Abstracting Call-Stacks for Interprocedural Verification of Imperative Programs

Bertrand Jeannet[1] and Wendelin Serwe[2]

[1] IRISA, 35042 Rennes, France, Bertrand.Jeannet@inria.fr
[2] INRIA Rhône-Alpes, 38334 St Ismier, France, Wendelin.Serwe@inria.fr

Abstract. We propose a new approach to interprocedural analysis and verification, consisting of deriving an interprocedural analysis method by abstract interpretation of the standard operational semantics of programs. The advantages of this approach are twofold. From a methodological point of view, it provides a direct connection between the concrete semantics of the program and the effective analysis, which facilitates implementation and correctness proofs. This method also integrates two main, distinct methods for interprocedural analysis, namely the call-string and the functional approaches introduced by Sharir and Pnueli. This enables strictly more precise analyses and additional flexibility in the tradeoff between efficiency and precision of the analysis.

1 Introduction

We consider the interprocedural verification of invariance properties of imperative programs. The applications we have in mind are automated debugging [4, 14] or automatic test selection [25], which may require precise and complex analyses. These are *flow-sensitive* (the analysis needs to take conditionals into account accurately), *attribute-dependent* (attributes (or properties) of variables are inter-related), and may require the use of *infinite* or even *infinite-height* lattices, in particular for the analysis of the properties numerical variables.

Our ambition is to design a method which is able to infer precise facts on the possible *call-stacks* of programs (and not only on the possible environments lying on top of the call-stack), and which can reuse existing abstract interpretation techniques and implementations available for intraprocedural analysis. For debugging applications, we would like to be able to answer questions like:

> Being in procedure P with environment ϵ_P, and assuming being called successively by R and Q with $x=y$, is it possible to enter procedure S?

Formally, this amounts to ask whether an execution path of the form

$$\langle c_R, ?\rangle \cdot \langle c_Q, x=y\rangle \cdot \langle c_P, \epsilon_P\rangle \rightarrow^* \Gamma \cdot \langle c_S, ?\rangle$$

exists. Such an *automatic state reaching* feature has already been implemented in a debugger for reactive programs [14].

We first present our method, before discussing existing approaches to interprocedural analysis and some of their limitations for the applications we have in mind.

Our method. We start from the standard (small-step) operational semantics of imperative programs and derive in two distinct abstraction steps an implementable analysis. Since we aim the verification of invariance properties, the property lattice L is the powerset of states $\wp(S)$. The difficulty in interprocedural analysis is that a state is an unbounded stack of *activation records*[3], i.e. pairs of control points and local environments, so that the state-space has the following form: $S = Act^+$ with $E^+ = \cup_{i \geq 1} E^i$ denoting the Kleene operator on a set E. This means that there are two sources of infinity in the lattice L: the unbounded length of stacks, and the cardinality of the domain Act, considered as infinite as soon as there are numerical variables or recursive data structures.

The first abstraction abstracts the collecting semantics — which manipulates (co-)reachable sets of stacks of activation records — to a semantics manipulating sets of extended activation records. This *stack abstraction*, which deals with the first source of infinity, is independent from a second, more classical *data abstraction*, which abstracts sets of extended activation records in one of the many computable abstract lattices that are available for intraprocedural analysis, in order to deal with the second source of infinity, *cf.* Fig. 1.

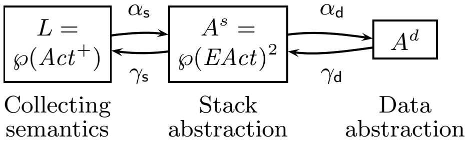


Fig. 1. Analysis scheme

The stack abstraction defines the interprocedural analysis method itself but does not lead directly to an effective analysis. Rather, it just simplifies the concrete lattice of sets of stacks into sets of simpler objects, for which computable abstract lattices already exists. Those need just to be equipped with a correct abstraction of procedure calls and returns operations in the stack abstraction lattice A^s, in order to obtain an implementable interprocedural analysis.

Existing approaches and some of their limitations for our applications. Interprocedural analysis methods have been widely studied and many of them can be classified as one of two classical approaches. We review in section 6 methods that do not fit in this classification. The *functional approach* [6, 28, 19] uses a denotational semantics of the analysed program and proceeds in two steps. The first step computes the predicate transformers associated with the procedures of the program, and the second step uses them to propagate an input predicate along the execution paths, to obtain the predicates holding at each control point. Some drawbacks of this approach are the following:

1. Predicates holding at control points are replaced by predicate transformers, which are more complex objects. This new viewpoint forbids the direct reuse of intraprocedural analyses (which use predicates).
2. The call-stack of the original program disappears, so no property can be specified on it.

The *operational approach*, generalising the "call-string approach" of [28], consists of adopting an operational semantics for programs. Fixpoint computation

[3] We assume in this paper that there are no global variables.

proceeds as in intraprocedural analysis, i.e. by propagating a predicate along execution paths. However, (an abstraction of) the call-stack has to be taken into account. The general abstraction scheme for call-stacks consists of using "tokens" to abstract stack-tails, and to label current activation records by them. These tokens represent the calling context of the current activation record [28, 18]. We bring a solution to the main limitations of this approach, namely:

- Tokens are used as labels. They are essentially enumerated and are not given any structure (as for instance a lattice structure).
- A notable consequence is that the set of tokens should be *finite* in practice, which means that the abstraction of stack-tails is very rough.

From a more synthetic point of view, both approaches lead to *context-sensitive* analyses. The functional approach takes into account the data part of the calling context of a procedure, whereas the operational approach focuses mainly on the control part. From this point of view, our stack abstraction enables an effective analysis which is both *data* and *control* context-sensitive.

Contributions. Using the principles of abstract interpretation, we unify two main interprocedural analysis methods and give correctness and optimality proofs with a minimal number of concepts. We provide a method which is both data and control context-sensitive, thus strictly more precise. Our approach enables backward analysis and its intersection with forward analysis. Finally, it facilitates implementation issues by clearly separating stack and data abstractions.

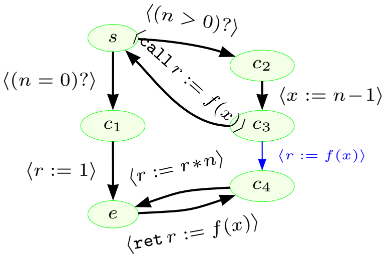
Outline. Section 2 presents the considered program model and its semantics. In Section 3 we revisit the functional approach with a first stack abstraction. We show the similarity of the two methods and discuss some advantages of our approach. In Section 4 we refine the previous abstraction by using call-strings, in order to subsume the call-string and the functional approaches. This allows us to accurately specify the above-mentioned debugging problem. Section 5 discusses the data abstraction step, in order to derive an effective analysis from a stack abstraction. Related work is described in Section 6 and Section 7 concludes. Due to the lack of space, we have omitted the proofs. They can be found in [16].

2 Program Model and Standard Semantics

We consider a simple imperative programming language with non-nested procedures and value parameter passing (as in Java or Ml). We suppose that each procedure has its own fixed set of variables, and do not consider global variables, which can be passed as additional procedure parameters. Similarly, programs are not restricted to programs manipulating scalar values: pointers and a memory heap can be modelled by adding to all procedures a special parameter modelling the memory heap. We require that formal input parameters *are not modified.* This is crucial to compute a relation between environments at the entry and any other point of a procedure (as in [28, 19]), and this is not restrictive either, as they can always be copied into additional local variables. The main restrictions

Var :	Variables: Var $= \bigcup_i LVar_i$
$LVar_i$, $\mathbf{loc}_i$:	Local variables of procedure P_i
In_i, $\mathbf{fpi}_i$:	Formal input parameters of P_i
Out_i, $\mathbf{fpo}_i$:	Formal output parameters of P_i
$G_i = \langle Ctrl_i, I_i \rangle$:	Flow graph of P_i
$\mathsf{s}_i, \mathsf{e}_i \in Ctrl_i$:	Entry and exit points of P_i
$G = \langle Ctrl, I \rangle$:	Flow graph of the program

Table 1. Syntactic domains.


s
c_1
c_2
c_3
c_4
e
$\langle (n > 0)? \rangle$
$\langle (n = 0)? \rangle$
$\langle \mathtt{call}\, r := f(x) \rangle$
$\langle x := n-1 \rangle$
$\langle r := f(x) \rangle$
$\langle r := 1 \rangle$
$\langle r := r * n \rangle$
$\langle \mathtt{ret}\, r := f(x) \rangle$

CFG for the Factorial Program

v	$\in Value$	: values
ϵ_i	$\in LEnv_i = LVar_i \to Value$	: local environments for P_i
ϵ	$\in LEnv = \bigcup_i LEnv_i$	: local environments
$\langle c, \epsilon \rangle$	$\in Act = Ctrl \times LEnv$	: activation record
Γ	$\in State = Act^+$	: stacks/program states

Table 2. Semantic domains.

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.

Program Syntax. The syntactic domains are summarised in Table 1.

A *program* is defined by a set $(P_i)_{0 \le i \le p}$ of procedures. Since we specify the initial states of an analysis separately, there is no particular "main" procedure.

Each *procedure* $P_i = \langle LVar_i, In_i, Out_i, G_i \rangle$ is defined by its intraprocedural control-flow graph G_i and its sets of local variables $LVar_i$, formal input parameters $In_i \subseteq LVar_i$ and formal output parameters $Out_i \subseteq LVar_i$. We note $\mathbf{x} = \langle \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)} \rangle$ vectors of variables. As vectors, the above-mentioned sets are written $\mathbf{loc}_i$, $\mathbf{fpi}_i$ and $\mathbf{fpo}_i$.

An *intraprocedural control-flow graph* is a graph $G_i = \langle Ctrl_i, I_i \rangle$ where $Ctrl_i$ is the set of *control points* of P_i, containing unique entry and exit control points s_i and e_i. $I_i : Ctrl_i \times Ctrl_i \to \mathsf{Inst}$ labels edges between control points 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 $R \subseteq LEnv^2$ describing the transformation of the top environment. We require the G_i to be deterministic for procedure calls, i.e. if $I_i(c, c')$ is a call then there exists no c'' such that $I_i(c, c'')$ or $I_i(c'', c')$ is a call.

The *interprocedural control-flow graph* (CFG) $G = \langle Ctrl, I \rangle$ is constructed from the set of intraprocedural ones. $Ctrl$ is defined as $Ctrl = \bigcup_{0 \le i \le p} Ctrl_i$ and I is defined as the "union" $\bigcup_i I_i$, where all procedure-call-edges are removed and replaced by two edges usually called *call-to-start* and *exit-to-return* edges:

$$\frac{I_i(c, c') \neq \langle \mathbf{y} := P_j(\mathbf{x}) \rangle}{I(c, c') = I_i(c, c')} \qquad \frac{I_i(c, c') = \langle \mathbf{y} := P_j(\mathbf{x}) \rangle}{\begin{array}{c} I(c, \mathsf{s}_j) = \langle \mathtt{call}\, \mathbf{y} := P_j(\mathbf{x}) \rangle \\ I(\mathsf{e}_j, c') = \langle \mathtt{ret}\, \mathbf{y} := P_j(\mathbf{x}) \rangle \end{array}}$$

$$\frac{I(c,c') = \langle R\rangle \quad R(\epsilon,\epsilon')}{\Gamma\cdot\langle c,\epsilon\rangle \to \Gamma\cdot\langle c',\epsilon'\rangle}\ \text{(Intra)} \qquad \frac{I(c,\mathsf{s}_j) = \langle \texttt{call}\,\mathbf{y} := P_j(\mathbf{x})\rangle \quad \forall k: \epsilon_j(\mathbf{fpi}_j^{(k)}) = \epsilon(\mathbf{x}^{(k)})}{\Gamma\cdot\langle c,\epsilon\rangle \to \Gamma\cdot\langle c,\epsilon\rangle\cdot\langle \mathsf{s}_j,\epsilon_j\rangle}\ \text{(Call)}$$

$$\frac{I(\mathsf{e}_j,c) = \langle \texttt{ret}\,\mathbf{y} := P_j(\mathbf{x})\rangle \quad \epsilon' = \epsilon[\mathbf{y}^{(k)} \mapsto \epsilon_j(\mathbf{fpo}_j^{(k)})]}{\Gamma\cdot\langle \mathsf{call}(c),\epsilon\rangle\cdot\langle \mathsf{e}_j,\epsilon_j\rangle \to \Gamma\cdot\langle c,\epsilon'\rangle}\ \text{(Return)}$$

Fig. 2. SOS rules defining →.

$$\begin{aligned}
post(c \xrightarrow{\langle R\rangle} c')(X) &= \left\{\Gamma\cdot\langle c',\epsilon'\rangle \;\middle|\; \Gamma\cdot\langle c,\epsilon\rangle \in X \wedge R(\epsilon,\epsilon')\right\}\\
post(c \xrightarrow{\langle \texttt{call}\,\mathbf{y}:=P_j(\mathbf{x})\rangle} \mathsf{s}_j)(X) &= \left\{\Gamma\cdot\langle c,\epsilon\rangle\cdot\langle \mathsf{s}_j,\epsilon_j\rangle \middle| \Gamma\cdot\langle c,\epsilon\rangle \in X \wedge \epsilon_j(\mathbf{fpi}_j^{(k)}) = \epsilon(\mathbf{x}^{(k)})\right\}\\
post(\mathsf{e}_j \xrightarrow{\langle \texttt{ret}\,\mathbf{y}:=P_j(\mathbf{x})\rangle} c)(X) &= \left\{\Gamma\cdot\langle c,\epsilon'\rangle \;\middle|\; \begin{array}{l}\Gamma\cdot\langle \mathsf{call}(c),\epsilon\rangle\cdot\langle \mathsf{e}_j,\epsilon_j\rangle \in X\\ \epsilon' = \epsilon[\mathbf{y}^{(k)} \mapsto \epsilon_j(\mathbf{fpo}_j^{(k)})]\end{array}\right\}
\end{aligned}$$

Table 3. Forward transfer function *post*.

Thus there are three kinds of instructions labelling edges of interprocedural CFGs: intraprocedural instructions $\langle R\rangle$, procedure calls $\langle \texttt{call}\,\mathbf{y} := P_j(\mathbf{x})\rangle$ and procedure returns $\langle \texttt{ret}\,\mathbf{y} := P_j(\mathbf{x})\rangle$, see the factorial program beside Table 1.

In the sequel, we use the following notations. For $c \in \mathit{Ctrl}$, c is a *call-site* to P_j if $\exists c' \ :\ I(c,c') = \langle \texttt{call y} := P_j(\mathbf{x})\rangle$. $\mathsf{proc}(c)$ denotes j such that $c \in \mathit{Ctrl}_j$. For any edge $c \xrightarrow{\langle \mathbf{y}:=P_j(\mathbf{x})\rangle} c'$, we define $\mathsf{ret}(c) = c'$ and $\mathsf{call}(c') = c$.

Operational Semantics. The semantic domains are summarised in Table 2.

The operational semantics is given by a *transition system* (*State*,→). *States* are *stacks* $\Gamma = \langle c_0,\epsilon_0\rangle\cdot\ldots\cdot\langle c_n,\epsilon_n\rangle$ of *activation records* (i.e. pairs of a control point c_i and an environment ϵ_i). $\langle c_n,\epsilon_n\rangle$ is the *current activation record* or *top* of Γ; the *tail* of Γ is Γ without its top, i.e. $\langle c_0,\epsilon_0\rangle\cdot\ldots\cdot\langle c_{n-1},\epsilon_{n-1}\rangle$. Environments map variables to values; their update is written $\epsilon[x \mapsto v]$. The *transition relation* $\to\ \subseteq \mathit{State} \times \mathit{State}$ is defined (in SOS-style) by the rules in Fig. 2. As long as a variable is not initialised, it holds nondeterministically any value in its domain, *cf.* rule (Call). As usual, $\to^*$ denotes the reflexive-transitive closure of →.

Standard Collecting Semantics. The forward collecting semantics describes the set of reachable states of a program. It is the natural choice for expressing and verifying invariance properties and is derived from the operational semantics by collecting the states belonging to executions of the program. We define the function $\mathit{reach} : \wp(\mathit{State}) \to \wp(\mathit{State})$ computing the states reachable from a set of *initial states* X_0 as:

$$\mathit{reach}(X_0) \stackrel{\text{def}}{=} \{q \mid \exists q_0 \in X_0\ ,\ q_0 \to^* q\}$$

It is also the least fix-point solution of $X = X_0 \cup \mathit{post}(X)$ where the *forward transfer function post* is defined by $\mathit{post}(X) = \{q' \mid \exists q \in X : q \to q'\}$. Table 3 gives the decomposition of *post* according to the transitions of the CFG, i.e. $\mathit{post}(X) = \bigcup_{(c,c')\in \mathit{Ctrl}\times\mathit{Ctrl}} \mathit{post}(c \xrightarrow{I(c,c')} c')(X)$. For $X_0, X \subseteq S$, we define $F[X_0](X) \stackrel{\text{def}}{=} X_0 \cup \mathit{post}(X)$. Since $F[X_0]$ is monotone and continuous, Kleene's fix-point theorem allows us to compute the forward collecting semantics by iterated application of $F[X_0]$ starting from $\emptyset$:

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

The collecting semantics can also be considered backward, yielding the set of states X from which a given set of *final states* X_0 is reachable. In this case we call X the set of *coreachable states* of X_0. We get the following definitions:

$$\begin{aligned} \mathit{pre}(X) &= \{q \mid \exists q' \in X : q \to q'\} \\ \mathit{coreach}(X_0) &= \mathrm{lfp}(G[X_0]) \;\text{ with }\; G[X_0](X) = X_0 \cup \mathit{pre}(X) \end{aligned}$$

Properties of the Stacks in the Standard Semantics. The assumption that formal input parameters are read-only variables induces strong properties on stacks which are the basis of our stack abstractions. A necessary condition for $q = \Gamma \cdot \langle c, \epsilon\rangle$ to lead to $q' = \Gamma \cdot \langle c, \epsilon\rangle \cdot \langle c', \epsilon'\rangle$ (where c is a call site to $P_{\mathsf{proc}(c')}$) is that the values of actual input parameters in ϵ have to match those of the formal input parameters in ϵ'. This is formalised by the following definition.

Definition 1. *$\langle c, \epsilon\rangle$ is a* valid calling activation record *(or* valid*) for $\langle c', \epsilon'\rangle$ if (i) c is a call site for procedure P_j: $\exists j : c' = \mathsf{s}_j \;\wedge\; I(c, \mathsf{s}_j) = \langle \mathtt{call}\,\mathbf{y} := P_j(\mathbf{x})\rangle$; (ii) actual and formal parameters are equal: $\forall k : \epsilon(\mathbf{x}^{(k)}) = \epsilon'(\mathbf{fpi}_j^{(k)})$.*

Extending Definition 1, we call a stack $\langle c_0, \epsilon_0\rangle \ldots \langle c_n, \epsilon_n\rangle$ *consistent* if $\langle c_i, \epsilon_i\rangle$ is valid for $\langle c_{i+1}, \epsilon_{i+1}\rangle$ ($\forall 0 \leq i < n$). From now on, we focus on consistent states and restrict *State* to its consistent subset.

3 Revisiting the Functional Approach

We present a first stack abstraction, which allows analysis results at least as accurate as the classical functional approach for forward analysis, but in addition:

- It enables a more direct reuse of standard data abstractions (we manipulate environments and not functions from environments to environments).
- Stacks can be rebuild by concretisation of abstract values.
- Defining backward analysis is straightforward.

3.1 Stack Abstraction

The main idea is to forget all about sequences of activation records in the stack, keeping information only on the activation records themselves. As already observed in a different context [19], we can nevertheless get accurate results.

An abstract state $Y = \langle Y_{hd}, Y_{tl}\rangle$ is composed of two sets of activation records. Y_{hd} represents top activation records, whereas Y_{tl} is the collapse of all activation records in stack-tails. This leads to the following abstract domain:

$$A^f \stackrel{\text{def}}{=} \wp(\mathit{Act}) \times \wp(\mathit{Act}) \tag{1}$$

which comes equipped with the standard lattice structure $A^f(\sqsubseteq, \sqcup, \sqcap, \top, \bot)$ of a (smashed) Cartesian product of lattices.

$$post^f_{hd}(c \xrightarrow{\langle R\rangle} c')(Y) = \big\{\langle c',\epsilon'\rangle \big| \langle c,\epsilon\rangle \in Y_{hd} \ \wedge \ R(\epsilon,\epsilon')\big\} \tag{3a}$$

$$post^f_{hd}(c \xrightarrow{\langle \mathtt{call}\,\mathbf{y}{:=}P_j(\mathbf{x})\rangle} \mathsf{s}_j)(Y) = \big\{\langle \mathsf{s}_j,\epsilon_j\rangle \big| \langle c,\epsilon\rangle \in Y_{hd} \ \wedge \ \epsilon_j(\mathbf{fpi}_j^{(k)}) = \epsilon(\mathbf{x}^{(k)})\big\} \tag{3b}$$

$$post^f_{hd}(\mathsf{e}_j \xrightarrow{\langle \mathtt{ret}\,\mathbf{y}{:=}P_j(\mathbf{x})\rangle} c)(Y) = \left\{\langle c,\epsilon'\rangle \;\middle|\; \begin{array}{l} \langle \mathsf{call}(c),\epsilon\rangle \in Y_{tl} \ \wedge \ \epsilon(\mathbf{x}^{(k)}) = \epsilon_j(\mathbf{fpi}_j^{(k)}) \\ \langle \mathsf{e}_j,\epsilon_j\rangle \in Y_{hd} \ \wedge \ \epsilon' = \epsilon\big[\mathbf{y}^{(k)} \mapsto \epsilon_j(\mathbf{fpo}_j^{(k)})\big] \end{array}\right\} \tag{3c}$$

$$post^f_{tl}(c \xrightarrow{\langle \mathtt{call}\,\mathbf{y}{:=}P_j(\mathbf{x})\rangle} \mathsf{s}_j)(Y) = Y_{tl} \ \cup \ \big\{\langle c,\epsilon\rangle \in Y_{hd}\big\} \tag{3d}$$

$$post^f_{tl}(c \xrightarrow{i} c')(Y) = Y_{tl} \quad \text{otherwise } (i \text{ is an instruction}) \tag{3e}$$

Table 4. Equations defining $post^f$.

We define the Galois connection $\wp(State) \underset{\alpha^f}{\overset{\gamma^f}{\leftrightarrows}} A^f$, with the *abstraction function* $\alpha^f = \langle \alpha^f_{hd}, \alpha^f_{tl}\rangle$ and *concretisation function* γ^f defined by:

$$\begin{array}{lccl}
\alpha^f : & X & \longmapsto & \bigsqcup_{q\in X} \ \alpha^f\big(\{q\}\big) \\
& \{\langle c_0,\epsilon_0\rangle\ldots\langle c_n,\epsilon_n\rangle\} & \longmapsto & \big\langle \ \big\{\langle c_n,\epsilon_n\rangle\big\} \ , \ \big\{\langle c_i,\epsilon_i\rangle \mid 0{\le}i{<}n\big\} \ \big\rangle \\
\gamma^f : & Y = \langle Y_{hd},Y_{tl}\rangle & \longmapsto & \left\{ q = \langle c_0,\epsilon_0\rangle\ldots\langle c_n,\epsilon_n\rangle \;\middle|\; \begin{array}{l} \langle c_n,\epsilon_n\rangle \in Y_{hd} \\ \forall 0{\le}i{<}n : \ \langle c_i,\epsilon_i\rangle \in Y_{tl} \\ q \text{ is a consistent stack} \end{array} \right\}
\end{array}$$

α^f_{hd} gathers the top activation records, whereas α^f_{tl} collects all the other activation records. To rebuild a stack from an abstract state, γ^f uses the notion of consistent stacks. Notice that $\langle c,\epsilon\rangle \in Y_{tl}$ implies that c is a call site.

The extensive function $\gamma^f \circ \alpha^f$ describes the information lost by A^f. If $\langle c_0,\epsilon_0\rangle$ is valid for $\langle c_1,\epsilon'_1\rangle$ and $\epsilon_1 \neq \epsilon'_1$, we might have:

$$\begin{array}{rl}
\langle c_0,\epsilon_0\rangle\cdot\langle c_1,\epsilon'_1\rangle \in (\gamma^f \circ \alpha^f) & \Big(\big\{\langle c_0,\epsilon_0\rangle\cdot\langle c_1,\epsilon_1\rangle, \ \langle c_1,\epsilon'_1\rangle\big\}\Big) \\
\notin & \ \ \big\{\langle c_0,\epsilon_0\rangle\cdot\langle c_1,\epsilon_1\rangle, \ \langle c_1,\epsilon'_1\rangle\big\}
\end{array} \tag{2}$$

However, $\alpha^f_{hd}(X)$ keeps *exact information* on *top* activation records, which is the only information computed by functional approaches.

The main reason why we separate top activation records from those below in the stack is for defining properly an abstract backward analysis, and also for being able to perform a forward analysis from an non empty stack.

3.2 Forward Analysis

Abstract Postcondition $\boldsymbol{post^f}$. To compute reachable states in the abstract domain, we need an abstract postcondition operator $post^f$. Table 4 specifies $post^f$, using the decomposition already used for $post$. We prove in [16] that $post^f$ is a correct approximation of $post$, i.e. $post^f \sqsupseteq \alpha^f \circ post \circ \gamma^f$.

Only procedure returns need comment, since the stack contents before the call has to be recovered. [28, 18, 19] use for this purpose a function *combine* to combine the environment of the caller at the call site with the environment of the callee at its exit point. In Section 2, we noticed that $\Gamma\cdot\langle \mathsf{call}(c),\epsilon\rangle\cdot\langle \mathsf{e}_j,\epsilon_j\rangle$ can be a successor of $\Gamma\cdot\langle \mathsf{call}(c),\epsilon\rangle$ only if $\langle \mathtt{call}(c),\epsilon\rangle$ is valid for $\langle \mathsf{e}_j,\epsilon_j\rangle$. So, for $\langle \mathsf{e}_j, \epsilon_j \rangle \in Y_{hd}$, (3c) selects the activation records $\langle \mathsf{call}(c), \epsilon \rangle \in Y_{tl}$ valid for it. Then, the reachable activation record(s) at c are obtained by assigning return parameters to $\mathbf{y}$. This is our combine operation, defined by abstraction of the concrete procedure return operation.

Because the input parameters are frozen, at the exit point e_j of a procedure P_j, the top of the stack (in Y_{hd}) contains a relation $\phi_j(\mathbf{fpi}_j, \mathbf{fpo}_j)$ between *reachable* call and return parameters, defined by $\phi_j = \big\{(\mathbf{x}, \mathbf{y}) \;\big|\; \exists \langle \mathsf{e}_j, \epsilon \rangle \in Y_{hd} : \mathbf{x} = \epsilon(\mathbf{fpi}_j) \wedge \mathbf{y} = \epsilon(\mathbf{fpo}_j)\big\}$. Since ϕ_j is the predicate transformer of P_j specialised on the reachable inputs, our handling of procedure returns can be seen as applying the predicate transformer of the callee to the valid calling activation records that are reachable in Y_{tl} at the call site.

Correctness and Optimality. Transposing the notion of reachable states into the abstract lattice A^f, we define $F^f[Y_0](Y) \overset{\text{def}}{=} Y_0 \sqcup \mathit{post}^f(Y)$ and $\mathit{reach}^f(Y_0) \overset{\text{def}}{=} \mathrm{lfp}(F^f[Y_0])$ $(\forall Y_0 \in A^f)$. Since post^f correctly approximates post, we deduce from abstract interpretation theory that we correctly approximate reachable states:

Theorem 1 (Correctness). *For any* $Y_0 \in A^f$*,* $\mathit{reach}^f(Y_0) \sqsupseteq \alpha^f \circ \mathit{reach} \circ \gamma^f(Y_0)$

The fact that we incrementally build the predicate transformer of a procedure at its exit point together with [28] suggests that we could improve and get the best we can hope for with a Galois connection, that is

$$\mathit{reach}^f \circ \alpha^f = \alpha^f \circ \mathit{reach}$$

This means it doesn't matter if we compute the fixpoint in the concrete lattice and then abstract the result, or directly compute in the abstract lattice.

Since Theorem 1 implies $\mathit{reach}^f \circ \alpha^f \sqsupseteq \alpha^f \circ \mathit{reach}$, we just have to prove the inverse inclusion. We show that $Y = \alpha^f \circ \mathit{reach}(X_0)$ is a post-fixpoint of post^f, i.e. that $\mathit{post}^f(Y) \sqsubseteq Y$, under additional assumptions on X_0. First, we require initial states to be one-element stacks, guaranteeing exactness of their abstraction. Second, we require that initial states/activation records belong to procedures that are never called; otherwise the abstraction might allow procedure returns even if there is no other activation record on the (concrete) stack.

Under these conditions, we get:

Theorem 2 (Optimality). *Let* $X_0 \in \wp(\mathit{State})$ *such that* $q \in X_0$ *implies that* $q = \langle c, \epsilon \rangle$ *and that* $\mathsf{proc}(c)$ *is never called by any procedure.*
Then $\mathit{reach}^f \circ \alpha^f(X_0) = \alpha^f \circ \mathit{reach}(X_0)$*. This implies that*

$$\begin{aligned} \mathit{reach}^f_{hd} \circ \alpha^f(X_0) &= \big\{\langle c, \epsilon \rangle \mid \exists \mathit{\Gamma} : \mathit{\Gamma} \cdot \langle c, \epsilon \rangle \in \mathit{reach}(X_0)\big\} \\ \mathit{reach}^f_{tl} \circ \alpha^f(X_0) &= \big\{\langle c, \epsilon \rangle \mid \exists \mathit{\Gamma}, \mathit{\Gamma}' : \mathit{\Gamma} \cdot \langle c, \epsilon \rangle \cdot \mathit{\Gamma}' \in \mathit{reach}(X_0)\big\} \end{aligned}$$

Whereas our analysis loses information on stack contents, we get *exact results* if we are interested only in the values held by variables at some control points (which is the case for many applications).

As we relate our abstract semantics directly to the standard semantics, we get by Theorem 2 the *Meet Over all Valid Paths* property defined in [28, 19], without having to introduce the notion of interprocedural valid paths. Theorems 1 and 2 are thus very similar to the optimality results in those papers.

$$pre^f_{hd}(c \xrightarrow{\langle R \rangle} c')(Y) = \left\{ \langle c, \epsilon \rangle \;\middle|\; \langle c', \epsilon' \rangle \in Y_{hd} \wedge R(\epsilon, \epsilon') \right\} \tag{4a}$$

$$pre^f_{hd}(c \xrightarrow{\langle \mathtt{call}\, \mathbf{y}{:=}P_j(\mathbf{x}) \rangle} \mathsf{s}_j)(Y) = \left\{ \langle c, \epsilon \rangle \;\middle|\; \begin{array}{l} \langle \mathsf{s}_j, \epsilon_j \rangle \in Y_{hd} \wedge \langle c, \epsilon \rangle \in Y_{tl} \\ \epsilon(\mathbf{x}^{(k)}) = \epsilon_j(\mathbf{fpi}^{(k)}) \end{array} \right\} \tag{4b}$$

$$pre^f_{hd}(\mathsf{e}_j \xrightarrow{\langle \mathtt{ret}\, \mathbf{y}{:=}P_j(\mathbf{x}) \rangle} c)(Y) = \left\{ \langle \mathsf{e}_j, \epsilon_j \rangle \;\middle|\; \begin{array}{l} \langle c, \epsilon \rangle \in Y_{hd} \wedge \epsilon_j(\mathbf{fpo}_j^{(k)}) = \epsilon(\mathbf{y}^{(k)}) \\ \forall \mathbf{x}^{(k)} \notin \mathbf{y} : \epsilon_j(\mathbf{fpi}_j^{(k)}) = \epsilon(\mathbf{x}^{(k)}) \end{array} \right\} \tag{4c}$$

$$pre^f_{tl}(\mathsf{e}_j \xrightarrow{\langle \mathtt{ret}\, \mathbf{y}{:=}P_j(\mathbf{x}) \rangle} c)(Y) = Y_{tl} \;\cup\; \left\{ \langle \mathsf{call}(c), \epsilon \rangle \;\middle|\; \begin{array}{l} \langle c, \epsilon' \rangle \in Y_{hd} \\ \forall z \notin \mathbf{y} : \epsilon(z) = \epsilon'(z) \end{array} \right\} \tag{4d}$$

$$pre^f_{tl}(c \xrightarrow{i} c')(Y) = Y_{tl} \quad \text{otherwise } (i \text{ is an instruction}) \tag{4e}$$

Table 5. Equations Defining pre^f.

3.3 Backward Analysis

Backward analysis is implemented in a similar fashion as forward analysis. Table 5 gives the definition of a correct approximation of *pre* in A^f. By defining, for $Y_0, Y \in A$, $G^f[Y_0](Y) = Y_0 \sqcup pre^f(Y)$ and $coreach^f(Y_0) = \mathrm{lfp}(G^f[Y_0])$ we get the correctness of the analysis: $coreach^f \sqsupseteq \alpha^f \circ coreach \circ \gamma^f$. Here we do not have any optimality result, as formal output parameters are not frozen during a backward execution. But we could use a similar mechanism by duality.

3.4 Comparison with the functional approach

Re-expressing the functional approach (*cf.* introduction) by abstracting the stacks of the concrete semantics presents some advantages. From a conceptual point of view, we manipulate environments instead of functions from environments to environments. The relational character of the stack abstraction is inherited from the relational character of the concrete semantics, induced by the freezing of the formal input parameters.

From an algorithmical point of view, we merge the two fixpoint computations of the traditional functional approach to a unique fixpoint computation. Thus,

- Our method is less compositional: we have to perform a new analysis for different initial states, whereas in the functional approach only the second fixpoint has to be computed again, as the predicate transformers associated with procedures are defined by the first fixpoint and do not depend on the initial states.
- It may however be more accurate: indeed, we compute at the exit point of the procedures their predicate transformers *specialised on their reachable inputs*. As the functions $post^f$ and $F^f[Y_0]$ are *not* distributive, applying them on smaller abstract values may prevent some loss of information. In addition, the data abstraction that follows the stack abstraction is often not distributive either (e.g., [20, 8]), which may increase the gain in precision.

Moreover, backward analysis can be easily defined with a stack abstraction. Backward analysis is especially useful when combined with forward analysis, when one is interested in the set of states reachable from some initial states and leading possibly to some final states, as for the debugging problem of the introduction or test selection. In this case, forward analysis from the initial states returns a set *reach* of reachable states. Then we perform a backward analysis from the final states intersected with *reach* ; practically, $\mathit{pre}^f(Y)$ is replaced by $\mathit{pre}^f(Y) \sqcap \mathit{reach}$ in the definition of $G^f[Y_0]$. The intersection *during the fixpoint iteration* allows a more accurate analysis, as the transfer functions are not distributive. This scheme can be iterated, each fixpoint computation being performed on a restricted state space. Such a scheme has been successfully applied to the verification of reactive systems [15], and we are interested in extending it to recursive programs.

4 Subsuming Functional and Call-String Approaches

Refined Stack Abstraction. In this section we combine the previous abstraction A^f with the call-string approach. We replace activation records $\langle c_n, \epsilon_n\rangle$ used in A^f by objects of the form $\langle c_0 \dots c_n, \epsilon_n\rangle$, called *extended activation records.* This corresponds to replacing single control points c_n labelling environments by sequences $c_0 \dots c_n$, called *call strings* in [28]. The abstract semantics proposed in this section can be seen as a synthesis of the two distinct methods described in [28].

Let us note $\mathit{EAct} \overset{\text{def}}{=} \mathit{Ctrl}^+ \times \mathit{LEnv}$ the set of extended activation records. We extend the notion of valid calling activation record (see Definition 1) to extended activation records. $\langle \omega, \epsilon\rangle$ is valid for $\langle \omega', \epsilon'\rangle$ if $\omega = \omega_0 \cdot c$, $\omega' = \omega_0 \cdot c \cdot c'$, and $\langle c, \epsilon\rangle$ is valid for $\langle c', \epsilon'\rangle$. This additional condition increases the precision of the analysis.

We extend the domain A^f defined in Section 3.1, (1) by replacing activation records by extended activation records, yielding the abstract domain:

$$A^s \overset{\text{def}}{=} \wp(\mathit{EAct}) \times \wp(\mathit{EAct}) \tag{5}$$

A^s is connected to $\wp(\mathit{State})$ by the Galois connection $\wp(\mathit{State}) \underset{\alpha^s}{\overset{\gamma^s}{\leftrightarrows}} A^s$, where abstraction α^s and concretisation γ^s are defined by:

$$\begin{array}{lrcl}
\alpha^s : & X & \longmapsto & \bigsqcup_{q \in X} \ \alpha^s\big(\{q\}\big) \\
 & \{\langle c_0, \epsilon_0\rangle \dots \langle c_n, \epsilon_n\rangle\} & \longmapsto & \Big\langle \ \big\{\langle c_0 \dots c_n, \epsilon_n\rangle\big\} \ , \ \big\{\langle c_0 \dots c_i, \epsilon_i\rangle \mid 0 \le i < n\big\} \ \Big\rangle \\
\gamma^s : & Y = \langle Y_{hd}, Y_{tl}\rangle & \longmapsto & \left\{ \underbrace{\langle c_0, \epsilon_0\rangle \dots \langle c_n, \epsilon_n\rangle}_{=q} \ \middle| \ \begin{array}{l} \langle c_0 \dots c_n, \epsilon_n\rangle \in Y_{hd} \\ \forall 0 \le i < n : \ \langle c_0 \dots c_i, \epsilon_i\rangle \in Y_{tl} \\ q \text{ is a consistent stack} \end{array} \right\}
\end{array}$$

A^s loses less information than A^f: thanks to call-strings, we cannot have any more the problem of (2). However we have a subtler phenomenon: if $\langle c_0, \epsilon_0\rangle$ is a valid calling activation record for $\langle c_1, \epsilon_1'\rangle$ and $\epsilon_1 \neq \epsilon_1'$ we might have

$$\langle c_0, \epsilon_0\rangle \cdot \langle c_1, \epsilon_1'\rangle \in \gamma^s \circ \alpha^s \Big(\{\langle c_0, \epsilon_0\rangle \cdot \langle c_1, \epsilon_1\rangle, \ \langle c_0, \epsilon_0'\rangle \cdot \langle c_1, \epsilon_1'\rangle\} \Big) \tag{6}$$

Hence, while the abstraction keeps the stack length exact, it can still induce some "cross-over" of activation records belonging to different concrete stacks.

Forward Analysis. The transfer function *post*^{s}, fully defined in [16], is very similar to *post*^{f} (*cf.* Tab. 4) but call-strings allow a more accurate "matching" of possible calling contexts with top contexts. For instance,

$$post^s_{hd}(\mathsf{e}_j \xrightarrow{\langle \mathtt{ret}\ \mathbf{y}:=P_j(\mathbf{x})\rangle} c)(Y) = \left\{ \langle \omega \cdot c, \epsilon' \rangle \;\middle|\; \begin{array}{l} \langle \omega \cdot \mathsf{call}(c) \cdot \mathsf{e}_j, \epsilon_j \rangle \in Y_{hd} \\ \langle \omega \cdot \mathsf{call}(c), \epsilon \rangle \in Y_{tl} \\ \epsilon(\mathbf{x}^{(k)}) = \epsilon_j(\mathbf{fpi}_j^{(k)}) \\ \epsilon' = \epsilon[\mathbf{y}^{(k)} \mapsto \epsilon_j(\mathbf{fpo}_j^{(k)})] \end{array} \right\} \qquad (7)$$

which is to be compared to (3c).

As in Section 3 and using similar definitions, the forward analysis is not only correct but also optimal. Moreover, the second condition of Theorem 2 is no longer needed, because we know that an extended activation record can return to a caller only when its call-string component is of length at least 2.

Theorem 3 (Optimality). *Let $X_0 \in \wp(State)$ be a set one-element stacks. Then we have $reach^s \circ \alpha^s(X_0) = \alpha^s \circ reach(X_0)$. This implies that*

$$reach^s_{hd} \circ \alpha^f(X_0) = \{\langle c_0 \ldots c_n, \epsilon_n \rangle \mid \langle c_0, \epsilon_0 \rangle \ldots \langle c_n, \epsilon_n \rangle \in reach(X_0)\}$$

$$reach^s_{tl} \circ \alpha^f(X_0) = \left\{\langle c_0 \ldots c_n, \epsilon_n \rangle \mid \exists \Gamma \in Act^+ : \langle c_0, \epsilon_0 \rangle \ldots \langle c_n, \epsilon_n \rangle \cdot \Gamma \in reach(X_0)\right\}$$

Discussion. The lattice A^s encompasses the two methods proposed in [28]: abstracted to A^f, it corresponds to the functional approach of that paper, whereas A^s, further abstracted with a data abstraction *which does not relate* the values of input parameters and output parameters, would give the call-string approach. As told in the introduction, A^s leads to an interprocedural analysis which is both control and data context-sensitive. Here we used the call-strings of [28], but this could be generalised to the tokens of [18]. Backward analysis can of course be defined in the lattice A^s similarly as in Section 3.

5 Data Abstractions

Our stack abstractions in some way define each an interprocedural analysis method, but not an effective analysis, due to infinite data values or unbounded size of call strings. We show here which existing abstract lattices are suitable for these methods. [16] shows how to extend them with procedure call and return operations. We will consider only A^s, since A^f is a further abstraction of A^s.

We can use the isomorphism $A^s \simeq Ctrl \to \big(\wp(Ctrl^* \times LEnv)\big)^2$ to associate a set of values to each control point (as usual when analysing imperative programs). *Ctrl* being finite, we just need an abstract lattice for $\wp(Ctrl^* \times LEnv)$ in order to get an abstract lattice for A^s. A natural way to achieve this is to build an abstract lattice for $\wp(Ctrl^* \times LEnv)$ by composing abstract domains available for $\wp(Ctrl^*)$ and $\wp(LEnv)$. So we need to abstract $\wp(Ctrl^*)$ and $\wp(LEnv)$ by some lattices A_{Ctrl} and A_{LEnv}, as well as a method for combining them.

Abstractions for call-strings. Several abstract domains are possible for A_{Ctrl}:

1. $\wp(Ctrl^p)$, for some fixed $p \geq 0$: the top p elements of stacks are exactly represented, the others are completely forgotten,

2. $\mathsf{Reg}(\mathit{Ctrl})$, the set of regular languages over the finite alphabet Ctrl, together with a suitable widening operator, as the one suggested by [13] or
3. some subsets of regular languages: simple regular languages, star-free regular languages, etc., with widening operators if necessary.

Notice that apart the first one, these abstractions for call strings are more general than the one suggested in [28], where only finite partitioning of $\wp(\mathit{Ctrl}^*)$ is considered. Here we do not restrict abstract lattices for $\wp(\mathit{Ctrl}^*)$ neither to finite lattices nor to finite-height lattices, thanks to the use of widening.

Abstractions for environments. Any abstract lattice A_{LEnv} available for intraprocedural analysis can be chosen here [8, 20, 10, 27, 17]. However *it should be relational* in order to be effectively able to represent relations between input and output parameters at the exit point of procedures. In particular equality constraints should be possible in A_{LEnv} for implementing an accurate procedure return operation. The typical counter-example is the lattice of intervals for numerical variables, where no relationships between variables can be represented, only a conjunction of invariants for each variable.

Combining the two abstract lattices. To avoid the inherent difficulties of the design of an abstraction combining different data-types (in our case Ctrl^* and LEnv), we suggest to combine the lattices abstracting each datatype. We could use the tensor product $A_{\mathit{Ctrl}} \otimes A_{LEnv}$ [21]. However, this product is not finitely representable as soon as either A_{Ctrl} or A_{LEnv} is infinite. Instead, we suggest the simple solution of [15], which is to take the Cartesian product $A = A_{\mathit{Ctrl}} \times A_{LEnv}$. In this lattice, relationships between call strings and data values cannot be directly represented: an abstract value is just the conjunction of an invariant on call strings and an invariant on data values. However, partitioning on A can be used to establish relationships between the two components. This technique has been used for obtaining relational invariants on Boolean and numerical variables in [15], and can be considered as a particular instance of the disjunctive or down-set completion method discussed in [7], where the size of disjuncts is bounded by the size of the partition. For instance, if we partition $A_{\mathit{Ctrl}} \times A_{LEnv}$ according to the k last control points in call-strings, we get a lattice isomorphic to $\mathit{Ctrl}^k \rightarrow A_{\mathit{Ctrl}} \times A_{LEnv}$.

Example. Fig. 3 gives a program computing MacCarthy's 91-function and the analysis results, with as initial states $\langle s, n \in \mathbb{Z} \rangle$. The exact denotational semantics of this function is $\lambda n.(\mathit{if}\ n > 101\ \mathit{then}\ n-10\ \mathit{else}\ 91)$. We use the lattice $A=(\mathit{Ctrl} \cup \mathit{Ctrl}^2)\rightarrow\big(\mathsf{Reg}(\mathit{Ctrl})\times\mathsf{Pol}(\mathbb{Z})\big)^2$, i.e. we partition A w.r.t. the two top control points in the call-strings, and we use convex polyhedra for numerical variables. As widening operator on $\mathsf{Reg}(\mathit{Ctrl})$, we use the bisimulation on automata of order δ, $\sim_\delta$, as in [13], with $\delta=1$. Starting analysis with a one-element stack, we have $Y_{tl}=Y_{hd}$ for all call-sites and $Y_{tl}=\emptyset$ elsewhere. Thus Fig. 3 shows only Y_{hd}. Observe that the result at point e is both control and data context-sensitive: the partitioning on call-strings induces a differentiation of the possible values of the input n. We do not find the exact result at point e, because of the convex

```
proc MC(n: int): (r: int) is
  t1, t2: int;
begin s: $\{\langle s, ?\rangle,\ \langle \omega c_3 s, n \le 111\rangle,\ \langle \omega c_4 s, 91 \le n \le 101\rangle\}$
  if n > 100 then c_0: $\{\langle c_0, n \ge 101\rangle, \langle \omega c_3 c_0, 101 \le n \le 111\rangle, \langle \omega c_4 c_0, n = 101\rangle\}$
    r := n - 10 c_1: $\{\langle c_1, r = n-10 \ge 91\rangle, \langle \omega c_3 c_1, 101 \le n \le 111 \wedge r = n-10\rangle, \langle \omega c_4 c_1, r = n-10 = 91\rangle\}$
  else c_2: $\{\langle c_2 + \omega c_3 c_2, n \le 100\rangle\}, \langle \omega c_4 c_2, 91 \le n \le 100\rangle\}$
    t1 := n + 11; c_3: $\{\langle c_3 + \omega c_3 c_3, n \le 100 \wedge t1 = n+11\rangle, \langle \omega c_4 c_3, 91 \le n \le 100 \wedge t1 = n+11\rangle\}$
    t2 := MC(t1); c_4: $\left\{\begin{matrix}\langle c_4 + \omega c_3 c_4, n \le 100 \wedge t1 = n+11 \wedge 91 \le t2 \le 101 \wedge t2 \ge n+1\rangle, \\ \langle \omega c_4 c_4, 91 \le n \le 100 \wedge t1 = n+11 \wedge 91 \le t2 \le 101 \wedge t2 \ge n+1\rangle\end{matrix}\right\}$
    r  := MC(t2); c_5: $\left\{\begin{matrix}\langle c_5 + \omega c_3 c_5, n \le 100 \wedge t1 = n+11 \wedge 91 \le t2 \le 101 \wedge t2 \ge n+1 \wedge r = 91\rangle, \\ \langle \omega c_4 c_5, 91 \le n \le 100 \wedge t1 = n+11 \wedge 91 \le t2 \le 101 \wedge t2 \ge n+1 \wedge r = 91\rangle\end{matrix}\right\}$
  endif
end MC e: $\{\langle e, r \ge 91\rangle, \langle \omega c_3 e, n \le 111 \wedge 91 \le r \le 101 \wedge r \ge n-10\rangle, \langle \omega c_4 e, 91 \le n \le 101 \wedge r = 91\rangle\}$
```

Fig. 3. MacCarthy's 91-function (where $\omega \overset{\text{def}}{=} (c_3 + c_4)^*$).

hull on polyhedra performed on the results at points c_1 and c_5, which are *exact*. In particular, although the result at point c_5 depends on the approximate result at point e, it is exact thanks to the context-sensitiveness of the analysis. Clearly, partitioning A w.r.t. only the top control point would have lead to much less precise information, because of the more frequent use of convex hull.

6 Related Work

As explained in the introduction, one can distinguish two main approaches to interprocedural static analysis, namely the *functional* and the *operational*. The functional approach of [6] has been used for instance to analyse the access to arrays in interprocedural Fortran programs [9], using the abstract domain of convex polyhedra [8]. [22] can be seen as an algorithmical implementation of [19] using graph reachability techniques, which can be used with finite lattices and distributive data flow functions. This technique can be applied to all *bit-vector* analyses and the Bebop tool [1] is based on it. An extension [26] allows to tackle some finite-height infinite lattices, like (linear) constant propagation.

In the operational approach, [3] considers more complex Pascal programs with reference parameter passing, which introduces aliasing on the stack (i.e. several variables may refer to the same location in the stack), and nested procedure definitions. Unsurprisingly, the devised solution is quite complex. It has been implemented using the interval domain for integers [5]. Stacks are collapsed more severely than in our model. The proposal of [11], implemented in Moped [12] and applied to concurrent programs in [2], relies on the result that the set of reachable stacks of a pushdown automata is a regular language, that can be represented by finite-state automata. The analysed program is converted to a pushdown automaton, and is thus restricted to programs manipulating finite-state variables/properties, or requires the finite abstraction of data/properties prior the analysis. A recent extension allows the use of some infinite finite-height lattices [23] and represents a very interesting mix of the two approaches: pushdown automata are here extended by associating transformers to transition rules. This allows to encode the control part of the program and properties belonging to a finite lattice in the pushdown automata, whereas properties belonging to a finite-height lattice can be handled by the transformers attached to the transitions. Finally, [24] directly represents the stack as a linked list of activation records, using the shape analysis of [27].

7 Conclusion

In this paper, we presented an approach to the verification of imperative programs with recursive procedures and variables over possibly infinite domains. We showed that by relying solely on the principles of abstract interpretation, one can derive from the standard semantics of a program an interprocedural analysis method. This is done by abstracting in a proper way sets of call-stacks of the programs. Such an interprocedural analysis method can then be implemented after a second data abstraction. We defined two stack abstractions, the optimality of which suggests that they are good starting points for the following data abstraction. The first one is equivalent to the functional approach, but offers in addition the specialisation of predicate transformers and clarifies the intersection between forward and backward analysis.

The second stack abstraction, which is both data and control context sensitive, integrates the two approaches distinguished in [28]. It allows to specify complex constraints on the stack yielding an analysis which contains strictly more information. This is particularly useful for starting an analysis from initial or final states with a non-empty call-stack and also makes the combination of forward and backward analysis more efficient, as more information can be used for intersecting the two analyses.

It could be argued that our assumptions on the analysed programs are too restrictive. Non-local jumps could be easily added, as in [18], although this would suppress our current optimality results. Allowing reference parameter passing and handling aliasing on the stack would be very useful to tackle C programs. However, it should be noted that all the general approaches, with the notable exception of [3], assume like ourselves that intraprocedural instructions modify only the top activation record in the stack. Thus they cannot tackle directly reference parameter passing. The simplest way for adding this feature in our case would be to add aliasing information in activation records and to use it to properly update variables passed by reference upon procedure returns.

An implementation of our analysis is under work, targeted to programs with enumerated or numeric variables, following the implementation guidelines described in [16]. The next step would be its application to programs manipulating pointers to dynamically allocated objects, using abstract domains such as the one of [27].

References

1. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN 2000 Workshop*, LNCS 1885, pages 113–130. Springer, 2000.
2. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL'03*, pages 62–73. ACM, 2003.
3. F. Bourdoncle. Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity. In *PLILP'90*, LNCS 456, pages 307–323. Springer, 1990.
4. F. Bourdoncle. Assertion-based debugging of imperative programs by abstract interpretation. In *ESEC'93*, LNCS 717, pages 501–516. Springer, 1993.

5. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252. ACM, 1977.
6. P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In *Formal Description of Programming Concepts*, pages 237–277. North Holland, 1977.
7. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3), 1992.
8. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL'78*, pages 84–97. ACM, 1978.
9. B. Creusillet and F. Irigoin. Interprocedural array region analyses. *International Journal of Parallel Programming*, 24(6), 1996.
10. A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *PLDI'94*, pages 230–241. ACM, 1994.
11. J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *FoSSaCS'99*, LNCS 1578, pages 14–30. Springer, 1999.
12. J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *CAV'01*, LNCS 2102, pages 324–336. Springer, 2001.
13. J. Feret. Abstract interpretation-based static analysis of mobile ambients. In *SAS'01*, LNCS 2126, pages 412–430. Springer, 2001.
14. F. Gaucher, E. Jahier, B. Jeannet, and F. Maraninchi. Automatic state reaching for debugging reactive programs. In *AADEBUG'03*, 2003.
15. B. Jeannet. Dynamic partitioning in linear relation analysis. Application to the verification of reactive systems. *Formal Methods in System Design*, 23(1), 2003.
16. B. Jeannet and W. Serwe. Abstracting call-stacks for interprocedural verification of imperative programs. Research Report 4904, INRIA, July 2003.
17. T. Jensen and F. Spoto. Class analysis of object-oriented programs through abstract interpretation. In *FoSSaCS'01*, LNCS 2030, pages 261–275. Springer, 2001.
18. N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *POPL '82*. ACM, 1982.
19. J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *CC'92*, LNCS 641, pages 125–140. Springer, 1992.
20. A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, pages 310–319. IEEE, 2001.
21. F. Nielson. Tensor products generalize the relational data flow analysis method. In 4^{th} *Hungarian Computer Science Conference*, 1985.
22. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL'95*, pages 49–61. ACM, 1995.
23. T. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *SAS'03*, LNCS 2694. Springer, 2003.
24. N. Rinetzky and M. Sagiv. Interprocedural shape analysis for recursive programs. In *CC'01*, LNCS 2027, pages 133–149. Springer, 2001.
25. V. Rusu, L. du Bousquet, and T. Jéron. An approach to symbolic test generation. In *IFM'00*, LNCS 1945, pages 338–357. Springer, 2000.
26. M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *TCS*, 167(1–2):131–170, 1996.
27. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM ToPLaS*, 24(3), 2002.
28. M. Sharir and A. Pnueli. Semantic foundations of program analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7. Prentice Hall, 1981.