Precise Interprocedural Analysis in the Presence of Pointers to the Stack

Pascal Sotin, Bertrand Jeannet

Precise Interprocedural Analysis in the Presence of Pointers to the Stack

Pascal Sotin, Bertrand Jeannet

INRIA Grenoble – Rhône-Alpes

Abstract. In a language with procedures and pointers as parameters, an instruction can modify memory locations anywhere in the call-stack. The presence of such side effects breaks most generic interprocedural analysis methods, which assume that only the top of the stack may be modified. We present a method that addresses this issue, based on the definition of an equivalent local semantics in which writing through pointers has a local effect on the stack. Our second contribution in this context is an adequate representation of summary functions that models the effect of a procedure, not only on the values of its scalar and pointer variables, but also on the values contained in pointed memory locations. Our implementation in the interprocedural analyser PInterproc results in a verification tool that infers relational properties on the value of Boolean, numerical and pointer variables.

1 Introduction

Relational interprocedural analysis is a well-understood static analysis technique [7, 26, 20]. It consists in associating at each program point a *relation* between the input state and the current state of the current procedure, so that at the exit point of a procedure P one obtains its input/output *summary function* capturing the effect of a call to P. Interprocedural analysis is also a form of modular analysis that enables the analysis of recursive programs.

Applying it requires the ability to identify precisely the input context of a procedure in the program, that is, the relevant part of the call-context that influences the execution of the callee procedure, as well as the output context, that is, the part of the state-space that may be altered by the procedure. This might be more or less simple:

- it is simple for procedures taking integer parameters and returning integer results; summary functions are relations $R \subseteq \mathbb{Z}^n$
- if a procedure accesses and modifies global variables, these may be treated as implicit input/output parameters that are added to the explicit ones; this applies to procedures manipulating dynamically allocated objects, if the memory heap is viewed as a special global variable [25, 17].

This paper addresses the case where procedures might take pointer parameters referring to stack variables, as in Fig. 2. This occurs in C/C++ programs, and in a weaker way in Pascal and Ada languages through reference parameter passing.

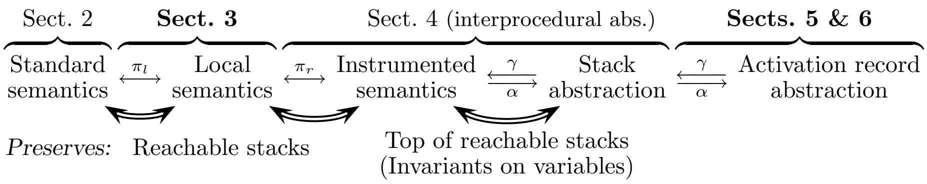


Fig. 1: Methodology followed in the paper.

Pointers to the execution stack raise two difficulties in this context:

(*i*) the effect of an instruction `*p=*p+1` might modify a location anywhere in the stack, instead of being located in the top activation record;

(*ii*) aliasing of different pointer expressions referring to the same location.

Point (*i*) makes difficult to isolate precisely the effective input context of a procedure, keeping in mind that filtering out the irrelevant parts of the call-context is important for modularity and thus efficiency. This has been addressed in [2] for the simpler case of references, and in [30] in for general pointers, but in a less general way that us. Point (*ii*) is a widely studied problem in compiler optimisation and program verification. Points-to or alias analysis have been widely studied [13]. However, most work target compiler optimisation and the precision achieved is insufficient for program verification. Array and shape analyses [11, 25, 17], which may be seen as sophisticated alias analyses, target automatic program parallelization or program verification, but focus mainly on arrays or heap objects, and much less on pointers to the stack in an interprocedural context. We also observe that many established static analyzers avoid this specific problem: Astrée [8] inlines on the fly procedure calls and does not perform an interprocedural analysis. This is also the case for Fluctuat [9]. Caduceus [10], before being embedded in Frama-C, explicitly discarded pointers on the stack. On the other hand, PolySpace Verifier [24] has necessarily a specific treatment for them, but the technique is unknown (unpublished).

Our goal is thus to enhance existing interprocedural analysis that infers invariants on the values of scalar variables with the treatment of pointers to the execution stack. Typically, we want to infer an enough precise summary function for the `swap` procedure, in order to infer the postcondition below:

$$\{0 \leq x \leq 2y \leq 10\}\ \texttt{swap(\&x,\&y)}\ \{0 \leq y \leq 2x \leq 10\}$$

Our approach is based on abstract interpretation [6]. We focus on pointers to the stack and we do not consider global variables, structured types and dynamic allocation; the combination with these features is discussed in the conclusion.

Our approach is original in several ways:

- our approach derives an effective analysis by decomposing it in the well-identified steps depicted on Fig. 1, and delays approximations on pointers and scalar variables to the last step, instead of mixing interprocedural and data abstractions;
- as a result, we perform in parallel an alias and a scalar analysis;

- we infer summary functions and invariance properties that are fully relational between alias properties (on pointers) and scalar properties (on Booleans and integers), and we define a symbolic representation for such properties;
- in the special case of Boolean programs, we prove our analysis to be exact.

Outline and contributions. Fig. 1 depicts graphically the methodology followed in the paper and emphasises (in bold font) our theoretical contributions. Sect. 2 describes the analysed language. Sect. 3 propose an alternative *local* semantics, in which the input context of a procedure is made explicit and the effect of a procedure is local (within the top of the activation stack). We prove this semantics to be equivalent to the standard one. Sect. 4 reminds how relational interprocedural analysis can be expressed as a stack abstraction [19], which allows us to formalize correctness proofs. It reduces the analysis on stacks to an analysis on activation records. Sect. 5 investigates the problem of representing efficiently sets and relations on activation records. Sect. 6 describes the abstraction of activation records with the BddApron library and shows an experiment with the PInterproc analyzer that we implemented. It discusses alternative abstract domains that lead to more classical analyses. Sect. 7 discusses related work and we draw some perspectives in Sect. 8.

2 The Language and its Standard Semantics

We extend with pointers (Tab. 1) the language analysed by Interproc [16], which features procedures with call-by-value parameter passing, local variables, numerical and boolean expressions, assignments, conditionals and loops, see Figs. 2, 7 and 9. Our language excludes structured types and dynamically allocated objects.

We consider types of the form $\tau_0 *^k = \tau_0 \overbrace{* \cdots *}^{k}$ and left values are of the form $*^k$ id, which can be found either on the left side of an assignment, or anywhere in expressions (like `**x + *y - 2`). We also have the nil pointer constant and the expression &id. Procedure calls have the form $\mathbf{y} = P(\mathbf{x})$, where $\mathbf{x}$ and $\mathbf{y}$ are the vectors of effective input and output parameters. Procedure definitions have the form

$$\begin{array}{l}\textbf{proc } P(fp_1 : t_1, \ldots, fp_m : t_m) : (fr_1 : t'_1, \ldots, fr_n : t'_n)\\ \textbf{var } z_1 : t''_1, \ldots, z_p : t''_p;\\ \textbf{begin} \ldots \textbf{end}\end{array}$$

where **fp** and **fr** are (vectors of) formal input and output parameters, **z** are the local variables, and **t**, **t'**, **t"** their associated type. We write $\mathbf{fp}^{(i)}$ for the ith component of the vector. The code of the full program is modelled with a Control Flow Graph (CFG), Fig. 2, in which an edge is either

- an intraprocedural edge $c \xrightarrow{\texttt{instr}} c'$ (test or assignment),
- a call edge $c \xrightarrow{\texttt{call } \mathbf{y}=P(\mathbf{x})} s$ linking a *call point* c (in the caller) to the *start point* s of the callee,
- or a return edge $e \xrightarrow{\texttt{call } \mathbf{y}=P(\mathbf{x})} r$ linking the *exit point* e of the callee procedure to a *return point* r of the caller.

The return point associated to a call point c will be denoted by $\mathrm{ret}(c)$.

τ	::=	τ_0
	|	$\tau*$
τ_0	::=	`bool`
	|	`int`

(a) Types.

left	::=	id	variable
	|	$*left$	dereferencing
expr	::=	*left*	read memory
	|	&id	address taking
	|	`nil`	nil pointer
	|	...	

(b) Expressions.

Table 1: Language extension

$$\begin{aligned} \overbrace{\langle n, F\rangle}^{\Gamma} &\in \mathit{Stack} = \mathbb{N} \times (\mathbb{N} \to \mathit{Act}) \\ \mathit{Act} &= \mathit{Var} \to \overbrace{\mathit{Addr} \cup \mathit{Scalar}}^{\mathit{Val}} \\ \mathit{Addr} &= (\mathbb{N} \times \mathit{Var}) \cup \{\mathit{nil}, \bot\} \\ \mathit{Scalar} &= \mathbb{B} \cup \mathbb{Z} \end{aligned}$$

Table 2: Semantic domains

```
proc swap(p:int*,q:int*) returns ()
var tmp:int;
begin           // (s)
    tmp = *p;
    *p = *q;
    *q = tmp;   // (e)
end
var a:int,b:int,
    x:int*,y:int*;
begin
  x=&a; y=&b; // (c1)
  swap(x,y);  // (r1)=(c2)
  swap(y,x);  // (r2)
end
```

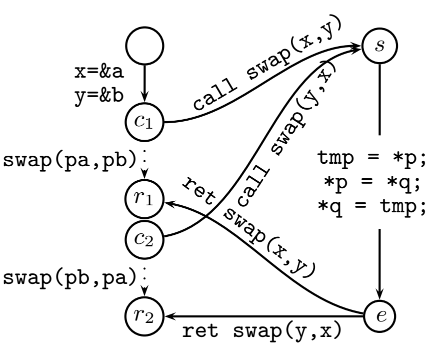


Fig. 2: Program example and Control Flow Graph (CFG)

Semantics. The semantic domains are given on Tab. 2. A program state is a stack of activation records. A stack $\Gamma = \langle n, F\rangle$ is defined by its size n and a function $F : [1..n] \to \mathit{Act}$ that returns the activation record at the given index. An activation record contains the current control point encoded in a `pc` variable and the values of the local variables. A pointer value is either the null value *nil*, the special value $\bot$ denoting a pending pointer, or a normal address (n, id), referring to the variable id located at the index n. A pending pointer value $\bot$ typically occurs when a callee procedure returns a pointer to one of its local variables. We adopt the following alternative view of the stack:

$$\mathit{Stack} = \mathbb{N} \times \big(\mathbb{N} \to (\overbrace{\mathit{Var} \to \mathit{Val}}^{\mathit{Act}})\big) \simeq \mathbb{N} \times \big((\mathbb{N} \times \mathit{Var}) \to \mathit{Val}\big)$$

Tab. 3 defines the semantics of expressions. The semantics of a nil or a pending pointer dereference is undefined.

The semantics of the language is given as a transition system $(\mathit{Stack}, I, \rightsquigarrow)$, where $\rightsquigarrow$ is a transition relation on stacks and I is the set of initial stacks of height one. Tab. 4 defines $\rightsquigarrow$ for the interprocedural edges of the CFG. The call copies the effective parameters in the formal parameters. It also initializes the local variables of pointer type to *nil*. The return copies the formal results in the effective results, then it forgets the last activation record. Pointers to addresses of this activation record are turned to pending values. Intraprocedural edges generate simpler transitions, like choosing the next control point for tests, and

$$
\begin{array}{rclr}
[\![\mathit{left}]\!]^{\mathcal{A}} & : & \mathit{Stack} \rightarrow \mathbb{N} \times \mathit{Var} & \text{Address of a left value} \\
[\![\mathit{expr}]\!]^{\mathcal{V}} & : & \mathit{Stack} \rightarrow (\mathbb{N} \times \mathit{Var}) \cup \{\mathit{nil}, \bot\} \cup \mathit{Scalar} & \text{Value of an expression} \\
[\![\mathrm{id}]\!]^{\mathcal{A}} & = & \langle n, \mathrm{id} \rangle & [\![\mathit{left}]\!]^{\mathcal{V}} \ = \ F([\![\mathit{left}]\!]^{\mathcal{A}}) \\
[\![{*}\mathit{left}]\!]^{\mathcal{A}} & = & [\![\mathit{left}]\!]^{\mathcal{V}} & [\![\&\mathrm{id}]\!]^{\mathcal{V}} \ = \ [\![\mathrm{id}]\!]^{\mathcal{A}} \\
 & & \text{when } [\![\mathit{left}]\!]^{\mathcal{V}} \notin \{\mathit{nil}, \bot\} & [\![\mathtt{nil}]\!]^{\mathcal{V}} \ = \ \mathit{nil} \\
 & & & \text{(The argument } \Gamma = \langle n, F \rangle \text{ is implicit)}
\end{array}
$$

Table 3: Standard semantics: expressions.

$$
\begin{array}{c}
F(n, \mathtt{pc}) = c \quad F'(n{+}1, \mathtt{pc}) = c' \\
\forall i, \qquad F'(n{+}1, \mathbf{fp}^{(i)}) = F(n, \mathbf{x}^{(i)}) \\
\forall z \in \mathit{Var}_{\tau *} \backslash \mathbf{fp}, \quad F'(n{+}1, z) = \mathit{nil} \\
\forall z \in \mathit{Var}_{\tau_0} \backslash \mathbf{fp}, \quad F'(n{+}1, z) \in \mathit{Scalar} \\
\forall z \in \mathit{Var}, \forall k \leq n, \quad F'(k, z) = F(k, z) \\
\hline
\langle n, F \rangle \rightsquigarrow \langle n{+}1, F' \rangle
\end{array}
\qquad (\text{Call } c \xrightarrow{\mathtt{call}\ \mathbf{y}{:=}P(\mathbf{x})} c')
$$

$$
\begin{array}{c}
F(n{+}1, \mathtt{pc}) = c \\
F' = F \begin{bmatrix} \langle n, \mathtt{pc} \rangle \ \mapsto c' \\ \langle n, \mathbf{y}^{(i)} \rangle \mapsto F(n{+}1, \mathbf{fr}^{(i)}) \end{bmatrix} \\
F'' = F'[\ a \mapsto \bot \ \mathit{if}\ F'(a) = \langle n{+}1, \mathrm{id} \rangle \ ] \\
\hline
\langle n{+}1, F \rangle \rightsquigarrow \langle n, F'' \rangle
\end{array}
\qquad (\text{Ret } c \xrightarrow{\mathtt{ret}\ \mathbf{y}{:=}P(\mathbf{x})} c')
$$

Table 4: Standard semantics: transitions

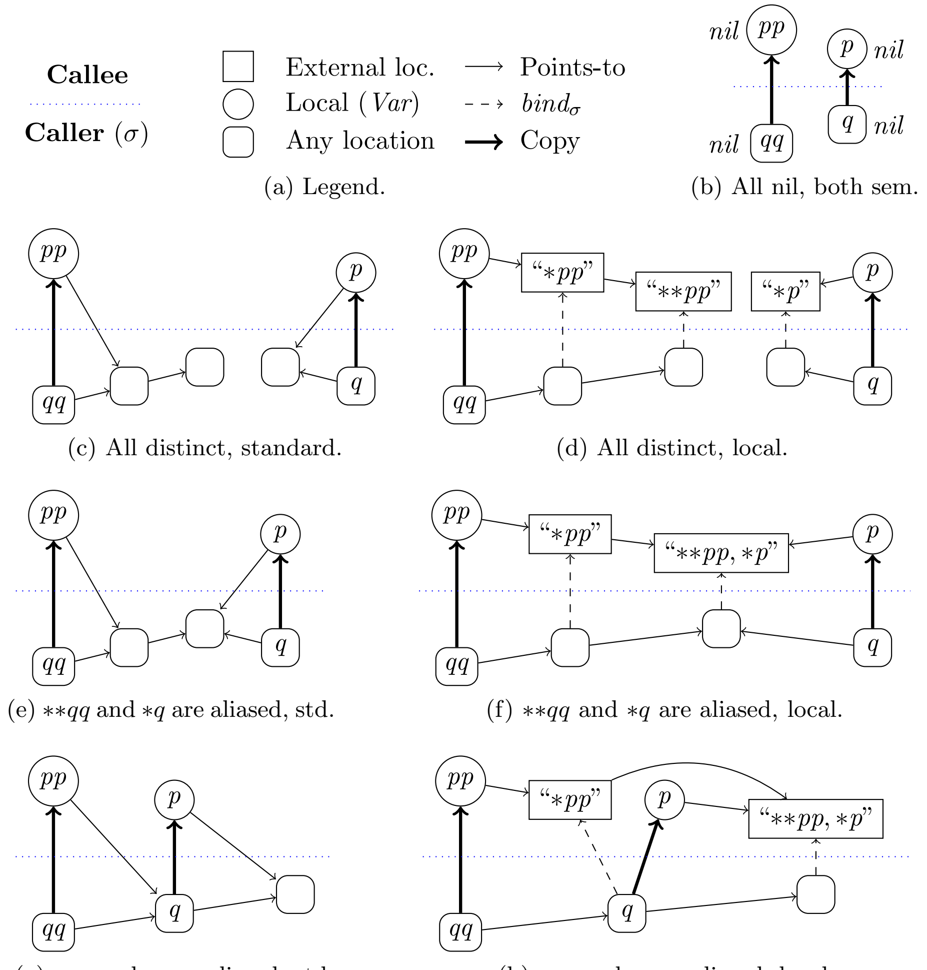
updating the stack for assignments. In this semantics, undefinedness (comparison with a pending value or invalid dereference) generates a sink state without successors.

Analysis goal. Our aim is to perform a reachable state analysis of such programs and more precisely to compute for each program point an invariant on the values of variables. Formally, the set of reachable stacks is $\mathit{Reach}(I, \rightsquigarrow) = \{\Gamma \mid \exists \Gamma_0 \in I, \Gamma_0 \rightsquigarrow^* \Gamma\}$, and our goal is to compute the set of top activation records of these stack.

We want to apply relational interprocedural analysis methods for this purpose. As discussed in the introduction, this requires to identify the input context of a procedure, which may include in our case the content of a pointed location anywhere in the stack, which may be later modified by the procedure during its execution. In contrast, general interprocedural analysis techniques [20] assume that side-effects performed by a procedure are limited to the current activation record.

3 An Equivalent Local Semantics

The aim of this section is to define a *local* semantics in which the effect of a procedure is limited to the top activation record. The idea of this semantics, in-



(a) Legend.

(b) All nil, both sem.

(c) All distinct, standard.

(d) All distinct, local.

(e) $**qq$ and $*q$ are aliased, std.

(f) $**qq$ and $*q$ are aliased, local.

(g) $*qq$ and q are aliased, std.

(h) $*qq$ and q are aliased, local.

Fig. 3: Procedure call in the standard and local semantics: call `f(qq,q)` to a procedure `f(int** pp, int* p)`.

spired by [2], is that a procedure works on local copies (called *external locations*) of the locations that it can reach with its pointer parameters.

The first challenge is to take into account aliasing properties between pointer parameters and to define a correct input parameter passing mechanism. Consider a call `f(qq,q)` to a procedure `f(int** pp, int* p)`. Fig. 3 considers different aliasing situation between `qq` and `q` in the caller, and the consequences on the set of external locations in the callee. We depict the situation on the left for the standard semantics, on the right for the local semantics. We will define a function *bind*, depicted with dashed arrows, that maps locations in the caller to external locations in the callee that are reachable from its input parameters.

Figs. 3(g)(h) illustrates a non trivial situation: in the caller, `qq` points to `q`, but in the callee `pp` does not point to `p`.

The second point is then to define a correct return parameter passing mechanism. When the procedure `f` returns, the modifications on its external locations should be propagated back to the corresponding locations in the caller, which may be themselves local or external w.r.t. their own caller.

3.1 Local Semantics

Tab. 5 defines the semantic domains. Values can be stored as before in local variables, but also in external locations. These external locations are identified by the set of left values that refer to them at the beginning of the current procedure. Fig. 3 illustrates this naming scheme.

$$
\begin{aligned}
\mathit{\Gamma} \in \mathit{Stack} &= \mathit{Act}^{\star} \\
\sigma \in \mathit{Act} &= \mathit{Loc} \to \overbrace{\mathit{Loc} \cup \{\mathit{nil}, \bot\} \cup \mathit{Scalar}}^{\mathit{Val}} \\
l \in \mathit{Loc} &= \mathit{Var} \cup \mathit{External} \\
\mathit{External} &= \mathcal{P}(\mathit{Deref}) \\
\mathit{Deref} &= \{*^{k}\mathbf{fp}^{(i)}\}
\end{aligned}
$$

Table 5: Semantic domains

The evaluation of the expressions are the same as in Tab. 1b, except that left values are now all fetched in the top activation record σ. We have $[\![\mathit{left}]\!]^{\mathcal{V}}_{\sigma} = \sigma([\![\mathit{left}]\!]^{\mathcal{A}}_{\sigma})$ and $[\![\mathrm{id}]\!]^{\mathcal{A}}_{\sigma} = \mathrm{id}$. The semantics of an interprocedural instruction is captured by a relation $R_{\mathrm{instr}}(\sigma, \sigma')$ between two top activation records.

Binding. The key point of the local semantics lies in the procedure calls and returns. The external locations (*ie.* local copies) have to be determined and valued at call time, then propagated back at return time.

The purpose of the function bind_{σ} is to map locations of the caller that can be reached by effective parameters to the external locations in the callee:

$$
\begin{aligned}
&\mathit{bind}_{\sigma} : \mathit{Loc}_{(\mathrm{Caller})} \to \mathit{External}_{(\mathrm{Callee})} \\
&\mathit{bind}_{\sigma}(l) = D \quad \text{if} \quad D = \left\{ *^{k}\,\mathbf{fp}^{(i)} \mid [\![*^{k}\mathbf{x}^{(i)}]\!]^{\mathcal{V}}_{\sigma} = l \wedge k \geq 1 \right\} \neq \emptyset
\end{aligned}
\qquad (1)
$$

with $\sigma \in \mathit{Act}_{(\mathrm{Caller})}$ being the activation record at the call point. The function $\mathit{bind}_{\sigma}(l)$ binds a location l of the caller to the set of dereferences in the callee that can refer to it at call time. If this set is empty, $\mathit{bind}_{\sigma}(l)$ is undefined and l cannot be modified by the callee. The constraint $k \geq 1$ reflects the fact that modifications of the formal parameters do not alter the effective parameters (call-by-value). This function is injective and can be reversed into the function $\mathit{bind}_{\sigma}^{-1}$. Fig. 3 depicts bind_{σ} for different contexts σ.

Procedure calls and returns. Tab. 6 formalizes the transitions, and Fig. 4 illustrates the relationships between the involved sets and functions.

During a procedure call, a new activation record σ' is pushed on the stack, and initialized with the adequate values in the caller (parameters, external locations). The return operation first propagates the side effects by copying back the externals, then copies the return values and pops the activation record. The copies between caller and callee rely on the functions $\widetilde{\mathit{bind}_{\sigma}}$ and $\widetilde{\mathit{bind}_{\sigma}}^{-1}$ which take care of address conversion.

$$\widetilde{bind_\sigma}(v) = \begin{cases} bind_\sigma(v) & \text{if } v \in \mathit{Loc} \\ v & \text{otherwise} \end{cases} \qquad \text{(Pass)}$$

$$\widetilde{bind_\sigma^{-1}}(v) = \begin{cases} bind_\sigma^{-1}(v) & \text{if } v \in \mathcal{P}(\mathit{Deref}) \\ \bot & \text{if } v \in \mathit{Var} \\ v & \text{otherwise} \end{cases} \qquad (\text{Pass}^{-1})$$

$$\frac{R^{+}_{\mathbf{y}:=P(\mathbf{x})}(c)(\sigma,\sigma') \begin{cases} \quad \sigma(\mathtt{pc}) = c \wedge \sigma'(\mathtt{pc}) = c' \\ \forall i, & \sigma'(\mathbf{fp}^{(i)}) = \widetilde{bind_\sigma}(\sigma(\mathbf{x}^{(i)})) \\ \forall e \in dom(bind_\sigma^{-1}), & \sigma'(e) = \widetilde{bind_\sigma} \circ \sigma \circ bind_\sigma^{-1}(e) \\ \forall z \in \mathit{Var}_{\tau*} \backslash \mathbf{fp}, & \sigma'(z) = \mathit{nil} \\ \forall z \in \mathit{Var}_{\tau_0} \backslash \mathbf{fp}, & \sigma'(z) \in \mathit{Scalar} \end{cases}}{\mathit{\Gamma}.\sigma \to \mathit{\Gamma}.\sigma.\sigma'} \qquad (\text{CallL } c \xrightarrow{\texttt{call } \mathbf{y}=P(\mathbf{x})} c')$$

$$\frac{R^{-}_{\mathbf{y}:=P(\mathbf{x})}(c)(\sigma,\sigma',\sigma'') \begin{cases} \quad \sigma(\mathtt{pc}) = c \wedge \sigma'(\mathtt{pc}) = c' \wedge \sigma''(\mathtt{pc}) = \mathrm{ret}(c) \\ \sigma''_{\text{side}} = \sigma \left[l \mapsto \widetilde{bind_\sigma^{-1}} \circ \sigma' \circ bind_\sigma(l) \;\middle|\; l \in dom(bind_\sigma) \right] \\ \sigma'' = \sigma''_{\text{side}} \left[\mathbf{y}^{(i)} \mapsto \widetilde{bind_\sigma^{-1}}(\sigma'(\mathbf{fr}^{(i)})) \right] \end{cases}}{\mathit{\Gamma}.\sigma.\sigma' \to \mathit{\Gamma}.\sigma''} \qquad (\text{RetL } c' \xrightarrow{\texttt{ret } \mathbf{y}=P(\mathbf{x})} \mathrm{ret}(c))$$

Table 6: Local semantics: transitions.

.

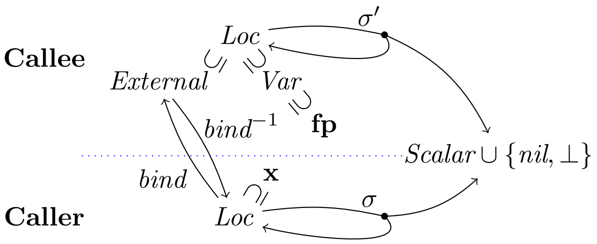
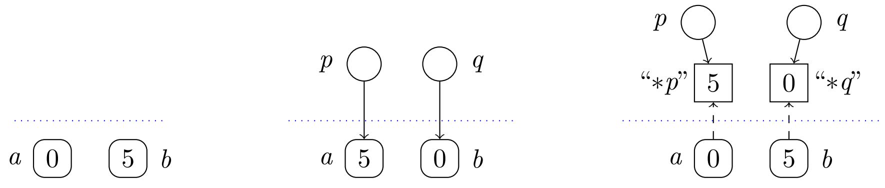


Fig. 4: Binding.



(a) Before call. (b) Before return, standard. (c) Before return, local.

Fig. 5: Standard and local stack before returning from `swap(&a,&b)`.

3.2 Preservation of properties by the local semantics

Proving that the standard and local semantics are equivalent w.r.t. reachability properties raises a technical difficulty: side-effects due to pointers to the stacks are propagated immediately in the standard semantics, whereas this propagation is delayed until procedure return in the local semantics, as illustrated on Fig.5.

To deal with this, we define a function that "projects" a local stack into a standard stack and takes care of the above-mentioned propagation. We first define a function $\mathit{address} : \mathbb{N} \times \mathit{Loc} \rightarrow \mathbb{N} \times \mathit{Var}$ that returns the original variable in the stack referred to by a location at the i^{th} activation record in a local stack $\sigma_1 \dots \sigma_n$:

$$\mathit{address}(i,l) = \begin{cases} \mathit{address}(i-1, \mathit{bind}^{-1}_{\sigma_{i-1}}(l)) & \text{if } l \in \mathit{External} \\ \langle i,l \rangle & \text{if } l \in \mathit{Var} \end{cases}$$

This function is then generalized to values:

$$\widetilde{\mathit{address}}(i,v) = \begin{cases} \mathit{address}(i,v) & \text{when } v \in \mathit{Loc} \\ v & \text{when } v \in \{\mathit{nil}, \bot\} \cup \mathit{Scalar} \end{cases}$$

We last define a function that assigns values to variables in a standard stack, possibly by searching the external location representing it at the highest index in the local stack:

$$\mathit{value}(i,z) = \begin{cases} \mathit{value}(i+1, \mathit{bind}_{\sigma_i}(z)) & \text{if } z \in \mathit{dom}(\mathit{bind}_{\sigma_i}) \wedge i \neq n \\ \widetilde{\mathit{address}}(i, \sigma_i(z)) & \text{otherwise} \end{cases}$$

Definition 1. *The projection function* $\pi_l : \mathit{Stack}_{ins} \rightarrow \mathit{Stack}_{std}$ *is defined by*

$$\pi_l(\sigma_1 \dots \sigma_n) = \langle n, F \rangle \text{ where } F(i,z) = \mathit{value}(i,z)$$

π_l is not injective because a local stack keeps track of past values of variables when these are copied in external locations. For instance, if one considers the local stack on Fig. 5c, modifying the value of location a does not modify its image by π.

Thm. 1 states that both transition systems behave the same way (proof in appendix). We can thus compute the exact reachability set in the standard semantics by computing it in the local semantics and projecting the result (Cor. 1).

Theorem 1. $\pi_l(i) = s \Rightarrow \begin{cases} \forall s', s \rightsquigarrow s' \Rightarrow \exists i', i \rightarrow i' \wedge \pi_l(i') = s' \\ \forall i', i \rightarrow i' \Rightarrow \exists s', s \rightsquigarrow s' \wedge \pi_l(i') = s' \end{cases}$

Corollary 1. $\mathit{Reach}(I, \rightsquigarrow) = \pi_l(\mathit{Reach}(I, \rightarrow))$

4 Interprocedural Abstraction

The previous section defined a local semantics in which side-effects involve only the top of the stack, so as to enable the application of relational interprocedural analysis techniques, which manipulate *relations* between activation records (but

$$\frac{\begin{array}{c}\langle \text{return } \mathbf{y}\,\texttt{:=}P_j(\mathbf{x})\rangle \\ X_c(\sigma_i^0,\ \sigma_i) \quad \boxed{X_e(\sigma_j^0,\ \sigma_j)} \\ \text{(call point)} \quad \text{(exit point)} \\ R^+_{\mathbf{y}\texttt{:=}P_j(\mathbf{x})}(c)(\sigma_i,\sigma_j^0) \\ \text{(unification of actual and formal parameters)} \\ R^-_{\mathbf{y}\texttt{:=}P_j(\mathbf{x})}(c)(\sigma_i,\sigma_j',\ \sigma_i') \\ \text{(output parameter passing)}\end{array}}{\boxed{X_r(\sigma_i^0,\ \sigma_i')} \quad \text{(return point)}}$$

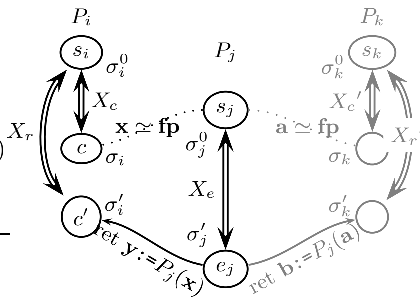


Fig. 6: Procedure return in relational interprocedural analysis. The relation X_r at return point is obtained by a (special) composition of relations X_c and X_e.

$$\frac{\Gamma.\sigma \to \Gamma.\sigma'}{\Upsilon.\langle\sigma^0,\sigma\rangle \hookrightarrow \Upsilon.\langle\sigma^0,\sigma'\rangle} \quad \text{(IntraI)}$$

$$\frac{\Gamma.\sigma \to \Gamma.\sigma.\sigma'}{\Upsilon.\langle\sigma^0,\sigma\rangle \hookrightarrow \Upsilon.\langle\sigma^0,\sigma\rangle.\langle\sigma',\sigma'\rangle} \ \text{(CallI)} \qquad \frac{\Gamma.\sigma.\sigma' \to \Gamma.\sigma''}{\Upsilon.\langle\sigma^0,\sigma\rangle.\langle\sigma^{0'},\sigma'\rangle \hookrightarrow \Upsilon.\langle\sigma^0,\sigma''\rangle} \ \text{(RetI)}$$

Table 7: Relational instrumentation for interprocedural analysis

not relations between stacks). In this section, we formalize a relational interprocedural analysis based on the local semantics.

As explained in the introduction, relational interprocedural analysis associates at each program point a *relation* between the input state and the current state of the current procedure, so that the exit point of a procedure contains its input/output *summary*. Fig. 6 illustrates the use of the summary X_e which captures the effect of a call to P_j to obtain the relation X_r at the return point.

We follow the formalization of [19], that reformulates classical presentations [7, 26, 20] by deriving relational interprocedural analysis as an abstract interpretation of the concrete (local effects) semantics. The advantage of this formalization is twofold:

- it allows to derive automatically abstract transfer functions and to prove their correctness; this is not obvious when the input and output parameter passing mechanisms are as complex as in Tab. 6; for instance [20] does not investigate this issue.
- it separates the abstraction made by the interprocedural analysis method from the abstraction performed on activation records, as depicted on Fig. 1.

Relational instrumentation. Establishing a relation between the input and the current state at any point of a procedure requires to memorize the input state. We thus consider an semantics on instrumented pairs $(\sigma^0, \sigma) \in Act^2$ where σ_0 is the input state. Alternatively (as in [19]), it can be seen as introducing copies $\mathbf{fp}_0$ and $\mathbf{l}_0$ of the formal parameters and external locations in σ.

$post^\sharp(\tau) : \mathcal{P}(Act^2) \to \mathcal{P}(Act^2)$ defined by $Y^\sharp = post^\sharp(\tau)(X^\sharp)$ with

$$\tau = c \xrightarrow{\texttt{instr}} c' \qquad Y^\sharp = \left\{ \langle \sigma^0, \sigma' \rangle \,\middle|\, \begin{array}{c} \langle \sigma^0, \sigma \rangle \in X^\sharp \\ R_{\texttt{instr}}(c, c')(\sigma, \sigma') \end{array} \right\} \tag{2}$$

$$\tau = c \xrightarrow{\texttt{call } \mathbf{y}=P_j(\mathbf{x})} s_j \qquad Y^\sharp = \left\{ \langle \sigma_j, \sigma_j \rangle \,\middle|\, \begin{array}{c} \langle \sigma^0, \sigma \rangle \in X^\sharp \\ R^+_{\mathbf{y}=P(\mathbf{x})}(c)(\sigma, \sigma_j) \end{array} \right\} \tag{3}$$

$$\tau = e_j \xrightarrow{\texttt{ret } \mathbf{y}=P_j(\mathbf{x})} \mathrm{ret}(c) \quad Y^\sharp = \left\{ \langle \sigma^0, \sigma' \rangle \,\middle|\, \begin{array}{c} \langle \sigma^0, \sigma \rangle \in X^\sharp \wedge \langle \sigma^0_j, \sigma_j \rangle \in X^\sharp \\ R^+_{\mathbf{y}=P(\mathbf{x})}(c)(\sigma, \sigma^0_j) \\ R^-_{\mathbf{y}=P(\mathbf{x})}(c)(\sigma, \sigma_j, \sigma') \end{array} \right\} \tag{4}$$

Table 8: Abstract postcondition defining a forward semantics on activation records.

Formally, this *relational instrumentation* is defined from any local transition system $(Act^*, I, \to)$ by a transition system $((Act \times Act)^*, I', \hookrightarrow)$ where $I' = \{(\sigma, \sigma) \mid \sigma \in I\}$ and $\hookrightarrow$ is defined by the rules of Tab. 7. It is clear that there is a one-to-one correspondence between the executions of the transition systems $\to$ and $\hookrightarrow$. The second important point is that all stacks reachable by $\hookrightarrow$ are coherent stacks (see Definition 2).

Definition 2 (Coherent stack). *Given a local semantics* $(Act^*, I, \to)$*, an instrumented stack* $\Upsilon = \langle \sigma^0_1, \sigma_1 \rangle \ldots \langle \sigma^0_n, \sigma_n \rangle$ *in* $(Act^2)^\star$ *is* coherent *if* $\forall i < n : R^+(\sigma_i, \sigma^0_{i+1})$*, where* $R^+(\sigma, \sigma') = \exists \Gamma : \Gamma.\sigma \to \Gamma.\sigma.\sigma'$.

Reachable stacks are coherent because initial stacks are so, and this property is preserved by all transitions in Tab. 7. If $R^+(\sigma, \sigma')$, we say that σ is a *valid call-context* for σ'. We write $\mathcal{C}((Act^2)^\star)$ for the set of coherent stacks.

Stack abstraction and induced forward semantics. The stack abstraction consists in collapsing sets of stacks into sets of activation records, and conversely in using the coherence property to rebuild stacks. We define the Galois connection $\mathcal{P}(\mathcal{C}((Act^2)^\star)) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{P}(Act^2)$ with:

$$\alpha(X) = \{\upsilon_i \mid \upsilon_1 \cdots \upsilon_n \in X \wedge 1 \le i \le n\} \tag{5}$$

$$\gamma(X^\sharp) = \left\{ \upsilon_1 \cdots \upsilon_n \,\middle|\, \begin{array}{l} \forall 1 \le i \le n, \upsilon_i \in X^\sharp \\ \upsilon_1 \cdots \upsilon_n \text{ is coherent} \end{array} \right\} \tag{6}$$

Let $post(c \xrightarrow{\texttt{instr}} c') : \mathcal{P}(\mathcal{C}((Act^2)^\star)) \to \mathcal{P}(\mathcal{C}((Act^2)^\star))$ be the concrete postcondition operator associated to a CFG edge, which can be deduced from Tabs. 6 and 7. The induced abstract postcondition is defined in Tab. 8. Notice that Eqn. (4) reformulates the rule of Fig. 6; the condition $R^+_{\mathbf{y}=P(\mathbf{x})}(c)(\sigma, \sigma^0_j)$ tells that σ is valid context for σ^0_j.

Proposition 1 ($\boldsymbol{post^\sharp}$ is a correct approximation of $\boldsymbol{post}$).
For any τ, $post^\sharp(\tau) \circ \alpha \sqsupseteq \alpha \circ post(\tau)$.
If τ is not a return transition, then $post^\sharp(\tau) \circ \alpha = \alpha \circ post(\tau)$.

Not surprisingly, the transfer function is less precise for return transitions. However, generalizing a result of [19] we get the following optimality result that reformulates in our setting the Interprocedural Coincidence Theorem of [20].

Theorem 2. $lfp(\lambda X^\sharp \,.\, \alpha(I') \sqcup post^\sharp(X^\sharp)) = \alpha(Reach(I, \hookrightarrow)) = hd(Reach(I, \hookrightarrow))$

This means that as far as we are interested in invariants at each control point, which concern only top activation records of reachable stacks, the stack abstraction is exact.

5 Representing Sets and Relations of Activation Records

Let us remind the steps we followed, depicted in Fig. 1: we defined in Sect. 3 a local semantics, to which we apply the stack abstraction defined in Sect. 4, so as to obtain a reachability analysis on sets of activation records. In this section, we discuss the encoding of activation records by functions of signature $Id \to \mathbb{B} \cup \mathbb{E} \cup \mathbb{Z}$, and their symbolic manipulation with logical formula ($\mathbb{E}$ denotes enumerated values). This is a first step toward an abstraction leading to an effective implementation.

External locations and pointers. The set of external locations appears both in the domain and the codomain of activation records σ, see Tab. 5. Two facts are important:

(1) the set of *potential* external locations $External = \mathcal{P}(\{*^k fp^{(i)}\})$ is finite;
(2) the set of *active* external locations depends through the function $bind_\sigma$ on the call-context σ, see Fig. 3, and it is much smaller.

(1) implies that the domain of σ is finite and that the value of pointers belongs to a finite set. (2) comes from two properties: for a given call-context σ,

- typing forbids some subsets: all elements of $bind_\sigma(l) \in External = \mathcal{P}(Deref)$ represent aliased dereferences and thus have the same type;
- for two locations $l_1 \neq l_2$, $bind_\sigma(l_1)$ and $bind_\sigma(l_2)$ are disjoints.

We thus pick a representative for each set with a function $repr : \mathcal{P}(Deref) \to Deref$ defined by $repr(D) = \min_\preceq D$ with $*^{k_1} fp^{(i_1)} \preceq *^{k_2} fp^{(i_2)} \Leftrightarrow i_1 \leq i_2$. The order $\preceq$ is a total order on D because of the above-mentioned typing property

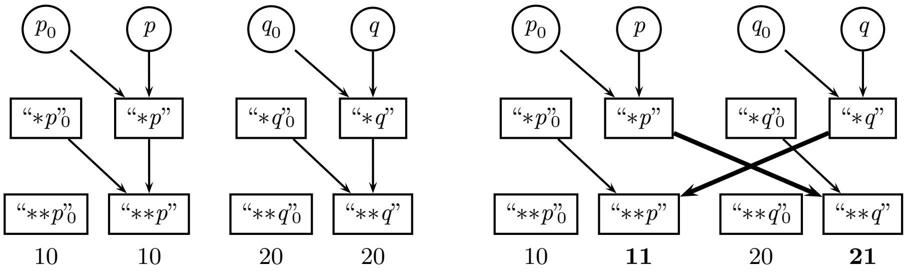
As a result, an activation record can be represented with a function of signature $Id \to \mathbb{B} \cup \mathbb{E} \cup \mathbb{Z}$ with $Id = Var \cup External$ and $\mathbb{E} = \{nil\} \cup Var \cup External$.

Concerning the number of external locations, in a procedure of signature $(fp^{(1)} : \tau_0 *^{k_1}, \ldots, fp^{(n)} : \tau_0 *^{k_n})$ in which all scalars pointed by formal parameters have the same type τ_0, we have thus $\sum_{1 \leq i \leq n} k_i$ external locations, which are all active if there is no aliasing at all. If there are several scalar types involved, we sum up the sums associated to each scalar type.

```
proc swapi(p:int**,q:int**)
begin
  **p = **p + 1;
  **q = **q + 1;
  (*p,*q) = (*q,*p);
end
```

$$\begin{gathered}\texttt{p} = \texttt{p}_0 = \&``{*}\texttt{p}" \wedge ``{*}\texttt{p}"_0 = \&``{*}{*}\texttt{p}" \wedge ``{*}{*}\texttt{p}"_0 = 10 \wedge \\ \texttt{q} = \texttt{q}_0 = \&``{*}\texttt{q}" \wedge ``{*}\texttt{q}"_0 = \&``{*}{*}\texttt{q}" \wedge ``{*}{*}\texttt{q}"_0 = 20 \wedge \\ ``{*}\texttt{p}" = \&``{*}{*}\texttt{q}" \wedge ``{*}\texttt{q}" = \&``{*}{*}\texttt{p}" \wedge \\ ``{*}{*}\texttt{p}" = ``{*}{*}\texttt{p}"_0 + 1 \wedge ``{*}{*}\texttt{q}" = ``{*}{*}\texttt{q}"_0 + 1\end{gathered}$$

(a) At exit point: logical representation



(b) At start point

(c) At exit point

Fig. 7: Duplication of locations and examples of a summary function.

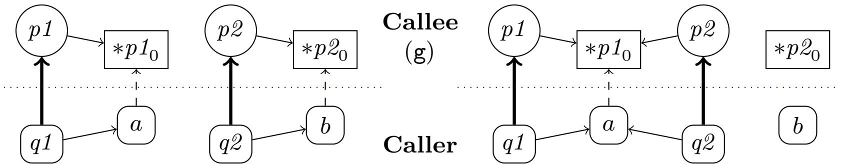
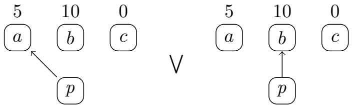


Fig. 8: Distinct aliasing, distinct bindings.

Representing sets. Our logical formula will use the following atoms:

$p = \&\texttt{x}$	p points to variable `x`	$l_1 = l_2$	l_1 and l_2 have same value
$p = \&l$	p points to location l	$l = 7$	location l contains scalar 7

For example, we can describe the set of activation records



with $X = (\texttt{p} = \&\texttt{a} \vee \texttt{p} = \&\texttt{b}) \wedge \texttt{a} = 5 \wedge \texttt{b} = 10 \wedge \texttt{c} = 0$

Instrumented activation records. We actually represent pairs $\langle \sigma^0, \sigma \rangle$ of activation records in Tab. 8 with single activation records containing copies $\mathbf{fp}_0$ and $\mathbf{l}_0$ of formal parameters and external locations. Fig. 7 illustrates this point. Fig. 7b depicts the activation record at the start point of the procedure, and Fig. 7c shows the modified values (in bold lines) at the exit point. Fig. 7a represent it as a logical formula. We remind that keeping the values of locations $\mathbf{fp}_0$ and $\mathbf{l}_0$ will allow to select valid calling context for procedure return, see Fig. 6.

Representing relations. In Tab. 8, postconditions associated to call and return are based on the relation $R^{+}(\sigma, \sigma')$ defined in Tab. 6 that relates a call-context to the initial activation record in the callee. We illustrate the transcription of this relation with a quantifier-free logical formula. Consider the function `g(int* p1, int* p2)` and a call instruction `g(q1,q2)` in the context depicted on Fig. 8 where `q1` and `q2` can point to `a` and/or `b`, see Fig. 8. We have

$$
\begin{aligned}
R^{+}_{\mathrm{g(q1,q2)}} = {} & (\texttt{q1} = \mathit{nil} && \Leftrightarrow \texttt{p1} = \mathit{nil}) \\
\wedge {} & (\texttt{q1} = \bot && \Leftrightarrow \texttt{p1} = \bot) \\
\wedge {} & (\texttt{q1} = \&\texttt{a} && \Leftrightarrow (\texttt{p1} = \&\text{``}{*}\texttt{p1}\text{''} \wedge \text{``}{*}\texttt{p1}\text{''} = \texttt{a})) \\
\wedge {} & (\texttt{q1} = \&\texttt{b} && \Leftrightarrow (\texttt{p1} = \&\text{``}{*}\texttt{p1}\text{''} \wedge \text{``}{*}\texttt{p1}\text{''} = \texttt{b})) \\
\wedge {} & (\texttt{q2} = \mathit{nil} && \Leftrightarrow \texttt{p2} = \mathit{nil}) \\
\wedge {} & (\texttt{q2} = \bot && \Leftrightarrow \texttt{p2} = \bot) \\
\wedge {} & ((\texttt{q2} \neq \texttt{q1} \wedge \texttt{q2} = \&\texttt{a}) && \Leftrightarrow (\texttt{p2} = \&\text{``}{*}\texttt{p2}\text{''} \wedge \text{``}{*}\texttt{p2}\text{''} = \texttt{a})) \\
\wedge {} & ((\texttt{q2} \neq \texttt{q1} \wedge \texttt{q2} = \&\texttt{b}) && \Leftrightarrow (\texttt{p2} = \&\text{``}{*}\texttt{p2}\text{''} \wedge \text{``}{*}\texttt{p2}\text{''} = \texttt{b})) \\
\wedge {} & (\texttt{q2} = \texttt{q1} && \Leftrightarrow \texttt{p2} = \texttt{p1})
\end{aligned}
$$

When `q1` and `q2` are aliased ($\texttt{q2} = \texttt{q1}$), the value of "$*$`p2`" is unconstrained. The external location used is still named $*$`p1`, meaning that $\mathit{repr}(\{*\texttt{p1}, *\texttt{p2}\}) = *\texttt{p1}$. Generally speaking we have to enumerate the possible values of the actual parameter $q1, q2$, and in each case assigning the correct value for the formal parameters, according to Tab. 6.

Once sets of and relations on activation records are represented with such logical formula, it is possible to compute the application of the relation to a set or to perform relation composition.

6 Abstracting Sets of Activation Records

The forward semantics of Tab. 8 manipulates activation records, the structure of which has been investigated in the previous section. We begin by an important result:

> *For recursive Boolean programs with pointers on the stacks, we obtain an exact analysis w.r.t. invariance properties that can be implemented.*

This is a consequence of Theorems 1 and 2, and the observation that the state-space induced by activation records is finite in this case. However in the presence of numerical variables it is not any more the case, and we need to perform an abstraction. We start by describing the one we implemented in our tool, and we then discuss alternatives which also abstract pointers and lead to more classical analyses.

Logico-numerical abstraction with BddApron. BddApron [14] is a static analysis library that provides an abstract domain based on the following Galois connection:

$$\mathcal{P}(\mathit{Id} \to \mathbb{B} \cup \mathbb{E} \cup \mathbb{Z}) \underset{\alpha_{\mathcal{N}}}{\overset{\gamma_{\mathcal{N}}}{\leftrightarrows}} A = (\mathbb{B}^n \to \mathcal{N})$$

where Id is a finite set of identifiers, $\mathbb{E}$ is a set of user-defined enumerated types, and $\mathcal{N}$ is any abstract domain for numerical variables provided by the Apron library [18], for instance intervals or convex polyhedra. This abstraction treats finite-state expressions exactly and approximates the operations involving numerical tests and assignments. Abstract values are efficiently represented using MTBDDs with numerical abstract values in terminal nodes; we use for this the CUDD library [27].

As discussed in the previous section, an activation record can be encoded with a function $Id \rightarrow \mathbb{B} \cup \mathbb{E} \cup \mathbb{Z}$, thus sets of activations records can be effectively abstracted with this library. We implemented a version of the Interproc analyzer [18, 16] based on it, that analyses the class of program defined in Sect. 2.

We obtain a tool called PInterproc that infers at each program point invariants that are fully relational between aliasing/pointer properties and invariance/scalar properties. PInterproc can be tried on-line[1] on a number of examples, or on any program provided by the user.

Compared to the original version of Interproc, PInterproc has to preprocess expressions and assignments before feeding them to BddApron. Expressions like `*p+*q>=0` are typically expressed as conditional expressions

```
    (if p=&a then a else b) + (if q=&a then a else b) >= 0
```

which are supported by the library and normalized by pushing operations `+`,`>=` under the tests. Assignments like `*p:=e` are decomposed in

```
    if p=&a then a:=e else b:=e
```

The relation R^+ discussed in Sect. 5 is encoded exactly as a Boolean expression with equality constraints on numerical locations, whereas relation $R^-(\sigma, \sigma', \sigma'')$ defined in Tab. 6 and used in Eqn. (4) is actually encoded as a parallel assignment defining σ''.

Example of analysis. We consider the procedure of Fig. 9 that transfers the content of a pointed integer into another one, in a way similar to bank account transfer. The source is set to 0 and its value is credited to the destination. If the source or destination are *nil*, then no movement is performed. We show that the summaries generated by our analysis are alias-sensitive, and more generally context sensitive. The summary generated for procedure `transfer` (at `(4)`) is dependent from the possible aliasing contexts. We show (partial) invariant at point `(4)` and its call-context dependencies:

$$\texttt{(4)} \left| \begin{array}{ll} \left(\begin{array}{l} \texttt{src} = \&\text{``}{*}\texttt{src}\text{''} \wedge \texttt{dest} = \&\text{``}{*}\texttt{dest}\text{''} \\ \wedge 0 \leq \text{``}{*}\texttt{src}\text{''}_0 \leq 10 \wedge \text{``}{*}\texttt{dest}\text{''}_0 = 3 \\ \wedge \text{``}{*}\texttt{src}\text{''} = 0 \wedge \text{``}{*}\texttt{dest}\text{''} = \text{``}{*}\texttt{dest}\text{''}_0 + \text{``}{*}\texttt{src}\text{''}_0 \end{array}\right) & \text{from } \texttt{(1)} \\ \vee \left(\begin{array}{l} \texttt{src} = \&\text{``}{*}\texttt{src}\text{''} \wedge \texttt{dest} = \mathit{nil} \\ \wedge 11 \leq \text{``}{*}\texttt{src}\text{''}_0 \leq 13 \wedge \text{``}{*}\texttt{src}\text{''} = \text{``}{*}\texttt{src}\text{''}_0 \end{array}\right) & \text{from } \texttt{(2)} \\ \vee \left(\begin{array}{l} \texttt{src} = \&\text{``}{*}\texttt{src}\text{''} \wedge \texttt{dest} = \&\text{``}{*}\texttt{src}\text{''} \\ \wedge 11 \leq \text{``}{*}\texttt{src}\text{''}_0 \leq 13 \wedge \text{``}{*}\texttt{src}\text{''} = 0 \end{array}\right) & \text{from } \texttt{(3)} \end{array}\right.$$

The aliasing context at call point `(3)` exhibits what can be a bad behaviour of the procedure (money or debt disappeared), when both parameters points to the same integer. Other experiments can be found at the webpage of PInterproc.

[1] http://pop-art.inrialpes.fr/interproc/pinterprocweb.cgi

```
var a:int,b:int;
begin
  a = 0;
  while(a <= 10) do
    b = 3;
    transfer(&a,&b); // (1)
  done;
  transfer(&a,nil);  // (2)
  transfer(&a,&a);   // (3)
end
```

```
proc transfer (src:int*,dest:int*)
begin
  if (not (src == nil)) and
     (not (dest == nil)) then
    *dest = *dest + *src;
    *src = 0;
  endif;   // (4)
end
```

Fig. 9: Bank account transfer program.

Complexity. Due to space constraints, we produce only a rough complexity analysis. Assuming $l = l_p + l_b + l_n$ external locations, resp. decomposed resp. into locations of pointer, Boolean, and numerical type (see Sect. 5 for an evaluation of l) and $v = v_p + v_b + v_n$ local variables decomposed in the same way, an abstract value can be represented by a MTBDD

- with at most $2(l_p + v_p)\log_2(l + v) + 2(l_b + v_b)$ Boolean variables,
- and with numerical abstract values on at most $2(l_n + v_n)$ dimensions.

The factor 2 is due to the copies of formal parameters and external locations. The discussion in Sect. 5 about the number of possible locations and the effective size of enumerated types encoding pointer values is very important in practice for efficiently encoding activation records and for controlling the number of disjunctions to handle in expressions and assignments. In particular the term $\log_2(l + v)$ totally ignores the type information that restricts the possible values of pointers, whereas our implementation performs such optimizations.

Alternative abstractions. An important idea of this work is to derive an effective analysis by decomposing it in the well-identified steps depicted on Fig. 1, and to delay as much as possible approximations on pointers and variables. We show here that our methodology allows to express more classical, previously published analyses. We assume that an activation record is encoded with a function $Id \to \mathbb{B} \cup \mathbb{E} \cup \mathbb{Z}$ and we decompose $Id = Id_p \cup Id_s$ into identifiers of resp. pointer and scalar types.

- We can obtain a pure alias analysis if we forget the values of scalars, using the abstraction
  $$\mathcal{P}(Id \to \mathbb{B} \cup \mathbb{E} \cup \mathbb{Z}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A = \mathcal{P}(Id_p \to \mathbb{E}) \simeq \mathcal{P}(\mathbb{B}^n)$$
  This defines a flow-sensitive, context-sensitive, and fully relational interprocedural alias analysis, in which procedure summaries establishes a relation between aliasing properties at start and exit points.
- If we perform a further non-relational abstraction:
  $$\mathcal{P}(Id_p \to \mathbb{E}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (Id_p \to \mathcal{P}(\mathbb{E}))$$
  we obtain a flow-sensitive, context-insensitive interprocedural points-to analysis. Procedure summaries are of the form $P_{in} \wedge P_{out}$ and does not really relate input to output alias properties.

– We can also obtain the reduced product of an alias and a scalar analysis as follows:

$$\mathcal{P}(Id \to \mathbb{B} \cup \mathbb{E} \cup \mathbb{Z}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{P}(Id_p \to \mathbb{E}) \times \mathcal{P}(Id_s \to \mathbb{B} \cup \mathbb{Z})$$

The two analyses interacts during the fixpoint computation; typically the logico-numerical abstract domain will query the alias property when computing the effect of an assignment `*p=*p+1`, to know to which location `p` may point to.

7 Related Work

In 2001, Hind made a survey [13] of twenty years of pointer analysis. He counted not least than seventy-five papers and nine Ph.D. theses. It is legitimate to ask what is the contribution of our paper in such a well-studied field. First, we need to emphasize that most pointer analyses try to determine the possible aliasing in a program, regardless of the boolean and numerical values manipulated. One of our contribution is to allow the functional analysis of programs with pointers, in which the aliasing information both benefits from and benefits to the logico-numerical information.

Alias analysis for compilation. Pointer analyses that are directed to program optimisation are not suitable for a precise data-flow analysis (DFA). Andersen [1] and Steensgaard [28] founded two families of algorithms for *flow-insensitive* points-to analysis. Their methods lead to fast analysis (million lines of code), with a precision sufficient for optimisation but unfortunately insufficient for program functional analysis. In [13], Hind reported the point of view of Manuel Fähndrich, who states that "For error detection and program understanding, [...] there seems to be a lower bound on precision, below which, pointer information is pretty useless". However, we share with those analyses the concern of a precision adapted to the "client analysis" needs (eg. [5] and [12]). In our approach, the pointer analysis is merged with the logico-numerical analysis, which enhances the precision of both analyses.

The abstract domain library BddApron we use relies on BDDs to encode the value of pointers. BDDs have already been used in alias analysis, together with Datalog, for a context-sensitive, flow-insensitive points-to analysis for Java [29], but they are necessarily more efficient than analysis in pure Datalog [3]. We actually use BDDs in combination with numerical values, so the observation of [3] does not apply as is to our case.

Alias analysis for verification. Precise, but expensive, pointer analyses have also been proposed. For example, Landi and Ryder [21] tackles the may-alias problem and they merge into an algorithm the semantic work we did in Section 3 and an abstraction (less precise than the one we present in Section 4 and 6) This fusion of two distinct aspects of the analysis prevent the reuse of the algorithm with different abstraction schemes. Wilson and Lam [30] presents a lot of similarities with our work. They tackle more general C programs and they use a notion of external locations. However, they only address may-analysis of pointers (thus ignoring properties on scalars), and their analysis is defined by algorithms and not by semantic domains. Compared to our work, the lack of such a formalisation makes difficult its generalization to a different context (eg, combining it with scalar analysis using BDDs and convex polyhedra, or with shape analysis using a shape domain) and forbids soundness proofs. The work that actually inspired us for the local semantics is Bourdoncle's [2], in which the semantics adaptation (of reference parameters) is done independently of the abstraction step. As this work targets the Pascal language it does not handle pointers to the stack. Pointer parameters bring more difficulties for the analysis and for context determination than reference parameters.

Pointers on the execution stack are different from pointers within the heap. Interprocedural shape analyses like [25, 17] address the specific problem of verifying the recursive data-structures in the heap. They are both based on a relational interprocedural analysis, but [25] uses a tabulated representation for summary functions whereas [17] exploits a more symbolic representation of relations between input/output memory heaps, in a spirit similar to Sect 5. Control-flow analysis (CFA) aims at discovering the partly implicit control flow of higher-order or object-oriented programs such as Java (see [22] for a survey). CFA often includes rather precise alias analysis, but of pointers to the heap only, as such programs do not have pointers to the execution stack.

Inferring invariants on variables of programs. The original Interproc analyser deal with a simple language with procedures and recursive calls. It cannot handle most C-like programs since they often rely on pointers, dynamic allocation and arrays. The work we present here is a step toward the C language. As mentioned in the introduction, several well-established C analysers like Astrée [8] and Fluctuat [9] that infer sophisticated properties on numerical variables target specific kinds of programs for which they can inline procedures, so they only need to handle intraprocedural use of pointers (eg. see [23] for Astrée).

8 Conclusion

We addressed the problem of interprocedural analysis in the presence of pointers to the stacks. By doing so, we make a step toward the analysis of C codes which often rely on pointers as parameters and side-effects.

Our approach follows the abstract interpretation scheme depicted on Fig. 1 and carefully separate semantic and algorithmic issues. The first contribution of the paper is an alternative but *local* semantics for our language, in which the instructions act locally on the stack, even in the presence of pointers and side-effects. We prove it to be equivalent to the original semantics, which is not so straightforward. We are convinced that this approach can be easily generalized to more complex languages with dynamic allocation.

We then apply relational interprocedural analysis to this local semantics, which result in a forward semantics manipulating sets of activation records.

Our second contribution concerns the symbolic representation of such activation records, that relate alias properties on pointers and properties on scalar variables and locations pointed to by pointers. We abstract these activation records with the relational abstract domains provided by the library BddApron, and we implemented the PInterproc analyzer as an extension of the Interproc for programs with pointers on local variables.

We prove as a side-effect that for Boolean programs, our analysis is exact w.r.t. invariance properties at each control point. We also show in Sect. 6 that by further abstractions our approach leads to known alias analysis techniques, that are less precise but more efficient.

The main question in our view is to which extend our approach can be generalized to more expressive programs. Concerning global variables, one can push them on the stack when instrumenting them (see [15]) and the main change is that the domain of pointer values is larger, as we get more locations in stacks. Adding structured types (*eg.* records) raises two difficulties: aliasing properties are more complex, and it becomes possible to build an unbounded linked list on the call stack, which induces an unbounded number of external locations in our local semantics. This calls for a mechanism to merge external locations, as done in shape analysis [4, 25, 17]. At last, in the presence of dynamically allocated objects, we are convinced that one can still exploit our local semantics, and the question is how to combine an existing shape abstract domain with our abstraction for scalar and pointer variables.

References

1. Andersen, L.: Program Analysis and Specialization for the C Programming Language. Ph.D. thesis (1994), ftp.diku.dk/pub/diku/semantics/papers/D-203.dvi.Z
2. Bourdoncle, F.: Interprocedural Abstract Interpretation of Block Structured Languages with Nested Procedures, Aliasing and Recursivity. In: Deransart, P., Maluszynski, J. (eds.) PLILP. Lecture Notes in Computer Science, vol. 456, pp. 307–323. Springer (1990)
3. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: Arora, S., Leavens, G.T. (eds.) OOPSLA. pp. 243–262. ACM (2009)
4. Chase, D.R., Wegman, M., Zadeck, F.K.: Analysis of pointers and structures. In: Proceedings of the ACM SIGPLAN 1990 conference on Prog. language design and implementation (1990)
5. Chatterjee, R., Ryder, B.G., Landi, W.: Relevant context inference. In: POPL. pp. 133–146 (1999)
6. Cousot, P., Cousot, R.: 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)
7. Cousot, P., Cousot, R.: Static determination of dynamic properties of recursive procedures. In: IFIP Conf. on Formal Description of Programming Concepts (1977), http://www.di.ens.fr/ cousot/COUSOTpapers/IFIP77.shtml
8. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Why does astrée scale up? Formal Methods in System Design 35(3) (2009)

9. Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., Védrine, F.: Towards an industrial use of fluctuat on safety-critical avionics software. In: Formal Methods for Industrial Critical Systems, FMICS'09. LNCS, vol. 5825 (2009)
10. Filliâtre, J.C., Marché, C.: Multi-prover verification of c programs. In: ICFEM. pp. 15–29 (2004)
11. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: Prog. Lang. Design and Implementation, PLDI'08. ACM (2008)
12. Heintze, N., Tardieu, O.: Demand-driven pointer analysis. In: PLDI. pp. 24–34 (2001)
13. Hind, M.: Pointer analysis: haven't we solved this problem yet? In: PASTE. pp. 54–61. ACM (2001)
14. Jeannet, B.: The BDDAPRON logico-numerical abstract domains library, http://www.inrialpes.fr/pop-art/people/bjeannet/bjeannet-forge/bddapron/
15. Jeannet, B.: Relational interprocedural verification of concurrent programs. In: Software Engineering and Formal Methods, SEFM'09. IEEE (Nov 2009), http://pop-art.inrialpes.fr/people/bjeannet/publications/sefm09.pdf
16. Jeannet, B., Argoud, M., Lalire, G.: The Interproc interprocedural analyzer, http://pop-art.inrialpes.fr/interproc/interprocweb.cgi
17. Jeannet, B., Loginov, A., Reps, T., Sagiv, M.: A relational approach to interprocedural shape analysis. ACM Trans. On Programming Languages and Systems (TOPLAS) 32(2) (2010)
18. Jeannet, B., Miné, A.: The APRON abstract domains library, http://apron.cri.ensmp.fr/library/
19. Jeannet, B., Serwe, W.: Abstracting call-stacks for interprocedural verification of imperative programs. In: Int. Conf. on Algebraic Methodology and Software Technology, AMAST'04. LNCS, vol. 3116 (2004)
20. Knoop, J., Steffen, B.: The interprocedural coincidence theorem. In: Compiler Construction, CC'92. LNCS, vol. 641 (1992)
21. Landi, W., Ryder, B.G.: A safe approximate algorithm for interprocedural pointer aliasing. In: PLDI. pp. 235–248 (1992)
22. Midtgaard, J.: Control-flow analysis of functional programs. ACM Computing Surveys (2011), forthcoming, preliminary version available as BRICS technical report RS-07-18
23. Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: Languages, Compilers and Tools for Embedded Systems, LCTES'06 (2006)
24. Polyspace, http://www.mathworks.com/products/polyspace/
25. Rinetzky, N., Sagiv, M., Yahav, E.: Interprocedural shape analysis for cutpoint-free programs. In: Static Analysis Symposium, SAS'05. LNCS, vol. 3672 (2005)
26. Sharir, M., Pnueli, A.: Semantic foundations of program analysis. In: Muchnick, S., Jones, N. (eds.) Program Flow Analysis: Theory and Applications, chap. 7. Prentice-Hall (1981)
27. Somenzi, F.: Cudd: Colorado University Decision Diagram Package, ftp://vlsi.colorado.edu/pub
28. Steensgaard, B.: Points-to Analysis in Almost Linear Time. In: POPL. pp. 32–41 (1996)
29. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: PLDI. pp. 131–144 (2004)
30. Wilson, R.P., Lam, M.S.: Efficient context-sensitive pointer analysis for c programs. In: PLDI. pp. 1–12 (1995)

A Proof of Equivalence of Theorem 1

First of all, we define the interpretation of the original stacks. This interpretation simply states that values above the stack are undefined. We define the function $\mathit{eval}[\langle n, F\rangle](k,x)$ which computes the value at a given address in a standard stack:

$$\mathit{eval}[\langle n, F\rangle](k,x) = \begin{cases} F(k,x) & \text{when } k \leq n \\ \text{undefined} & \text{otherwise} \end{cases}$$

We also define the function $\mathit{eval}[\sigma_1 \dots \sigma_n](k,x)$ which computes the value at a given (standard) address in an instrumented stack:

$$\begin{aligned}
\mathit{eval}[\sigma_1 \dots \sigma_n](k,l) &= \begin{cases} \mathit{eval}(k+1, \mathit{bind}_{\sigma_k}(l)) & \text{when } l \in \mathit{dom}(\mathit{bind}_{\sigma_k}) \wedge k \neq n \\ \widetilde{\mathit{address}}(k, \sigma_k(l)) & \text{otherwise} \end{cases} \\
\widetilde{\mathit{address}}(k,v) &= \begin{cases} \mathit{address}(k,v) & \text{when } v \in \mathit{Loc} \\ v & \text{when } v \in \mathit{Scalar} \cup \{\mathit{nil}, \bot\} \end{cases} \\
\mathit{address}(k,l) &= \begin{cases} \mathit{address}(k-1, \mathit{bind}^{-1}_{\sigma_{k-1}}(l)) & \text{when } l \in \mathit{External} \\ \langle k, l\rangle & \text{when } l \in \mathit{Var} \end{cases}
\end{aligned}$$

We make two assumptions on bind_σ. First, it is a one-to-one function. Then, a location reachable through the parameters of a function belongs to the binding, which is formally:

Property 1. For all $k < n$,

$$z \in \mathbf{x}_k \cup \mathit{dom}(\mathit{bind}_{\sigma_k}) \Rightarrow \sigma_k(z) \in \mathit{dom}(\mathit{bind}_{\sigma_k}) \cup \mathit{Scalar} \cup \{\mathit{nil}, \bot\}$$

Both bindings presented in the paper satisfies this conditions.

Proof of Theorem 1 is given by showing that:

$$\begin{aligned}
&\mathit{eval}[\langle n, F\rangle] = \mathit{eval}[\Gamma] \Rightarrow \\
&\quad \begin{cases} \langle n, F\rangle \rightsquigarrow \langle n', F'\rangle \Rightarrow \exists \Gamma', \Gamma \rightarrow \Gamma' \wedge \mathit{eval}[\langle n', F'\rangle] = \mathit{eval}[\Gamma'] \\ \Gamma \rightarrow \Gamma' \Rightarrow \exists \langle n', F'\rangle, \langle n, F\rangle \rightsquigarrow \langle n', F'\rangle \wedge \mathit{eval}[\langle n', F'\rangle] = \mathit{eval}[\Gamma'] \end{cases}
\end{aligned}$$

The complex structure of the formula we want to prove is due to the non-determinism of a few operations of the language (initialization of scalars in activation records, maybe a `random` expression). These operations have little to do with the proof and it is not hard to convince oneself that both semantics are non-deterministic in the same way. We simplify the formula we want to prove by assuming deterministic transition systems (for example, we assume that local variables are initialized to 0 or *false*). We then need to prove:

$$\begin{aligned}
&\mathit{eval}[\langle n, F\rangle] = \mathit{eval}[\Gamma] \Rightarrow \\
&\quad (\langle n, F\rangle \rightsquigarrow \langle n', F'\rangle \iff \Gamma \rightarrow \Gamma') \wedge \mathit{eval}[\langle n', F'\rangle] = \mathit{eval}[\Gamma']
\end{aligned} \tag{7}$$

This formula states that stacks which agrees on their content evolve together and keep agreeing on their contents. We prove this formula by considering the cases of procedure call, procedure return and intraprocedural instruction.

These cases all rely on the following lemmas:

Lemma 1. *The address function (and by extension the $\widetilde{address}$ function) is independent from the top of the stack (including the current store).*

$$address[\sigma_1 \dots \sigma_{k-1} \dots \sigma_n](k, z) = address[\sigma_1 \dots \sigma_{k-1}](k, z)$$

$$\text{We define } B_{k;n} \text{ to be } \begin{cases} bind_{\sigma_n} \circ \dots \circ bind_{\sigma_k} & \text{when } k \leq n \\ id & \text{when } k = n + 1 \end{cases}$$

Lemma 2. *Consider two instrumented stacks which agree up to some height k:*

$$\begin{cases} \Gamma & = \sigma_1 \dots \sigma_k . \sigma_{k+1} \dots \sigma_n \\ \Gamma' & = \sigma_1 \dots \sigma_k . \sigma'_{k+1} \dots \sigma'_{n'} \end{cases}$$

We have:

$$\forall z \in Var, \forall i \leq k, z \notin B_{i;k} \Rightarrow eval[\Gamma](k, z) = eval[\Gamma'](k, z)$$

Every proof follow the same scheme. We consider a given (class of) address a. For each class, we either

- give a value v, then show that both $eval[\Gamma'](a)$ and $eval[\langle n', F'\rangle](a)$ are equal to v,
- give an address a_0, then show that $eval[\Gamma'](a) = eval[\Gamma](a_0)$ and $eval[\langle n', F'\rangle](a) = eval[\langle n, F\rangle](a_0)$. We conclude with the hypothesis that $eval[\Gamma](a_0) = eval[\langle n, F\rangle](a_0)$.

A.1 Procedure call

The calling rules make the transitions $\langle n, F\rangle \rightsquigarrow \langle n+1, F'\rangle$ and $\Gamma.\sigma \rightarrow \Gamma.\sigma.\sigma'$. Conditions are given respectively on pages 5 and 8.

k	z	source	
$= n + 1$	$= \mathbf{fp}^{(i)}$	$\langle n, \mathbf{x}^{(i)}\rangle$	(address)
$= n + 1$	$\in Var_{\tau *} \backslash \mathbf{fp}$	nil	(value)
$= n + 1$	$\in Var_{\tau_0} \backslash \mathbf{fp}$	$\in Scalar_{\tau_0}$	(value)
$\leq n$	$\in B_{k;n}$	itself	(address)
$\leq n$	$\notin B_{k;n}$	itself	(address)

Lemma 3.

$$\begin{aligned} &\forall v \in dom(bind_{\sigma_k}) \cup Scalar \cup \{nil, \perp\}, \forall k < |\Gamma|, \\ &\quad \widetilde{address}[\Gamma](k+1, \widetilde{bind}_{\sigma_k}(v)) = \widetilde{address}[\Gamma](k, v) \end{aligned}$$

Case of a formal parameter in the new activation record. We have $k = n + 1$ and $z = \mathbf{fp}^{(i)}$. The source of this value is the address $\langle n, \mathbf{x}^{(i)}\rangle$.

$$\begin{aligned} eval[\langle n+1, F'\rangle](n+1, \mathbf{fp}^{(i)}) &= F'(n+1, \mathbf{fp}^{(i)}) && \text{(definition)} \\ &= F(n, \mathbf{x}^{(i)}) && \text{(rule)} \\ &= eval[\langle n, F\rangle](n, \mathbf{x}^{(i)}) && \text{(definition)} \end{aligned}$$

$$\begin{aligned}
&\mathit{eval}[\mathit{\Gamma}.\sigma.\sigma'](n+1,\mathbf{fp}^{(i)})\\
&\quad = \widetilde{\mathit{address}}[\mathit{\Gamma}.\sigma.\sigma'](n+1,\sigma'(\mathbf{fp}^{(i)})) && \text{(definition)}\\
&\quad = \widetilde{\mathit{address}}[\mathit{\Gamma}.\sigma.\sigma'](n+1,\widetilde{\mathit{bind}}_\sigma(\sigma(\mathbf{x}^{(i)}))) && \text{(rule)}\\
&\quad = \widetilde{\mathit{address}}[\mathit{\Gamma}.\sigma.\sigma'](n,\sigma(\mathbf{x}^{(i)})) && \text{(Prop. 1 + Lemma 3)}\\
&\quad = \widetilde{\mathit{address}}[\mathit{\Gamma}.\sigma](n,\sigma(\mathbf{x}^{(i)})) && \text{(Lemma 2)}\\
&\quad = \mathit{eval}[\mathit{\Gamma}.\sigma](n,\mathbf{x}^{(i)}) && \text{(definition)}
\end{aligned}$$

Case of a local pointer in the new activation record. We have $k = n+1$ and $z \in \mathit{Var}_{\tau *}\backslash\mathbf{fp}$. The source of this value is the value *nil*.

$$\begin{aligned}
\mathit{eval}[\langle n+1,F'\rangle](n+1,z) &= \mathit{nil} && \text{(definition + rule)}\\
\mathit{eval}[\mathit{\Gamma}.\sigma.\sigma'](n+1,z) &= \mathit{nil} && \text{(definition + rule)}
\end{aligned}$$

Case of a location accessible by the callee. We have $k \leq n$ and $z \in \mathit{dom}(B_{k;n})$. The value at this address is unchanged.

$$\begin{aligned}
\mathit{eval}[\langle n+1,F'\rangle](k,z) &= F'(k,z) && \text{(definition)}\\
&= F(k,z) && \text{(rule)}\\
&= \mathit{eval}[\langle n,F\rangle](k,z) && \text{(definition)}
\end{aligned}$$

$$\begin{aligned}
&\mathit{eval}[\mathit{\Gamma}.\sigma.\sigma'](k,z)\\
&\quad = \mathit{eval}[\mathit{\Gamma}.\sigma.\sigma'](n+1,B_{k;n}(z)) && \text{(definition)}\\
&\quad = \widetilde{\mathit{address}}[\mathit{\Gamma}.\sigma.\sigma'](n+1,\sigma'\circ B_{k;n}(z)) && \text{(definition)}\\
&\quad = \widetilde{\mathit{address}}[\mathit{\Gamma}.\sigma.\sigma'](n+1,\widetilde{\mathit{bind}}_\sigma\circ\sigma\circ\mathit{bind}_\sigma^{-1}\circ B_{k;n}(z)) && \text{(rule)}\\
&\quad = \widetilde{\mathit{address}}[\mathit{\Gamma}.\sigma.\sigma'](n+1,\widetilde{\mathit{bind}}_\sigma\circ\sigma\circ B_{k;n-1}(z)) && \text{(bijection)}\\
&\quad = \widetilde{\mathit{address}}[\mathit{\Gamma}.\sigma.\sigma'](n,\sigma\circ B_{k;n-1}(z)) && \text{(Prop. 1 + Lemma 3)}\\
&\quad = \widetilde{\mathit{address}}[\mathit{\Gamma}.\sigma](n,\sigma\circ B_{k;n-1}(z)) && \text{(Lemma 2)}\\
&\quad = \mathit{eval}[\mathit{\Gamma}.\sigma](n,B_{k;n-1}(z)) && \text{(definition)}\\
&\quad = \mathit{eval}[\mathit{\Gamma}.\sigma](k,z) && \text{(definition)}
\end{aligned}$$

Case of a location inaccessible by the callee. We have $k < n$ and $z \notin \mathit{dom}(B_{k;n})$.

$$\begin{aligned}
\mathit{eval}[\mathit{\Gamma}.\sigma.\sigma'](k,z) &= \mathit{eval}[\mathit{\Gamma}.\sigma](k,z) && \text{(Lemma 2)}\\
\mathit{eval}[\langle n',F'\rangle](k,z) &= \mathit{eval}[\langle n,F\rangle](k,z) && \text{(definition)}
\end{aligned}$$

A.2 Procedure return

The return rules make the transitions $\langle n+1, F\rangle \rightsquigarrow \langle n, F''\rangle$ and $\Gamma.\sigma.\sigma' \to \Gamma.\sigma''$. Conditions are given respectively on pages 5 and 8. We write Addr_n for the set of addresses having index n.

k	z		source	
$= n$	$= \mathbf{y}^{(i)}$		$\langle n+1, \mathbf{fr}^{(i)}\rangle$	(address)
$\leq n$	$\in \mathit{dom}(B_{k;n})$	$\notin \mathbf{y}$	itself	(address)
$\leq n$	$\notin \mathit{dom}(B_{k;n})$	$\notin \mathbf{y}$	itself	(address)

For the two first cases, we also a the sub-case where the evaluation of the source is in Addr_{n+1}, to deal with the pending pointers.

Lemma 4.

$$\mathit{eval}[\sigma_1 \dots \sigma_n](n, z) \in \mathit{Addr}_n \Longleftrightarrow \sigma_n(z) \in \mathit{Var}$$

Lemma 5.

$$\begin{aligned}&\forall v \in \mathit{dom}(\mathit{bind}_{\sigma_k}^{-1}) \cup \mathit{Scalar} \cup \{\mathit{nil}, \bot\}, \forall k < |\Gamma|\,,\\ &\widetilde{\mathit{address}}[\Gamma](k+1, v) = \widetilde{\mathit{address}}[\Gamma](k, \widetilde{\mathit{bind}}_{\sigma_k}^{-1}(v))\end{aligned}$$

Case of a return value. We have $k = n$ and $z = \mathbf{y}^{(i)}$. In a first time, we assume $\mathit{eval}[\langle n+1, F\rangle](n+1, \mathbf{fr}^{(i)}) = \mathit{eval}[\Gamma.\sigma.\sigma'](n+1, \mathbf{fr}^{(i)}) \notin \mathit{Addr}_{n+1}$.

$$\begin{aligned}&\mathit{eval}[\langle n, F''\rangle](n, \mathbf{y}^{(i)})\\ &\quad = \begin{cases} F'(n, \mathbf{y}^{(i)}) \text{ when } F'(n, \mathbf{y}^{(i)}) \notin \mathit{Addr}_{n+1} \\ \bot \qquad\qquad \text{otherwise} \end{cases} && \text{(definition)}\\ &\quad = F(n+1, \mathbf{fr}^{(i)}) && \text{(rule + hyp.)}\\ &\quad = \mathit{eval}[\langle n+1, F\rangle](n+1, \mathbf{fr}^{(i)}) && \text{(definition)}\end{aligned}$$

$$\begin{aligned}&\mathit{eval}[\Gamma.\sigma''](n, \mathbf{y}^{(i)})\\ &\quad = \widetilde{\mathit{address}}[\Gamma.\sigma''](n, \sigma''(\mathbf{y}^{(i)})) && \text{(definition)}\\ &\quad = \widetilde{\mathit{address}}[\Gamma](n, \sigma''(\mathbf{y}^{(i)})) && \text{(Lemma 1)}\\ &\quad = \widetilde{\mathit{address}}[\Gamma.\sigma.\sigma'](n, \sigma''(\mathbf{y}^{(i)})) && \text{(Lemma 1)}\\ &\quad = \widetilde{\mathit{address}}[\Gamma.\sigma.\sigma'](n, \widetilde{\mathit{bind}}_{\sigma}^{-1}(\sigma'(\mathbf{fr}^{(i)}))) && \text{(rule)}\\ &\quad = \widetilde{\mathit{address}}[\Gamma.\sigma.\sigma'](n+1, \sigma'(\mathbf{fr}^{(i)})) && \text{(Lemmas 5, 4 + hyp.)}\\ &\quad = \mathit{eval}[\Gamma.\sigma.\sigma'](n+1, \mathbf{fr}^{(i)}) && \text{(definition)}\end{aligned}$$

We now assume that $\mathit{eval}(n+1, \mathbf{fr}^{(i)}) \in \mathit{Addr}_{n+1}$.

$$\begin{aligned}
&\mathit{eval}[\langle n, F''\rangle](n, \mathbf{y}^{(i)}) \\
&\quad= \begin{cases} F'(n, \mathbf{y}^{(i)}) \text{ when } F'(n, \mathbf{y}^{(i)}) \notin \mathit{Addr}_{n+1} \\ \bot \qquad\qquad\; \text{otherwise} \end{cases} && \text{(definition)} \\
&\quad= \bot && \text{(rule + hyp.)}
\end{aligned}$$

$$\begin{aligned}
&\mathit{eval}[\varGamma.\sigma''](n, \mathbf{y}^{(i)}) \\
&\quad= \widetilde{\mathit{address}}[\varGamma.\sigma.\sigma'](n, \widetilde{\mathit{bind}}_\sigma^{-1}(\sigma'(\mathbf{fr}^{(i)}))) && \text{(as before)} \\
&\quad= \bot && \text{(Lemma 4 + hyp.)}
\end{aligned}$$

Case of non-pending side effect. We have $k \leq n$, $z \in \mathit{dom}(B_{k;n})$ and $z \notin \mathbf{y}$. and we assume $\mathit{eval}(n+1, B_{k;n}(z)) \notin \mathit{Addr}_{n+1}$.

$$\mathit{eval}[\langle n, F''\rangle](k, z) = \mathit{eval}[\langle n, F\rangle](k, z) \qquad \text{(definition + rule)}$$

$$\begin{aligned}
&\mathit{eval}[\varGamma.\sigma''](k, z) \\
&\quad= \mathit{eval}[\varGamma.\sigma''](n, B_{k;n-1}(z)) && \text{(definition)} \\
&\quad= \widetilde{\mathit{address}}[\varGamma.\sigma''](n, \sigma'' \circ B_{k;n-1}(z)) && \text{(definition)} \\
&\quad= \widetilde{\mathit{address}}[\varGamma.\sigma.\sigma'](n, \sigma'' \circ B_{k;n-1}(z)) && \text{(Lemma 1)} \\
&\quad= \widetilde{\mathit{address}}[\varGamma.\sigma.\sigma'](n, \widetilde{\mathit{bind}}_\sigma^{-1} \circ \sigma' \circ \mathit{bind}_\sigma \circ B_{k;n-1}(z)) && \text{(rule)} \\
&\quad= \widetilde{\mathit{address}}[\varGamma.\sigma.\sigma'](n, \widetilde{\mathit{bind}}_\sigma^{-1} \circ \sigma' \circ B_{k;n}(z)) && \text{(definition)} \\
&\quad= \widetilde{\mathit{address}}[\varGamma.\sigma.\sigma'](n+1, \sigma' \circ B_{k;n}(z)) && \text{(Lemma 5 + hyp.)} \\
&\quad= \mathit{eval}[\varGamma.\sigma.\sigma'](n+1, B_{k;n}(z)) && \text{(definition)} \\
&\quad= \mathit{eval}[\varGamma.\sigma.\sigma'](k, z) && \text{(definition)}
\end{aligned}$$

Case of the tail of the stack. Using Lemma 2.

A.3 Assignments

The correctness of the interprocedural transitions (assignments and tests) rely on the fact that expressions evaluate to the same value in both kind of stacks. We show a lemma that gives us the correlation between standard and instrumented expression semantics ($[\![]\!]^{\mathcal{V}}$ and $[\![]\!]^{\mathcal{A}}$), when the two stack agrees on the evaluation function.

Lemma 6 (Address and value semantics.).

$$\mathit{eval}[\langle n, F\rangle] = \mathit{eval}[\varGamma.\sigma] \Rightarrow \begin{cases} [\![\mathit{left}]\!]^{\mathcal{A}}_{\langle n,F\rangle} = \mathit{address}[\varGamma.\sigma](n, [\![\mathit{left}]\!]^{\mathcal{A}}_\sigma) \\ [\![\mathit{expr}]\!]^{\mathcal{V}}_{\langle n,F\rangle} = \widetilde{\mathit{address}}[\varGamma.\sigma](n, [\![\mathit{expr}]\!]^{\mathcal{V}}_\sigma) \end{cases}$$

We first consider the following lemma:

Lemma 7 (*eval* and *address*). *$\Gamma.\sigma$ is a stack of size n.*

$$\mathit{eval}[\Gamma.\sigma](\mathit{address}[\Gamma.\sigma](n,l)) = \mathit{eval}[\Gamma.\sigma](n,l)$$

Lemma 7 holds because the structure of *address* allows to descend in the stack, possibly down to the initial activation record, until it reaches a non-external location. The *eval* function can then get back to the top of the stack, since *address* was able to descend.

The proof of Lemma 6 is done by induction on the size of the expressions. We show the property for atoms:

$$\mathit{address}[\Gamma.\sigma](n, \llbracket \mathrm{id} \rrbracket^{\mathcal{A}}_{\sigma}) = \mathit{address}[\Gamma.\sigma](n, \mathrm{id}) = \langle n, \mathrm{id}\rangle = \llbracket \mathrm{id} \rrbracket^{\mathcal{A}}_{\langle n,F\rangle} \qquad (\mathrm{id} \in \mathit{Var})$$

$$\widetilde{\mathit{address}}[\Gamma.\sigma](n, \llbracket \mathtt{nil} \rrbracket^{\mathcal{V}}_{\sigma}) = \widetilde{\mathit{address}}[\Gamma.\sigma](n, \mathtt{nil}) = \mathtt{nil} = \llbracket \mathtt{nil} \rrbracket^{\mathcal{V}}_{\langle n,F\rangle}$$

$$\begin{cases} \widetilde{\mathit{address}}[\Gamma.\sigma](n, \llbracket \& \mathrm{id} \rrbracket^{\mathcal{V}}_{\sigma}) = \widetilde{\mathit{address}}[\Gamma.\sigma](n, \llbracket \mathrm{id} \rrbracket^{\mathcal{A}}_{\sigma}) \\ \qquad = \widetilde{\mathit{address}}[\Gamma.\sigma](n, \mathrm{id}) = \mathit{address}[\Gamma.\sigma](n, \mathrm{id}) = \langle n, \mathrm{id}\rangle \qquad (\mathrm{id} \in \mathit{Var}) \\ \llbracket \& \mathrm{id} \rrbracket^{\mathcal{V}}_{\langle n,F\rangle} = \llbracket \mathrm{id} \rrbracket^{\mathcal{A}} = \langle n, \mathrm{id}\rangle \end{cases}$$

We now assume that Lemma 6 holds for *left* and we show that it also holds for $*\mathit{left}$.

$$\begin{aligned} \llbracket *\mathit{left} \rrbracket^{\mathcal{A}}_{\langle n,F\rangle} &= \llbracket \mathit{left} \rrbracket^{\mathcal{V}}_{\langle n,F\rangle} \qquad (\text{and } \llbracket \mathit{left} \rrbracket^{\mathcal{V}}_{\langle n,F\rangle} \in \mathit{Loc}) \\ &= \widetilde{\mathit{address}}[\Gamma.\sigma](n, \llbracket \mathit{left} \rrbracket^{\mathcal{A}}_{\sigma}) && \text{(induction)} \\ &= \mathit{address}[\Gamma.\sigma](n, \llbracket \mathit{left} \rrbracket^{\mathcal{A}}_{\sigma}) && (\llbracket \mathit{left} \rrbracket^{\mathcal{V}}_{\langle n,F\rangle} \in \mathit{Loc}) \\ \mathit{address}[\Gamma.\sigma](n, \llbracket *\mathit{left} \rrbracket^{\mathcal{A}}) &= \mathit{address}[\Gamma.\sigma](n, \llbracket \mathit{left} \rrbracket^{\mathcal{V}}) && \text{(induction)} \end{aligned}$$

This proves one partial step of induction. Before concluding, we must complete the step by showing that property also holds for $\llbracket *\mathit{left} \rrbracket^{\mathcal{V}}$. We show a more general case which covers also $\llbracket \mathrm{id} \rrbracket^{\mathcal{V}}$ which had been left aside.

$$\begin{aligned} \llbracket \mathit{left} \rrbracket^{\mathcal{V}}_{\langle n,F\rangle} &= F(\llbracket \mathit{left} \rrbracket^{\mathcal{A}}_{\langle n,F\rangle}) = \mathit{eval}[\langle n,F\rangle](\llbracket \mathit{left} \rrbracket^{\mathcal{A}}_{\langle n,F\rangle}) && \text{(definitions)} \\ &= \mathit{eval}[\Gamma.\sigma](\llbracket \mathit{left} \rrbracket^{\mathcal{A}}_{\langle n,F\rangle}) && \text{(lemma hyp.)} \\ &= \mathit{eval}[\Gamma.\sigma](\mathit{address}[\Gamma.\sigma](n, \llbracket \mathit{left} \rrbracket^{\mathcal{A}}_{\sigma})) && \text{(partial induc.)} \\ &= \mathit{eval}[\Gamma.\sigma](n, \llbracket \mathit{left} \rrbracket^{\mathcal{A}}_{\sigma}) && \text{(Lemma 7)} \\ \widetilde{\mathit{address}}[\Gamma.\sigma](n, \llbracket \mathit{left} \rrbracket^{\mathcal{V}}_{\sigma}) &= \widetilde{\mathit{address}}[\Gamma.\sigma](n, \sigma(\llbracket \mathit{left} \rrbracket^{\mathcal{A}}_{\sigma})) \\ &= \mathit{eval}[\Gamma.\sigma](n, \llbracket \mathit{left} \rrbracket^{\mathcal{A}}_{\sigma}) && \text{(definitions)} \end{aligned}$$

□

For an assigment *left*`:=`*expr*, we consider the following cases:

$\langle k,z\rangle$	source	
$= \llbracket \mathit{left} \rrbracket^{\mathcal{A}}_{\langle n,F\rangle}$	$\llbracket \mathit{expr} \rrbracket^{\mathcal{V}}_{\langle n,F\rangle}$	(value)
$\neq \llbracket \mathit{left} \rrbracket^{\mathcal{A}}_{\langle n,F\rangle}$	itself	(address)

Case of the modified variable.

$$\mathit{eval}[\langle n, F'\rangle](\llbracket \mathit{left}\rrbracket^{\mathcal{A}}_{\langle n,F\rangle}) = \llbracket \mathit{expr}\rrbracket^{\mathcal{V}}_{\langle n,F\rangle} \qquad \text{(definition)}$$

$$\begin{aligned}
&\mathit{eval}[\Gamma.\sigma'](\llbracket \mathit{left}\rrbracket^{\mathcal{A}}_{\langle n,F\rangle}) \\
&\quad = \mathit{eval}[\Gamma.\sigma'](\mathit{address}[\Gamma.\sigma](n, \llbracket \mathit{left}\rrbracket^{\mathcal{A}}_{\Gamma.\sigma})) && \text{(Lemma 6)}\\
&\quad = \mathit{eval}[\Gamma.\sigma'](\mathit{address}[\Gamma.\sigma'](n, \llbracket \mathit{left}\rrbracket^{\mathcal{A}}_{\Gamma.\sigma})) && \text{(Lemma 1)}\\
&\quad = \mathit{eval}[\Gamma.\sigma'](n, \llbracket \mathit{left}\rrbracket^{\mathcal{A}}_{\Gamma.\sigma}) && \text{(Lemma 7)}\\
&\quad = \widetilde{\mathit{address}}[\Gamma.\sigma'](n, \sigma'(\llbracket \mathit{left}\rrbracket^{\mathcal{A}}_{\Gamma.\sigma})) && \text{(definition)}\\
&\quad = \widetilde{\mathit{address}}[\Gamma.\sigma'](n, \llbracket \mathit{expr}\rrbracket^{\mathcal{V}}_{\Gamma.\sigma}) && \text{(rule)}\\
&\quad = \llbracket \mathit{expr}\rrbracket^{\mathcal{V}}_{\langle n,F\rangle} && \text{(Lemma 6)}
\end{aligned}$$

A.4 Proof of Lemmas

Proof of Lemmas 1 and 2. Proof of Lemma 1 is trivial. It is sufficient to study the two cases of the definition. The *Var* case is straightforward, the *External* case is inductive and the induction is stopped by the last record of the stack, which does not contains external locations.

Proof of Lemma 2 is similar. It uses the fact that the considered variable does not escape the tail of the stack. We unfold the evaluation function until it stops, and conclude with Lemma 1.

Proof of Lemma 3 and 5. Assume $\Gamma = \sigma_1 \ldots \sigma_n$, $k < n$, $v \in \mathit{dom}(\mathit{bind}_{\sigma_k}) \cup \mathit{Scalar} \cup \{\mathit{nil}, \bot\}$ then:

$$\begin{aligned}
&\widetilde{\mathit{address}}[\Gamma](k+1, \widetilde{\mathit{bind}_{\sigma_k}}(v)) = \\
&\quad = \begin{cases} \mathit{address}[\Gamma](k+1, \mathit{bind}_{\sigma_k}(v)) & \text{when } v \in \mathit{Loc} \\ v & \text{when } v \notin \mathit{Loc} \end{cases} && \text{(definition)}\\
&\quad = \begin{cases} \mathit{address}[\Gamma](k, v) & \text{when } v \in \mathit{Loc} \\ v & \text{when } v \notin \mathit{Loc} \end{cases} && \text{(definition + bijection)}\\
&\quad = \widetilde{\mathit{address}}[\Gamma](k, v) && \text{(definition)}
\end{aligned}$$

Proof of Lemma 5 is similar.

Proof of Lemma 4 Trivial. Done by unfolding of the evaluation function.