge of the global CFG, and we detail only the edges of the CFG induced by the CFG of the thread 1 (the case of edges induced by the CFG of thread 2 is obtained by symmetry).

The case of intraprocedural instruction (Eqn (10)) is simple : the top environment of thread 1 is modified according to the relation $R$, and the top

$$apost(c \xrightarrow{\langle R \rangle} c')(Y_{hd}, Y^1_{tl}, Y^2_{tl}) = (Z_{hd}, Z^1_{tl}, Z^2_{tl}) \text{ with}$$

$$Z_{hd} = \left\{ \langle c', \epsilon', c^2, (\epsilon^2)' \rangle \;\middle|\; \begin{array}{c} \langle c, \epsilon, c^2, \epsilon^2 \rangle \in Y_{hd} \\ R(\epsilon, \epsilon') \wedge \epsilon(\mathbf{g}_0, \mathbf{fp}_0) = \epsilon'(\mathbf{g}_0, \mathbf{fp}_0) \\ (\epsilon^2)' = \epsilon^2[\mathbf{g} \mapsto \epsilon'(\mathbf{g})] \end{array} \right\} \tag{10a}$$

$$Z^1_{tl} = Y^1_{tl} \tag{10b}$$

$$Z^2_{tl} = Y^2_{tl} \tag{10c}$$

$$apost(c \xrightarrow{\langle \texttt{call}\, \mathbf{y} := P_j(\mathbf{x}) \rangle} s_j)(Y_{hd}, Y^1_{tl}, Y^2_{tl}) = (Z_{hd}, Z^1_{tl}, Z^2_{tl}) \text{ with}$$

$$Z_{hd} = \left\{ \langle s_j, \epsilon_j, c^2, \epsilon^2 \rangle \;\middle|\; \begin{array}{c} \langle c, \epsilon, c^2, \epsilon^2 \rangle \in Y_{hd} \\ R^+_{\mathbf{y} := P_j(\mathbf{x})}(\epsilon, \epsilon_j) \end{array} \right\} \tag{11a}$$

$$Z^1_{tl} = Y^1_{tl} \cup \left\{ \langle c, \epsilon \rangle \mid \langle c, \epsilon, c^2, \epsilon^2 \rangle \in Y_{hd} \right\} \tag{11b}$$

$$Z^2_{tl} = Y^2_{tl} \tag{11c}$$

$$apost(e_j \xrightarrow{\langle \texttt{ret}\, \mathbf{y} := P_j(\mathbf{x}) \rangle} c)(Y_{hd}, Y^1_{tl}, Y^2_{tl}) = (Z_{hd}, Z^1_{tl}, Z^2_{tl}) \text{ with}$$

$$Z_{hd} = \left\{ \langle c', \epsilon', c^2, (\epsilon^2)' \rangle \;\middle|\; \begin{array}{c} \langle e_j, \epsilon_j, c^2, \epsilon^2 \rangle \in Y_{hd} \wedge \langle call(c), \epsilon \rangle \in Y^1_{tl} \\ \epsilon(\mathbf{g}, \mathbf{x}) = \epsilon_j(\mathbf{g}_0, \mathbf{fp}^j_0) \\ R^-_{\mathbf{y} := P_j(\mathbf{x})}(\epsilon, \epsilon_j, \epsilon') \\ (\epsilon^2)' = \epsilon^2[\mathbf{g} \mapsto \epsilon'(\mathbf{g})] \end{array} \right\} \tag{12a}$$

$$Z^1_{tl} = Y^1_{tl} \tag{12b}$$

$$Z^2_{tl} = Y^2_{tl} \tag{12c}$$

Table 7: Abstract postcondition $apost_c : A_c \to A_c$. The relation $R^+$ and $R^-$ are defined by Eqns. (R+) and (R-) of Tab. 5

environment of thread 2 is modified to reflect the new values of global variables, as in Rule (IntraF) of Tab. 5. The sets of tail activation records are not modified (Eqns. (10b) and (10c)). For procedure call, Eqn. (11), the new top environment of thread 1 is initialized using the relation $R^+$ defined in Tab. 5. The set of tail activation records of thread 1 is extended by pushing the former top environment of thread 1, Eqn (11b).

The case of procedure return is the most complex. We select a global top activation record in $r_{hd} \in Y_{hd}$ and a tail activation record $r^1_{tl} \in Y^1_{tl}$ for thread 1, so that actual parameters in $r^1_{tl}$ match frozen copy of formal parameters in $r_{hd}$. This is a necessary condition for the tail activation record $r^1_{tl}$ to lie just below the projection of the top activation record $r_{hd}$ in the stack of thread 1. The new top activation record is then obtained using the relation $R^-$ defined in Tab. 5, and the top environment of thread 2 is modified to reflect the new values of global variables (when they are assigned by the procedure return). Observe that we do not remove tail activation records from $Y^1_{tl}$: we could in some cases, using the well-formedness condition, but this would be incomplete in the general case.

We then define the abstract forward transfer function $F_c[X_0](Y) = \alpha_c(X_0) \sqcup apost_c(Y)$.

**Proposition 5** (**$apost_c$ is a correct approximation of** $post$) *For any set $X \in S_i$ of well-formed states,*

$$apost_c \circ \alpha(X) \sqsupseteq \alpha_c \circ post(X)$$

*More precisely:*

1. *If $\tau$ is an intraprocedural or a call instruction, $apost_c(\tau) \circ \alpha(X) = \alpha \circ post(\tau)(X)$.*
2. *If $\tau$ is a return instruction, $apost_c(\tau) \circ \alpha(X) \sqsupseteq \alpha \circ post(\tau)(X)$*

**Corollary 1** *For any sets $X_0, X \in S_i$ of well-formed states, $F_c[X_0] \circ \alpha_c(X) \sqsupseteq \alpha_c \circ F[X_0](X)$*

*Proof.* (of the corollary)

$$\begin{array}{rcl} F_c[X_0](\alpha_c(X)) & = & \alpha_c(X_0) \sqcup apost_c(\alpha_c(X)) \\ & \sqsupseteq & \alpha_c(X_0) \sqcup \alpha_c(post(X)) \quad = \quad \alpha_c(X_0 \cup post(X)) \\ & & \quad = \quad \alpha_c(F[X_0](X)) \end{array}$$

□

Not surprisingly, the abstract semantics is less precise for return instructions, that are the most complex to compute, as they implicitly need to rebuild the stacks. We do not use the best correct approximation $\alpha_c \circ post \circ \gamma_c$, because its expression is too complex to be exploited in practice. We delay a deeper analysis of the precision of this abstract semantics to Section 6.

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

- Let $\tau = c \xrightarrow{\langle R \rangle} c'$. Observe that the function $apost_c(\tau)$ defined by Eqn (10) 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 Eqn (IntraF) we have

$$\begin{array}{l} \alpha_c \circ post(\tau)\left(\{\langle \Gamma^1 \cdot \langle c, \epsilon\rangle,\ \Gamma^2 \cdot \langle c^2, \epsilon^2\rangle\rangle\}\right) \\ = \alpha_c\left(\left\{\langle \Gamma^1 \cdot \langle c', \epsilon'\rangle,\ \Gamma^2 \cdot \langle c^2, (\epsilon^2)'\rangle\rangle \;\middle|\; \begin{array}{c} R(\epsilon, \epsilon') \\ \epsilon(\mathbf{g}_0, \mathbf{fp}_0) = \epsilon'(\mathbf{g}_0, \mathbf{fp}_0) \\ (\epsilon^2)' = \epsilon^2[\mathbf{g} \mapsto \epsilon'(\mathbf{g})] \end{array}\right\}\right) \\ = \left\langle \begin{array}{c} \{\langle c', \epsilon', c^2, (\epsilon^2)'\rangle \mid R(\epsilon, \epsilon') \wedge \epsilon(\mathbf{g}_0, \mathbf{fp}_0) = \epsilon'(\mathbf{g}_0, \mathbf{fp}_0) \wedge (\epsilon^2)' = \epsilon^2[\mathbf{g} \mapsto \epsilon'(\mathbf{g})]\}, \\ elts(\Gamma^1), \\ elts(\Gamma^2) \end{array} \right\rangle \end{array}$$

  On the other hand, from Eqn (10),

$$\begin{array}{l} apost_c(\tau) \circ \alpha\left(\left\{\langle \Gamma^1 \cdot \langle c, \epsilon\rangle,\ \Gamma^2 \cdot \langle c^2, \epsilon^2\rangle\rangle\right\}\right) \\ \qquad = apost_c(\tau)\left(\left\{\left\langle \begin{array}{c} \{\langle c, \epsilon, c^2, \epsilon^2\rangle\}, \\ elts(\Gamma^1), \\ elts(\Gamma^2) \end{array} \right\rangle\right\}\right) \\ = \left\langle \begin{array}{c} \{\langle c', \epsilon', c^2, (\epsilon^2)'\rangle \mid R(\epsilon, \epsilon') \wedge \epsilon(\mathbf{g}_0, \mathbf{fp}_0) = \epsilon'(\mathbf{g}_0, \mathbf{fp}_0) \wedge (\epsilon^2)' = \epsilon^2[\mathbf{g} \mapsto \epsilon'(\mathbf{g})]\}, \\ elts(\Gamma^1), \\ elts(\Gamma^2) \end{array} \right\rangle \end{array}$$

  The two expressions are equal.

- Let $\tau = c \xrightarrow{\mathtt{call}\ \mathbf{y} := P_j(\mathbf{x})} s_j$. Observe that the function $apost_c(\tau)$ defined by Eqn (10) 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 Eqn (CallF) we have

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

On the other hand, from Eqn (11),

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

The two expressions are equal.

- Let $\tau = e_j \xrightarrow{\mathrm{ret}\ \mathbf{y}:=P_j(\mathbf{x})} c$. 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 Eqn (RetF) we have, for $Y$ a set of well-formed states:

$$post(\tau)(Y) = \left\{\langle \Gamma^1 \cdot \langle c, \epsilon'\rangle, \Gamma^2 \cdot \langle c^2, (\epsilon^2)'\rangle\rangle \;\middle|\; \begin{array}{c} \langle \Gamma^1 \cdot \langle call(c), \epsilon\rangle \cdot \langle e_j, \epsilon_j\rangle,\ \Gamma^2 \cdot \langle c^2, (\epsilon^2)'\rangle\rangle \in Y \\ R^-_{\mathbf{y}:=P_j(\mathbf{x})}(\epsilon, \epsilon_j, \epsilon') \ \wedge\ (\epsilon^2)' = \epsilon^2[\mathbf{g} \mapsto \epsilon'(\mathbf{g})] \end{array}\right\}$$

Thus,

$$\begin{aligned}
\alpha_{hd} \circ post(\tau)(Y) &= \left\{\langle c, c^2, \epsilon', (\epsilon^2)'\rangle \;\middle|\; \begin{array}{c} \langle \Gamma^1 \cdot \langle call(c), \epsilon\rangle \cdot \langle e_j, \epsilon_j\rangle,\ \Gamma^2 \cdot \langle c^2, (\epsilon^2)'\rangle\rangle \in Y \\ R^-_{\mathbf{y}:=P_j(\mathbf{x})}(\epsilon, \epsilon_j, \epsilon') \ \wedge\ (\epsilon^2)' = \epsilon^2[\mathbf{g} \mapsto \epsilon'(\mathbf{g})] \end{array}\right\} \\
&= \left\{\langle c, c^2, \epsilon', (\epsilon^2)'\rangle \;\middle|\; \begin{array}{c} \langle \Gamma^1 \cdot \langle call(c), \epsilon\rangle \cdot \langle e_j, \epsilon_j\rangle,\ \Gamma^2 \cdot \langle c^2, (\epsilon^2)'\rangle\rangle \in Y \\ \epsilon(\mathbf{g}, \mathbf{x}) = \epsilon_j(\mathbf{g}_0, \mathbf{fp}_j) \\ R^-_{\mathbf{y}:=P_j(\mathbf{x})}(\epsilon, \epsilon_j, \epsilon') \ \wedge\ (\epsilon^2)' = \epsilon^2[\mathbf{g} \mapsto \epsilon'(\mathbf{g})] \end{array}\right\} \\
\alpha^1_{tl} \circ post(\tau)(Y) &= \bigcup\left\{elts(\Gamma^1) \mid \langle \Gamma^1 \cdot \langle call(c), \epsilon\rangle \cdot \langle e_j, \epsilon_j\rangle,\ \Gamma^2\rangle \in Y\right\} \\
\alpha^2_{tl} \circ post(\tau)(Y) &= \alpha^2_{tl}(Y)
\end{aligned}$$

On the other hand, from Eqn (12a),

$$\begin{aligned}
apost_{hd}(\tau) \circ \alpha_c(Y) &= apost_{hd}(\tau)(\langle Y_{hd}, Y^1_{tl}, Y^2_{tl}\rangle) \\
&= \left\{\langle c, \epsilon', c^2, (\epsilon^2)'\rangle \;\middle|\; \begin{array}{c} \langle e_j, \epsilon_j, c^2, \epsilon^2\rangle \in Y_{hd} \\ \langle call(c), \epsilon\rangle \in Y^1_{tl} \\ \epsilon(\mathbf{g}, \mathbf{x}) = \epsilon_j(\mathbf{g}_0, \mathbf{fp}_j) \\ R^-_{\mathbf{y}:=P_j(\mathbf{x})}(\epsilon, \epsilon_j, \epsilon') \ \wedge\ (\epsilon^2)' = \epsilon^2[\mathbf{g} \mapsto \epsilon'(\mathbf{g})] \end{array}\right\} \\
&= \left\{\langle c, \epsilon', c^2, (\epsilon^2)'\rangle \;\middle|\; \begin{array}{c} \langle \Gamma^1 \cdot \langle e_j, \epsilon_j\rangle,\ \Gamma^2 \cdot \langle c^2, \epsilon^2\rangle\rangle \in Y \\ \langle \Omega^1_0 \cdot \langle call(c), \epsilon\rangle \cdot \Omega^1_1,\ \Omega^2\rangle \in Y \\ \epsilon(\mathbf{g}, \mathbf{x}) = \epsilon_j(\mathbf{g}_0, \mathbf{fp}_j) \\ R^-_{\mathbf{y}:=P_j(\mathbf{x})}(\epsilon, \epsilon_j, \epsilon') \ \wedge\ (\epsilon^2)' = \epsilon^2[\mathbf{g} \mapsto \epsilon'(\mathbf{g})] \end{array}\right\} \\
&\sqsupseteq \alpha_{hd} \circ post(\tau)(Y)
\end{aligned}$$

From Eqn (12b), $apost^1_{tl}(\tau) \circ \alpha_c(Y) = \alpha^1_{tl}(Y) \sqsupseteq \alpha^1_{tl} \circ post(\tau)(Y)$
From Eqn (12c), $apost^2_{tl}(\tau) \circ \alpha_c(Y) = \alpha^2_{tl}(Y) = \alpha^2_{tl} \circ post(\tau)(Y)$

□

## 4.4 Backward abstract semantics

Tab. 8 defines similarly an abstract precondition operator $apre_c : A^b_c \to A^b_c$ induced by the concrete precondition $pre : S^b_i \to S^b_i$.

**Proposition 6 ($apre_c$ is a correct approximation of $pre$)** *For any set $X \in S_i$ of well-formed states,*

$$apre_c \circ \alpha(X) \sqsupseteq \alpha_c \circ pre(X)$$

*More precisely:*

$$apre(c \xrightarrow{\langle R \rangle} c')(Y_{hd}, Y^1_{tl}, Y^2_{tl}) = (Z_{hd}, Z^1_{tl}, Z^2_{tl}) \text{ with}$$

$$Z_{hd} = \left\{ \langle c, \epsilon, c^2, \epsilon^2 \rangle \;\middle|\; \begin{array}{c} \langle c', \epsilon', c^2, (\epsilon^2)' \rangle \in Y_{hd} \\ R(\epsilon', \epsilon) \wedge \epsilon'(\mathbf{g}_0, \mathbf{fr}_0) = \epsilon(\mathbf{g}_0, \mathbf{fr}_0) \\ \epsilon^2 = (\epsilon^2)'[\mathbf{g} \mapsto \epsilon(\mathbf{g})] \end{array} \right\} \tag{13a}$$

$$Z^1_{tl} = Y^1_{tl} \tag{13b}$$

$$Z^2_{tl} = Y^2_{tl} \tag{13c}$$

$$apre(c \xrightarrow{\langle \texttt{call}\, \mathbf{y} := P_j(\mathbf{x}) \rangle} s_j)(Y_{hd}, Y^1_{tl}, Y^2_{tl}) = (Z_{hd}, Z^1_{tl}, Z^2_{tl}) \text{ with}$$

$$Z_{hd} = \left\{ \langle c, \epsilon', c^2, \epsilon^2 \rangle \;\middle|\; \begin{array}{c} \langle c, \epsilon \rangle \in Y^1_{tl} \;\wedge\; \langle s_j, \epsilon_j, c^2, \epsilon^2 \rangle \in Y_{hd} \\ \epsilon(\mathbf{g}_0, \mathbf{y}_1) = \epsilon_j(\mathbf{g}_0, \mathbf{fr}^j_0) \\ R^+_{\langle \mathbf{y} := P_j(\mathbf{x}) \rangle}(\epsilon, \epsilon_j, \epsilon') \end{array} \right\} \tag{14a}$$

$$Z^1_{tl} = Y^1_{tl} \tag{14b}$$

$$Z^2_{tl} = Y^2_{tl} \tag{14c}$$

$$apre(e_j \xrightarrow{\langle \texttt{ret}\, \mathbf{y} := P_j(\mathbf{x}) \rangle} c)(Y_{hd}, Y^1_{tl}, Y^2_{tl}) = (Z_{hd}, Z^1_{tl}, Z^2_{tl}) \text{ with}$$

$$Z_{hd} = \left\{ \langle e_j, \epsilon_j, c^2, (\epsilon^2)' \rangle \;\middle|\; \begin{array}{c} \langle c, \epsilon, c^2, \epsilon^2 \rangle \in Y_{hd} \\ R^-_{\langle \mathbf{y} := P_j(\mathbf{x}) \rangle}(\epsilon, \epsilon', \epsilon_j) \\ (\epsilon^2)' = \epsilon^2[\mathbf{g} \mapsto \epsilon_j(\mathbf{g})] \end{array} \right\} \tag{15a}$$

$$Z^1_{tl} = Y^1_{tl} \cup \left\{ \langle call(c), \epsilon' \rangle \mid \langle c, \epsilon, c^2, \epsilon^2 \rangle \in Y_{hd} \;\wedge\; R^-_{\langle \mathbf{y} := P_j(\mathbf{x}) \rangle}(\epsilon, \epsilon', \epsilon_j) \right\} \tag{15b}$$

$$Z^2_{tl} = Y^2_{tl} \tag{15c}$$

Table 8: Abstract precondition $apre_c = \langle apre_{hd}, apre^1_{tl}, apre^2_{tl} \rangle : A_c \to A_c$

1. *If $\tau$ is an intraprocedural or a return instruction, $apre_c(\tau) \circ \alpha(X) = \alpha \circ pre(\tau)(X)$.*
2. *If $\tau$ is a call instruction, $apre_c(\tau) \circ \alpha(X) \sqsupseteq \alpha \circ pre(\tau)(X)$*

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

- Let $\tau = c \xrightarrow{\langle R \rangle} c'$. Observe that the function $apre_c(\tau)$ defined by Eqn (13) is distributive, hence $apre_c(\tau) \circ \alpha_c$ is distributive. We can thus compare the two distributive functions $apre_c(\tau) \circ \alpha_c$ and $\alpha_c \circ pre(\tau)$ on singleton sets.
  From Eqn (IntraF) we have

$$\begin{aligned} & \alpha_c \circ pre(\tau) \left( \left\{ \langle \Gamma^1 \cdot \langle c', \epsilon \rangle,\ \Gamma^2 \cdot \langle c^2, \epsilon^2 \rangle \rangle \right\} \right) \\ = & \alpha_c \left( \left\{ \langle \Gamma^1 \cdot \langle c, \epsilon' \rangle,\ \Gamma^2 \cdot \langle c^2, (\epsilon^2)' \rangle \rangle \mid R(\epsilon', \epsilon) \wedge \epsilon'(\mathbf{g}_0, \mathbf{fr}_0) = \epsilon(\mathbf{g}_0, \mathbf{fr}_0) \wedge (\epsilon^2)' = \epsilon^2[\mathbf{g} \mapsto \epsilon'(\mathbf{g})] \right\} \right) \\ = & \left\langle \begin{array}{c} \left\{ \langle c, \epsilon', c^2, (\epsilon^2)' \rangle \mid R(\epsilon', \epsilon) \wedge \epsilon'(\mathbf{g}_0, \mathbf{fr}_0) = \epsilon(\mathbf{g}_0, \mathbf{fr}_0) \wedge (\epsilon^2)' = \epsilon^2[\mathbf{g} \mapsto \epsilon'(\mathbf{g})] \right\}, \\ elts(\Gamma^1), \\ elts(\Gamma^2) \end{array} \right\rangle \end{aligned}$$

On the other hand, from Eqn (13),

$$\begin{aligned}
&apre_c(\tau) \circ \alpha\left(\left\{\left\langle \Gamma^1 \cdot \langle c', \epsilon\rangle,\ \Gamma^2 \cdot \langle c^2, \epsilon^2\rangle\right\rangle\right\}\right) \\
&\qquad = apre_c(\tau)\left(\left\{\left\langle \begin{array}{c} \{\langle c', \epsilon', c^2, (\epsilon^2)'\rangle\}, \\ elts(\Gamma^1), \\ elts(\Gamma^2) \end{array} \right\rangle\right\}\right) \\
&= \left\langle \begin{array}{c} \{\langle c, \epsilon, c^2, \epsilon^2\rangle \mid R(\epsilon(\mathbf{g}, \mathbf{l}), \epsilon'(\mathbf{g}, \mathbf{l})) \wedge \epsilon(\mathbf{g}_0) = \epsilon'(\mathbf{g}_0) \ \wedge\ (\epsilon^2)' = \epsilon^2[\mathbf{g} \mapsto \epsilon(\mathbf{g})]\}, \\ elts(\Gamma^1), \\ elts(\Gamma^2) \end{array} \right\rangle
\end{aligned}$$

The two expressions are equal.

- Let $\tau = c \xrightarrow{\texttt{call}\ \mathbf{y}:=P_j(\mathbf{x})} s_j$. Here $apre_{hd}(\tau)$, hence $apre(\tau)$, is not distributive, and we cannot obtain an equality, but just a soundness inclusion. From Eqn (CallF) we have

$$pre(\tau)(Y) = \left\{\left\langle \Gamma^1 \cdot \langle c, \epsilon\rangle,\ \Gamma^2\right\rangle \middle| \left\langle \Gamma^1 \cdot \langle c, \epsilon\rangle \cdot \langle s_j, \epsilon_j\rangle, \Gamma^2\right\rangle \in Y\right\}$$

Thus,

$$\begin{aligned}
\alpha_{hd} \circ pre(\tau)(Y) &= \left\{\left\langle c, c^2, \epsilon, \epsilon^2\right\rangle \middle| \left\langle \Gamma^1 \cdot \langle c, \epsilon\rangle \cdot \langle s_j, \epsilon_j\rangle,\ \Gamma^2\right\rangle \in Y\right\} \\
&= \left\{\left\langle c, c^2, \epsilon, \epsilon^2\right\rangle \middle| \left\langle \Gamma^1 \cdot \langle c, \epsilon\rangle \cdot \langle s_j, \epsilon_j\rangle,\ \Gamma^2\right\rangle \in Y \ \wedge\ \epsilon(\mathbf{g}_0, \mathbf{y}_1) = \epsilon_j(\mathbf{g}_0, \mathbf{fr}_0^j)\right\} \\
\alpha^1_{tl} \circ pre(\tau)(Y) &= \bigcup\left\{elts(\Gamma^1) \mid \left\langle \Gamma^1 \cdot \langle c, \epsilon\rangle \cdot \langle s_j, \epsilon_j\rangle,\ \Gamma^2\right\rangle \in Y\right\} \\
\alpha^2_{tl} \circ pre(\tau)(Y) &= \alpha^2_{tl}(Y)
\end{aligned}$$

On the other hand, from Eqn (14a),

$$\begin{aligned}
apre_{hd}(\tau) \circ \alpha_c(Y) &= apre_{hd}(\tau)(\langle Y_{hd}, Y^1_{tl}, Y^2_{tl}\rangle) \\
&= \left\{\left\langle c, \epsilon', c^2, \epsilon^2\right\rangle \middle| \begin{array}{c} \langle s_j, \epsilon_j, c^2, \epsilon^2\rangle \in Y_{hd} \\ \langle c, \epsilon\rangle \in Y^1_{tl} \\ \epsilon(\mathbf{g}_0, \mathbf{y}_1) = \epsilon_j(\mathbf{g}_0, \mathbf{fr}_0^j) \\ R^+_{\langle \mathbf{y}:=P_j(\mathbf{x})\rangle}(\epsilon, \epsilon_j, \epsilon') \end{array}\right\} \\
&= \left\{\left\langle c, \epsilon', c^2, \epsilon^2\right\rangle \middle| \begin{array}{c} \left\langle \Gamma^1 \cdot \langle s_j, \epsilon_j\rangle,\ \Gamma^2 \cdot \langle c^2, \epsilon^2\rangle\right\rangle \in Y \\ \left\langle \Omega^1_0 \cdot \langle c, \epsilon\rangle \cdot \Omega^1_1,\ \Omega^2\right\rangle \in Y \\ \epsilon(\mathbf{g}_0, \mathbf{y}_1) = \epsilon_j(\mathbf{g}_0, \mathbf{fr}_0^j) \\ R^+_{\langle \mathbf{y}:=P_j(\mathbf{x})\rangle}(\epsilon, \epsilon_j, \epsilon') \end{array}\right\} \\
&\sqsupseteq \alpha_{hd} \circ pre(\tau)(Y) \text{ from Eqns. (CallB) and (R+)}
\end{aligned}$$

From Eqn (14b), $apre^1_{tl}(\tau) \circ \alpha_c(Y) = \alpha^1_{tl}(Y) \sqsupseteq \alpha^1_{tl} \circ pre(\tau)(Y)$
From Eqn (14c), $apre^2_{tl}(\tau) \circ \alpha_c(Y) = \alpha^2_{tl}(Y) = \alpha^2_{tl} \circ pre(\tau)(Y)$

- Let $\tau = e_j \xrightarrow{\texttt{ret}\ \mathbf{y}:=P_j(\mathbf{x})} c$. Observe that the function $apre_c(\tau)$ defined by Eqn (13) is distributive, hence $apre_c(\tau) \circ \alpha_c$ is distributive. We can thus compare the two distributive functions $apre_c(\tau) \circ \alpha_c$ and $\alpha_c \circ pre(\tau)$ on singleton sets.
From Eqn (RetF) and the well-formedness condition on states, we have

$$\begin{aligned}
&\alpha_c \circ pre(\tau)\left(\left\{\left\langle \Gamma^1 \cdot \langle c, \epsilon\rangle,\ \Gamma^2 \cdot \langle c^2, \epsilon^2\rangle\right\rangle\right\}\right) \\
&\quad = \alpha_c\left(\left\{\left\langle \Gamma^1 \cdot \langle call(c), \epsilon'\rangle \cdot \langle e_j, \epsilon_j\rangle,\ \Gamma^2 \cdot \langle c^2, (\epsilon^2)'\rangle\right\rangle \middle| \begin{array}{c} R^-_{\langle \mathbf{y}:=P_j(\mathbf{x})\rangle}(\epsilon, \epsilon', \epsilon_j) \\ (\epsilon^2)' = \epsilon^2[\mathbf{g} \mapsto \epsilon'(\mathbf{g})] \end{array}\right\}\right) \\
&\quad = \left\langle \begin{array}{c} \{\langle e_j, \epsilon_j, c^2, (\epsilon^2)'\rangle \mid R^-_{\langle \mathbf{y}:=P_j(\mathbf{x})\rangle}(\epsilon, \epsilon', \epsilon_j) \ \wedge\ (\epsilon^2)' = \epsilon^2[\mathbf{g} \mapsto \epsilon'(\mathbf{g})]\}, \\ elts(\Gamma^1) \cup \{\langle call(c), \epsilon'\rangle \mid R^-_{\langle \mathbf{y}:=P_j(\mathbf{x})\rangle}(\epsilon, \epsilon', \epsilon_j)\}, \\ elts(\Gamma^2) \end{array} \right\rangle
\end{aligned}$$

On the other hand, from Eqn (15),

$$
\begin{aligned}
& apre_c(\tau) \circ \alpha \left( \left\{ \langle \Gamma^1 \cdot \langle c, \epsilon \rangle,\ \Gamma^2 \cdot \langle c^2, \epsilon^2 \rangle \rangle \right\} \right) \\
& \qquad = apre_c(\tau) \left( \left\{ \left\langle \begin{array}{c} \{\langle c, \epsilon, c^2, \epsilon^2 \rangle\}, \\ elts(\Gamma^1), \\ elts(\Gamma^2) \end{array} \right\rangle \right\} \right) \\
& = \left\langle \begin{array}{c} \left\{ \langle e_j, \epsilon_j, c^2, (\epsilon^2)' \rangle \mid R^-_{\langle \mathbf{y} := P_j(\mathbf{x}) \rangle}(\epsilon, \epsilon', \epsilon_j) \ \wedge\ (\epsilon^2)' = \epsilon^2[\mathbf{g} \mapsto \epsilon'(\mathbf{g})] \right\}, \\ elts(\Gamma^1) \cup \{\langle call(c), \epsilon' \rangle \mid R^-_{\langle \mathbf{y} := P_j(\mathbf{x}) \rangle}(\epsilon, \epsilon', \epsilon_j) \ \wedge\ \}, \\ elts(\Gamma^2) \end{array} \right\rangle
\end{aligned}
$$

The two expressions are equal.

□

## 4.5 Some optimality 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 this abstraction falls back to the *functional* abstraction defined in [JS04] for single-thread program, we have the following theorem, which reformulates the interprocedural coincidence theorem of [KS92].

**Theorem 1 ([JS04])** *For single-thread programs, and for sets of initial states $X_0$ composed only of one-element stacks, the abstract reachability analysis is optimal:*

$$areach_c(X_0) = \alpha_c(reach(X_0))$$

This implies that the set of *top* activation records of reachable stacks is computed exactly. In other words, the invariants inferred at each control point are the exact ones.

More interestingly, we could extend this result to multiple-threads programs in which only one thread performs procedure calls and manipulates local variables, whereas the other thread has only a main procedure without local variables. This generalization requires however the program counters of the other threads to be treated as global variables (this is discussed in Section 6.2). We did not formalize yet such a treatment of program counters, mostly to keep simple the presentation of the various semantics in Sections 2-4, so present it still as a conjecture.

**Conjecture 1** *For multiple-threads programs containing only one thread performing procedure calls, and for sets of initial states $X_0$ composed only of one-element stacks,*

$$areach_c(X_0) = \alpha_c(reach(X_0))$$

In the case of two-threads programs, with the second thread being composed of a single procedure, if $areach_c(X_0) = \langle Y_{hd}, Y^1_{tl}, \emptyset \rangle$,

$$Y_{hd} = \left\{ \langle c^1, \epsilon^1,\ c^2, \epsilon^2 \rangle \mid \left\langle \Gamma^1 \cdot \langle c^1, \epsilon^1 \rangle,\ \langle c^2, \epsilon^2 \rangle \right\rangle \in reach(X_0) \right\}$$

However, this optimality conjecture can certainly not be extended to two or more recursive threads. The main technical reason is that the effect of a procedure in a thread do no depend any more of the value of global variables and

effective parameters at call-site, but also of the *full* stack of the concurrent threads, because these threads can perform procedure returns and they can call, from smaller-size stacks, new procedures that modifies the shared global variables. A more concise observation is that such a result would contradict the undecidability result of [Ram00].

# 5 Combining stack and data abstractions for an effective analysis

We discuss now how to obtain an effective analysis using the concurrent stack abstraction introduced in the previous section, and we analyze the complexity of the resulting analysis.

We assume that we are given a Galois connection $\wp(Env) \underset{\alpha_e}{\overset{\gamma_e}{\leftrightarrows}} Env^\sharp$ that represents or abstracts properties on environments with abstract environments. The stack abstraction can then be combined with such a data abstraction, if we exploit the isomorphism of Eqn (9):

$$\begin{aligned}
\wp(S_i) &= \wp\left(\left(K^1 \times Env\right)^+ \times \left(K^2 \times Env\right)^+\right) \\
&\underset{\alpha_c}{\overset{\gamma_c}{\leftrightarrows}} \overbrace{\left(K^1 \times K^2 \to \wp(Env)\right) \times \left(K^1 \to \wp(Env)\right) \times \left(K^2 \to \wp(Env)\right)}^{A_c} \\
&\leftrightarrows \overbrace{\left(K^1 \times K^2 \to Env^\sharp\right) \times \left(K^1 \to Env^\sharp\right) \times \left(K^2 \to Env^\sharp\right)}^{A_c^\sharp} \qquad (16)
\end{aligned}$$

To precise how the abstract postcondition in $A_c$ can be abstracted in $A_c^\sharp$, we give in Tab. 9 a predicate formulation of the abstract postcondition defined in Tab. 7. Provided that the lattice $Env^\sharp$ is equipped with a least upper bound (approximating the logical conjunction), an abstract equality constraint between variables/dimensions, an abstract existential quantification on variables/dimensions, and of course an abstract operator $R^\sharp$ for intraprocedural instruction $R$, the equations of Tab. 9 can be implemented directly.

Literature offers several example of suitable representations/abstractions for programs environments.

- When all variables are Booleans (or more generally of finite type), $Env = (\mathrm{Var} \to \mathbb{B}) \simeq \mathbb{B}^n$, and properties in $\wp(Env)$ can be represented exactly with BDDs [Bry86], as well as the semantics of intraprocedural instructions, and the postcondition operator of Tab. 9 can be implemented exactly;

- When all variables are of numerical types, $Env = (\mathrm{Var} \to \mathbb{R}) \simeq \mathbb{R}^n$, and properties in $\wp(Env)$ can be abstracted by octagons [Min06], convex polyhedra [CH78], relational congruences [Gra91]. In this case equality constraints between variables and existential quantification can be applied exactly, so that 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. the abstract postcondition $apost_c : A_c \to A_c$.

Notations:
If $\langle X_{hd}, X^1_{tl}, X^2_{tl}\rangle \in A_c$

$$\text{we have } \begin{cases} X_{hd}(c^1,c^2)(\mathbf{g},\ \mathbf{g}_0,\mathbf{fp}_0,\mathbf{l},\ \mathbf{g}^2_0,\mathbf{fp}^2_0,\mathbf{l}^2) & \in \quad \wp(Env^1 \times Env^2) \\ X^1_{tl}(c^1)(\mathbf{g},\ \mathbf{g}^1_0,\mathbf{fp}^1_0,\mathbf{l}^1) & \in \quad \wp(Env^1) \\ X^2_{tl}(c^2)(\mathbf{g},\ \mathbf{g}^2_0,\mathbf{fp}^2_0,\mathbf{l}^2) & \in \quad \wp(Env^2) \end{cases}$$

We define below $Z = apost_c(\tau)(Y)$ for the various instruction $\tau$.

$$\tau = c \xrightarrow{\langle R\rangle} c'$$

$$Z_{hd}(c',c^2)(\mathbf{g}',\mathbf{g}_0,\mathbf{fp}_0,\mathbf{l}',\mathbf{g}^2_0,\mathbf{fp}^2_0,\mathbf{l}^2) = \exists \mathbf{g},\mathbf{l} : \begin{cases} Y_{hd}(c,c^2)(\mathbf{g},\mathbf{g}_0,\mathbf{fp}_0,\mathbf{l},\mathbf{g}^2_0,\mathbf{fp}^2_0,\mathbf{l}^2) \\ \wedge R(\mathbf{g},\mathbf{l},\mathbf{g}',\mathbf{l}') \end{cases} \tag{17a}$$

$$Z^1_{tl} = Y^1_{tl} \tag{17b}$$

$$Z^2_{tl} = Y^2_{tl} \tag{17c}$$

$$\tau = c \xrightarrow{\langle \texttt{call}\ \mathbf{y}{:=}P_j(\mathbf{x})\rangle} s_j$$

$$Z_{hd}(s_j,c^2)(\mathbf{g},\mathbf{g}'_0,\mathbf{fp}'_0,\mathbf{l}',\mathbf{g}^2_0,\mathbf{fp}^2_0,\mathbf{l}^2) = \exists \mathbf{g}_0,\mathbf{fp}_0,\mathbf{l} : \begin{cases} Y_{hd}(c,c^2)(\mathbf{g},\mathbf{g}_0,\mathbf{fp}_0,\mathbf{l},\mathbf{g}^2_0,\mathbf{fp}^2_0,\mathbf{l}^2) \\ \wedge(\mathbf{g}'_0,\mathbf{fp}'_0,\mathbf{fp}') = (\mathbf{g},\mathbf{x},\mathbf{x}) \end{cases} \tag{18a}$$

$$Z^1_{tl}(c)(\mathbf{g},\mathbf{g}_0,\mathbf{fp}_0,\mathbf{l}) = Y^1_{tl}(c)(\mathbf{g},\mathbf{g}_0,\mathbf{fp}_0,\mathbf{l}) \vee \exists \mathbf{g}^2_0,\mathbf{fp}^2_0,\mathbf{l}^2 : Y_{hd}(c)(\mathbf{g},\mathbf{g}_0,\mathbf{fp}_0,\mathbf{l},\mathbf{g}^2_0,\mathbf{fp}^2_0,\mathbf{l}^2) \tag{18b}$$

$$Z^2_{tl} = Y^2_{tl} \tag{18c}$$

$$\tau = e_j \xrightarrow{\langle \texttt{ret}\ \mathbf{y}{:=}P_j(\mathbf{x})\rangle} c$$

$$Z_{hd}(c,c^2)(\mathbf{g}'',\mathbf{g}_0,\mathbf{fp}_0,\mathbf{l}'',\mathbf{g}^2_0,\mathbf{fp}^2_0,\mathbf{l}^2) = \exists \mathbf{g},\mathbf{l},\mathbf{g}',\mathbf{g}'_0,\mathbf{l}_j : \begin{cases} Y^1_{tl}(call(c))(\mathbf{g},\mathbf{g}_0,\mathbf{fp}_0,\mathbf{l}) \\ \wedge Y_{hd}(e_j,c^2)(\mathbf{g}',\mathbf{g}'_0,\mathbf{fp}'_0,\mathbf{l}',\mathbf{g}^2_0,\mathbf{fp}^2_0,\mathbf{l}^2) \\ \wedge(\mathbf{g},\mathbf{x}) = (\mathbf{g}'_0,\mathbf{fp}'_0) \\ \wedge(\mathbf{g}\setminus\mathbf{y})'' = (\mathbf{g}\setminus\mathbf{y})' \\ \wedge(\mathbf{l}\setminus\mathbf{y})'' = (\mathbf{l}\setminus\mathbf{y}) \\ \wedge\mathbf{y}'' = \mathbf{fr}' \end{cases} \tag{19a}$$

$$Z^1_{tl} = Y^1_{tl} \tag{19b}$$

$$Z^2_{tl} = Y^2_{tl} \tag{19c}$$

Table 9: Abstract postcondition $apost_c : A_c \to A_c$, predicate formulation

| Program | single-thread | concurrent |
|---|---|---|
| single-procedure | $k \cdot \varphi(g+l)$ | $k^n \cdot \varphi(g+nl)$ |
| recursion | $2k \cdot \varphi(2g+l)$ | $k^n \cdot \varphi(g+n(g+l)) + nk \cdot \varphi(2g+l)$ |

$n$ : number of threads $g$ : number of global variables
$k$ : number of control points $l$ : number of local variables

Table 10: Complexity comparison

- When variables are either Boolean or pointers to memory cells, [SRW02] proposes an abstraction in which Boolean variables, pointers and memory configurations are represented and abstracted using 3-valued logical structures:

$$\wp(Env) \simeq \wp(2-STRUCT) \leftrightarrows \wp(3-BSTRUCT)$$

  As with numerical domains, all the needed operations can be implemented, as shown in [JLRS04] for interprocedural analysis of sequential programs.

**Complexity analysis.** We analyze the computational and/or space complexity of $A_c^\sharp$. Assume $n$ is the number of threads, $k$ the maximal number of control points, and $g$ and $l$ the number of global and local variables. In the abstract domain $A_c = A_{hd} \times (\prod_{0 \leq t < n} A_{tl}^t)$,

- $A_{hd}$ is a function with a finite domain of size $k^n$, the images of which are environments having at most $g + n(g+l)$ dimensions; [2]
- $A_{tl}^t$ is a function with a finite domain of size $k$, the images of which are environments having at most $2g+l$ dimensions. There are $n$ such functions.

Assuming that the (computational or spatial) complexity of abstract environments of dimension $d$ is $\varphi(d)$, the complexity of $A_c^\sharp$ is:

$$k^n \cdot \varphi(g+n(g+l)) + nk \cdot \varphi(2g+l)$$

It is clearly dominated by the first term. Assuming $\varphi(d)$ is bounded by $\mathcal{O}(2^d)$, the global complexity is

- polynomial in the size $k$ of the CFGs,
- exponential in the number $n$ of threads,
- in $\mathcal{O}(\phi(nd))$ if $d = g + l$ is the number of visible variables active in each thread: we inherit the complexity of the data abstraction modulo a multiplicative factor $n$ ($\varphi$ is exponential for convex polyhedra and BDDs, cubic for octagons, ...).

We compare in Tab. 10 the complexity results in function of the recursion and concurrency features. To summarize, the complexity of our method for concurrent, recursive programs is higher than for concurrent, non-recursive programs only due to the duplication of global variables. Provided that $g \approx l$, it is asymptotically identical.

Observe that most techniques aimed at reducing the practical complexity can be reused: partial order and symmetry reduction for concurrency [God96], Cartesian product and/or variables packing for data abstraction [HMG06, BCC+03].

[2] we merge the identical copies of global variables

# 6 Implementation and experiments

## 6.1 Implementation

We implemented our analysis for programs manipulating finite-type and numerical variables. The applied data abstraction abstracts sets of environments of the form $\wp(\mathbb{B}^n \times \mathbb{Q}^p)$ with functions of signature $\mathbb{B}^n \to \mathsf{Pol}(\mathbb{Q}^p)$ associating to the possible valuations of finite-type variables convex polyhedra approximating the possible values of numerical variables. Algorithmically, these functions are implemented as MTBDDs [Bry86], using the CUDD BDD library and the APRON numerical abstract domains library [Som, JM]. This abstraction is exact for finite-type variables, and resorts to linear relation analysis for numerical variables [HPR97].

Our ConcurInterproc analyzer takes as input a concurrent program, performs forward and/or backward analysis and displays the results. The analysis results can be projected on a chosen thread, as illustrated later on the examples. In order to specify start points for backward analysis, the language contains an instruction `fail` that not only stops the execution (as does `halt`), but also marks the control point just before the instruction as a start point. The default semantics of the input language w.r.t. concurrency is preemptive scheduling: the steps performed by the different threads are interleaved, as defined in Section 2.2. However, as we did not (yet) implement partial order reduction techniques, the results were difficult to inspect manually. Hence an option in the analyzer allows to choose a cooperative scheduling semantics, in which the instruction `yield` allow a thread to give the control in a non-deterministic way to another thread. Thus, any execution path between two `yield` instruction in a thread can be considered as an atomic section w.r.t. concurrency. More information about ConcurInterproc as well as an online analyzer can found at [Jea].

## 6.2 Experiments

We experimented a number of synchronisation algorithms to illustrate the effectiveness of our method, but also in order to analyze some of its limitation. We mainly evaluate here the precision of the analysis (evaluating its practical complexity deserves a separate study along the directions sketched in the conclusion). Moreover, in order to focus on the accuracy of the stack abstraction, we replace integers by bounded integers encoded with Booleans in some cases, to avoid approximations due to the data abstraction. The experimented programs are available at the URL [Jea].

**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. 3. A forward analysis succeeds to show that the two threads cannot be both at control point $(C)$ lying between calls to `acquire` and `release` procedures. We proved like that the correctness of Peterson and Kessel algorithms that we found in [Tau06].

More interestingly, we tried the program depicted on Fig. 4, on which the analysis of [QRR04] does not terminate, whereas ours of course terminate, but

```
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 true do
    acquire(tid);          /* C */
    release(tid);
  done;
end

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

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

Figure 3: 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);
    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);
  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
```

Figure 4: The example of [QRR04] (Fig. 8), on which our analysis terminates

proves that the mutual exclusion is ensured at the two sites and that the `fail` instruction is not reachable in any thread. Notice that here the `acquire` and `release` procedures may be called from different procedures in each thread.

**Barrier synchronisation algorithms.** We now experiment a synchronisation barrier algorithm working with one global Boolean `go` and one global counter `counter`, Fig. 5. The algorithm has a cycle of length 2 in the sense that after two calls, `go`, `lgo0` and `lgo1` take back their former value. Our analysis succeeds to show that points $B2$ and $B3$ of thread T1 are not reachable.

Some variants of this example will allows us to identify some kind of approximations are performed by our technique.

**Example 1** *If we uncomment the second call to `barrier` in thread T0 in Fig. 5, not only `C2` but also `C3` becomes reachable (the latter fact being impossible in an execution as the algorithm is really correct).*

*The explanation is that the same global top environment is associated to pair of control points* $(A0, B0)$ *and* $(A2, B2)$*, because of the cycle of length 2 mentioned above. Hence the tail environments of thread T1 associated to call-sites* $B0$ *and* $B2$ *are exactly the same. Now some top environments associated*

```
var go : bool, counter:int;
initial counter==0 and go;

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);
  endif;
  nlgo = lgo;
end                       /* E0 */

thread T0:
var lgo0:bool;
begin
  lgo0 = true;            /* A0 */
  lgo0 = barrier(lgo0); /* A1 */
/*lgo0 = barrier(lgo0); /* A2 */ */
end

thread T1:
var lgo1:bool;
begin
  lgo1 = true;            /* B0 */
  lgo1 = barrier(lgo1); /* B1 */
  lgo1 = barrier(lgo1); /* B2 */
  lgo1 = barrier(lgo1); /* B3 */
  fail;
end
```

Figure 5: A synchronisation barrier algorithm with a counter

```
var p0,p1:int;

thread T0:
var lgo0:bool;
begin
 begin
  p0 = 0; lgo0 = true;
  while p0<=5 do
    lgo0 = barrier(lgo0);
    p0 = p0 + 1;
  done;
end

thread T1:
var lgo1:bool;
begin
  p1 = 0; lgo1 = true;
  while p1<=10 do
    lgo1 = barrier(lgo1);
    p1 = p1 + 1;
  done;
  fail;
end
```

Figure 6: Variant of Fig. 5 with calls inside loops

*to* $(E0, E0)$ *(successors of calls from* $(A0, B0)$*) matches some tail environments associated to* $B0$ *to make* $(E0, B1)$ *reachable. Those same top environments will match tail environments attached to* $B2$ *to make* $(E0, B3)$*, hence* $B3$ *reachable (although at this point, the thread T0 cannot return from procedure* `barrier` *in the abstract semantics).*

*This example also illustrates the usefulness of backward analysis intersected with forward analysis: such a backward analysis enable the proof that the* `fail` *instruction is not reachable from initial configurations. However, if we add a third (resp. fourth) call to* `barrier` *in thread T0 (resp. T1), the proof fails to show that T1 does not reach its last control point, even with such a combination of forward and backward analysis.* □

A deeper explanation for the problem raised by Example 1 is that the program counters of each thread are treated as local variables in the instrumented semantics. If they were treated as global variables, when performing a call the active thread would memorize the program counter of the other thread when the call occurs, not only in the pushed tail activation record, but also in the new top environment with the frozen copies of global variables. We did not yet implement this technique, but we confirmed the intuition above with the following example.

**Example 2** *We now consider the program of Fig. 6, for which our analyzer succeeds to show that in thread T1,* $p1 \leq 6$ *at the entry of the loop, so that the*

*`fail` instruction is not reachable. Indeed, the call-sites of each thread are memorized with the global variables `p0` and `p1`, and the analysis avoid the mismatch occurring in Example 1.*

*Now, if we make the variables `p0` and `p1` local to each thread, the analysis is not able any more to show that globally $|p0 - p1| \leq 1$. These counters becomes uncorrelated when both threads are in the callee procedure barrier: in this case, neither the tail environments nor the top environments keeps track of the relation between these counters.* □

More generally, local variables are handled less precisely by our analysis than global variables, because of their temporary lifetime. If we inline the calls to `barrier`, the problem disappears, because all variables gain a global lifetime. Hence our analysis method is not insensitive to procedures inlining, as it may become more precise (and more costly) on the transformed program.

# 7 Related work

A lot of work has been dedicated to the analysis of concurrent programs. [Rin01] distinguishes several type of analysis and gives a survey of them, but most often those analysis are not defined as an instantiation of a general method to a particular goal, and their goal is not to directly infer invariants on data variables. We will focus mainly on analysis techniques dedicated to general reachability (or coreachability) analysis for concurrent programs with recursive procedures and synchronization, either by shared global variables or by using higher-level primitives like *rendez-vous*.

These techniques can first be classified according the representation of the program and of its state-space. For instance, the SPADE tool [Tou05, PST07] analyzes concurrent programs with dynamic threads and recursion by representing the program state by terms and by using rewriting techniques on sets of terms and abstracting them. [EP00] was a first step in this direction, but considered at that time unsynchronized concurrency. Another line of work exploits the automata theory and the principles of regular model-checking, with each thread being represented with a pushdown system [BET03, BMOT05]. [DRB02] uses Petri Nets models. Compared to our method, those techniques cannot be combined easily with infinite data-abstractions such as convex polyhedra. They are more adapted to finite abstraction techniques, and (concrete) counter-examples guided abstraction refinement (CEGAR) [CCK+06]. A strength of these techniques is that many of them can handle dynamic thread creation.

Thread-modular techniques have already been mentioned in the introduction [FQ03, FFQS05]. We discuss more specifically the method of [FQ03] which is presented very clearly. Its method is inherently less precise than our method, in particular because it does not track the order of the updates performed by the environment of a thread (i.e., the other threads), and it never relates the local data of the different threads. The gain of this approach is of course efficiency, and often the ability to handle dynamic thread creation as in [FFQS05]. We also think that this technique could be easily combined with abstract interpretation techniques for data variables, although only the alternative CEGAR technique has been explored until now [HJMQ03] It would be interesting to define the suitable stack abstraction, in the sense of our approach, which allows to derive

a modular method. Thread modular analysis have also been studied in more specific contexts, as in [GBCS07], but this paper does not consider recursion.

[QRR04] is closed to us in the ambition of extending relational interprocedural analysis to concurrent programs. However their method is based on the notion of transactions and transactional procedures, and is guaranteed to terminate only for an identified class of programs, which makes it less general than ours. We refer to the introduction for context-bounded analysis techniques.

# 8 Conclusion

Our approach unifies the relational approach to interprocedural analysis of sequential programs, and the classical approach to the analysis of concurrent systems composed of a fixed number of control-flow graphs, without procedure calls.

We think that our method is conceptually elegant, based on an instrumentation of the standard operational semantics, followed by an abstraction that collapses stacks of activation records into sets. The abstract semantics is then derived mechanically from the concrete semantics. Our main design efforts were to manage appropriately the global variables in the instrumented semantics, and to understand that we need one frozen copy of global variables for each thread when we relate the top activation records of each thread.

Our method separates clearly the control abstraction which replaces stacks by sets, from the data abstraction which deals with program environments. It is also very general, as it can be used in all cases where relational interprocedural analysis for sequential programs can be applied, as discussed in Section 5. For instance the interprocedural shape analysis described in [JLRS04] could be extended to concurrent programs.

We analyzed experimentally its precision, mainly on synchronisation protocols that requires a fine analysis of the interactions between threads. The conclusion is that the main source approximation is due to the temporary lifetime of local variables, and the fact that the local variables of the different threads are not related any more when they are pushed on call-stacks (although they are indirectly related via the global variables). This explains that inlining procedures may result in a better accuracy of the analysis.

In this paper we do not pretend to solve any of the well-known efficiency problems raised by concurrency, especially in the context of an interleaving semantics (*cf.* Tab. 10 in Section 5). Our ambition was rather to get as close as possible from a context-sensitive, synchronisation-sensitive analysis, although this ultimate goal is out of reach according to the undecidability result of [Ram00]. However, it should be noted that many techniques aiming at improving the efficiency of concurrent programs analysis can be applied in our context such as identifying atomic blocks [FFLQ08], partial order reduction techniques [God96, FG05, GFYS07], reducing the complexity due to the number of variables using variables packing or its variants [BCC$^+$03, HMG06].

Another already cited approach is thread-modular analysis. It would be interesting to known whether a further abstraction of our stack abstraction can result in an analysis similar to the one described in [FQ03]. Our analysis could then be used to generate annotations in a thread-modular, assume-guarantee

context [FFQS05]: the further abstraction of our analysis results would provide precise annotations describing the environment of a thread.

## References

[BCC+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Prog. Language Design and Implementation, PLDI'03*. ACM, 2003.

[BET03] Ahmed Bouajjani, Javier Esparza, and Tayssir Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Principles of Prog. Languages, POPL'03*, 2003.

[BMOT05] Ahmed Bouajjani, Markus Müller-Olm, and Tayssir Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *Concurrency Theory, CONCUR'05*, volume 3653 of *LNCS*, 2005.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8), 1986.

[Cau92] D. Caucal. On the regular structure of prefix rewriting. *Theoretical Computer Science*, 106(1), 1992.

[CC77a] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Prog. Languages, POPL'77*. ACM, 1977.

[CC77b] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of recursive procedures. In *IFIP Conf. on Formal Description of Programming Concepts*, 1977.

[CC92] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3), 1992.

[CCK+06] Sagar Chaki, Edmund M. Clarke, Nicholas Kidd, Thomas W. Reps, and Tayssir Touili. Verifying concurrent message-passing c programs with recursive calls. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'06*, volume 3920 of *LNCS*, 2006.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Prog. Languages, POPL'78*. ACM, 1978.

[DRB02] Giorgio Delzanno, Jean-François Raskin, and Laurent Van Begin. Towards the automated verification of multithreaded java programs. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'02*, volume 2280 of *LNCS*, 2002.

[EK99] Javier Esparza and Jens Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *Foundations of Software Science and Computation Structure, FoSSaCS '99*, volume 1578 of *LNCS*, 1999.

[EP00] Javier Esparza and Andreas Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *Principles of Prog. Languages, POPL'00*. ACM, 2000.

[FFLQ08] Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. Types for atomicity: Static checking and inference for java. *ACM Trans. Program. Lang. Syst.*, 30(4), 2008.

[FFQS05] Cormac Flanagan, Stephen N. Freund, Shaz Qadeer, and Sanjit A. Seshia. Modular verification of multithreaded programs. *Theor. Comput. Sci.*, 338(1-3), 2005.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Principles of Prog. Languages, POPL'05*. ACM, 2005.

[FQ03] C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN'03: Workshop on Model Checking Software*, volume 2648 of *LNCS*, 2003.

[GBCS07] Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. Thread-modular shape analysis. In *Prog. Language Design and Implementation, PLDI'07*. ACM, 2007.

[GFYS07] Guy Gueta, Cormac Flanagan, Eran Yahav, and Mooly Sagiv. Cartesian partial-order reduction. In *SPIN'07: Model Checking Software*, volume 4595 of *LNCS*, 2007.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. 1996.

[Gra91] P. Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT'91*, volume 493 of *LNCS*, 1991.

[HJMQ03] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular abstraction refinement. In *Computer Aided Verification, CAV'03*, volume 2725 of *LNCS*, 2003.

[HMG06] Nicolas Halbwachs, David Merchat, and Laure Gonnord. Some ways to reduce the space dimension in polyhedra computations. *Formal Methods in System Design*, 29(1), 2006.

[HPR97] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2), August 1997.

[Jea] B. Jeannet. The ConcurInterproc interprocedural analyzer for concurrent programs. `http://pop-art.inrialpes.fr/interproc/concurinterprocweb.cgi`.

[JLRS04] B Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *Static Analysis Symposium, SAS'04*, volume 3148 of *LNCS*, 2004.

[JM] B. Jeannet and A. Miné. The APRON abstract domains library. `http://apron.cri.ensmp.fr/library/`.

[JS04] B. Jeannet and W. Serwe. Abstracting call-stacks for interprocedural verification of imperative programs. In *Int. Conf. on Algebraic Methodology and Software Technology, AMAST'04*, volume 3116 of *LNCS*, 2004.

[KS92] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Compiler Construction, CC'92*, volume 641 of *LNCS*, 1992.

[LTKR08] Akash Lal, Tayssir Touili, Nicholas Kidd, and Thomas W. Reps. Interprocedural analysis of concurrent programs under a context bound. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08*, LNCS, 2008.

[Min06] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1), 2006.

[PST07] Gaël Patin, Mihaela Sighireanu, and Tayssir Touili. Spade: Verification of multithreaded dynamic and recursive programs. In *Computer Aided Verification, CAV'07*, volume 4590 of *LNCS*, 2007.

[QR05] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'05*, volume 3440 of *LNCS*, 2005.

[QRR04] Shaz Qadeer, Sriram K. Rajamani, and Jakob Rehof. Summarizing procedures in concurrent programs. In *Principles of programming languages, POPL'04*. ACM, 2004.

[Ram00] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. on Programing Language and Systems*, 22(2), 2000.

[RHS95] Tom Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Principles of Prog. Languages, POPL'95*. ACM, 1995.

[Rin01] M. Rinard. Analysis of multihreaded programs. In *Static Analysis Symposium, SAS'01*, volume 2126 of *LNCS*, 2001.

[RSJM05] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, 58(1–2), 2005.

[Som] F. Somenzi. CUDD: Colorado University Decision Diagram Package. `ftp://vlsi.colorado.edu/pub`.

[SP81] M. Sharir and A. Pnueli. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7. Prentice-Hall, 1981.

[SRW02] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Prog. Languages and Systems*, 24(3), 2002.

[Tau06] Gadi Taubenfeld. *Synchronization Algorithms and Concurrent Programming.* Prentice Hall, 2006.

[Tou05] Tayssir Touili. Dealing with communication for dynamic multi-threaded recursive programs. In *Verification of Infinite-State Systems with Applications to Security, VISSAS'05*, NATO Series, 2005.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399