Interprocedural Abstract Interpretation of Block Structured Languages with Nested Procedures, Aliasing and Recursivity

FRANÇOIS BOURDONCLE

Laboratoire d'Informatique de l'École Polytechnique (LIX)*
91128 Palaiseau Cedex, France
bourdonc@polytechnique.fr

1 Introduction

Since the birth of abstract interpretation [Cousot 77, Cousot 81], many languages have been studied, and frameworks for their abstract interpretation have been proposed. The first languages considered by the Cousots were very simple, procedureless, FORTRAN-like languages, and were used to set up the theory. The main characteristic of this class of languages is that there is a one to one correspondence between identifiers and locations. A store can therefore be represented as a vector of a space of finite dimension. Approximating a set of memory states at a given control point thus consists in finding an abstraction (i.e. a machine-representable lattice) of this vectorial space.

New problems arise however when considering a language such as Pascal, which has nested procedures, call-by-reference and recursivity, for there can be many activations of the same procedure in the run-time stack, each activation having its own environment binding identifiers to different locations anywhere in the stack by means of the call-by-address parameter passing method. The problem is therefore not only to approximate the set of values of a fixed vector of variables, but to approximate sets of run-time stacks, each stack having its own aliasing structure and being of arbitrary height.

The aim of this paper is to study Pascal-like languages, and to show that it is possible to design and prove abstract interpretations used to determine assertions about scalar variables, as was formerly done for simpler languages. We shall then talk about the implementation of a specific "semantic analyzer" of Pascal used to determine the value range of integer variables and give some examples of the results given by this analyser.

*This work was supported by *Esprit B.R.A.*, action 3124 "Sémantique".

2 Methodology

The language we are considering is basically pure Pascal. We do not consider procedural parameters and jumps outside the current block. This is only for a simplicity purpose; "long jumps" can be handled in our framework and are actually implemented. Procedural parameters will be considered in another paper.

The main idea behind our work is that the key problem in the abstract interpretation of Pascal is that although the environment can bind identifiers to any location in the run-time stack, the only information needed to perform a safe abstract interpretation of the body of a procedure is the *partition of its variables into sets of variables sharing the same location.* An idea can thus be to bind all variables of a given *procedure activation*, sharing the same location in the stack, to a temporary location, allocated by the called procedure, to execute the procedure body in this new environment, and to update the actual location (using the value of the temporary at the end of the procedure) when returning from the call. This parameter passing method seems to be similar to the *call-by-value-result* method (also known as *copy-restore linkage*, or *copy-in copy-out*). It is well known however that call-by-reference and call-by-value-result are not equivalent. The originality of our scheme is that the substitution from one parameter passing method to the other is *dynamic.* It cannot be done at compile time, unless an abstract interpretation is performed. This program transformation is illustrated in figure 1 by a program computing MacCarthy's 91 function, recursively defined by:

$$f(x) = \begin{cases} x - 10 & \text{if } x > 100 \\ f(f(x+11)) & \text{otherwise} \end{cases}$$

It can be shown by induction that $f(x) = x - 10$ if $x > 101$ and $f(x) = 91$ otherwise. Our semantic analyser has proven that $x \in [91, hi - 10]$ [1] at control point $\{2\}$, which is the best result that can be expected using the integer range lattice. The main procedure has been duplicated three times: in procedure `Mc1`, the formal parameter `r` is a lexical alias of `x` and has been replaced by `x`, and in procedures `Mc2` and `Mc3`, `r` is a "hidden" alias of `t` and `r` itself. It is noteworthy that the induction process performed by our system automatically discovers the induction cases of the hand made proof. These cases define a *finite* sublattice I of the integer range lattice which enable an automatic proof of the program using the finite lattice $I \to I$.

In order to define more precisely the program transformation we have presented and prove its correctness, we shall first define the operational semantics of a block structured language with nested procedures, call-by-value and call-by-address, and recursivity. Only the most important parts of the semantics will be defined, such as the stack structure, the address of an identifier and the procedure call semantics. We shall then define another semantics with a clean stack structure better suited to abstract interpretation and show that these two semantics are equivalent. At last, we shall define a general method for building safe approximations of stacks sets and show how these approximations can be tuned to get an abstract interpretation of the desired precision.

[1] Where $hi = \texttt{maxint}$, and $lo = -\texttt{maxint} + 1$ are the upper and lower bounds of the Pascal `integer` type.

```
program CallByAddress;

    var x : integer;

    procedure Mc(n : integer; var r : integer);

        var t : integer;

    begin
        if n > 100 then
            r := n - 10
        else begin
            Mc(n+11, t);
            Mc(t, r)
        end
    end;

begin
{1}
    Mac(x, x);
{2}
end.
```

```
program CallByValueResult;
    var x : integer;
    procedure Mc1(n : integer);
        var t : integer;
    begin   { n ∈ [lo, hi] }
        if n > 100 then x := n-10
        else begin Mc2(n+11, t); Mc1(t) end
    end;    { x ∈ [91, hi − 10] }
    procedure Mc2(n : integer; inout r : integer);
        var t : integer;
    begin   { n ∈ [lo + 11, 111] }
        if n > 100 then r := n-10
        else begin Mc2(n+11, t); Mc3(t, r) end
    end;    { r ∈ [91, 101] }
    procedure Mc3(n : integer; inout r : integer);
        var t : integer;
    begin   { n ∈ [91, 101] }
        if n > 100 then r := n-10
        else begin Mc2(n+11, t); Mc3(t, r) end
    end     { r = 91 }
begin
{1} { x ∈ [lo, hi] }
    Mc1(x)
{2} { x ∈ [91, hi − 10] }
end.
```

Figure 1: MacCarthy's 91 function

3 Definitions and notations

The basic domains we shall use are *Proc* (procedure names), *Local* (local and global variables), *Value* (call-by-value formal parameters), *Alias* (call-by-address formal parameters). The domain $Ident = Local + Value + Alias$ is thus the domain of variables and formal parameters. Note that there are no constants in our language. The domain $SVal = Integer + Boolean$ is the domain of scalar values. All names are built using a full "pathname" prefix in order to avoid conflicts, such as, for instance *Prog.Foo.Bar* if *Bar* is defined within *Foo*. The *Proc* domain is thus a CPO, of infimum *Prog*, for the following partial order.

Definition 1 $Prog.P_1.....P_n \leq Prog.P'_1.....P'_{n'} \iff n \leq n' \wedge \forall k \leq n : P_k = P'_k$

Definition 2 *The most closely nested procedure around procedure* $P = Prog.P_1.....P_n$ *is noted* $P^\bullet = Prog.P_1.....P_{n-1}$. *For a global procedure* P, *we define* $P^\bullet = Prog$. *We also note* $x^\bullet$ *the procedure in which the identifier* x *is defined (or Prog if* x *is a global variable).*

Definition 3 *We say that* $x \equiv y$ *if* $x^\bullet = y^\bullet$. *If a variable* x *is defined in procedure* P, *i.e. if* $x^\bullet = P$, *we also say that* $x \equiv P$. *If* x *is lexically accessible to* P, *i.e. if* $x^\bullet \leq P$, *we say (incorrectly) that* $x \leq P$.

Definition 4 *For a set of identifiers λ, we say that $\lambda < P$ if : $\forall x \in \lambda, x < P$, and that $\lambda \equiv P$ if : $\forall x \in \lambda, x \equiv P$*

Definition 5 *Let $f : A \rightarrow B$. We note $f^* : \mathcal{P}(A) \rightarrow \mathcal{P}(B)$ the pointwise extension of f defined by : $f^*(S) = \{f(s)\}_{s \in S}$, and $f_{|S}$ the restriction of f to $S \subseteq A$. If $f^*(S) = \{x\}$, then the value x will be noted $\overline{f}(S)$. At last, if $A = A_1 \times \ldots \times A_n$ and $X = \langle x_1, \ldots, x_n \rangle$ then: $\forall k \in [1, n], X_{\downarrow k} = x_k$.*

We also need some notations in order to handle procedure calls. For a call from procedure P_n to procedure P_{n+1}, and for a formal parameter x of procedure P_{n+1}, we define Π_{n+1} : $Alias + Value \rightarrow Ident + SVal$ by $\Pi_{n+1}(x) = y$ if x is a call-by-address formal parameter and the actual parameter is the identifier y, and $\Pi_{n+1}(x) = v$ if x is a call-by-value formal parameter and the value of the corresponding argument is v.

4 Standard operational semantics

The specific domains used in the standard operational semantics are listed below. Note that a context is local to its activation. The domain *Control* is the flat domain of the control points of procedure bodies.

$$\begin{array}{lcl} Stack & = & Act^* \\ Act & = & Proc \times Control \times Context \\ Context & = & Store \times Ref \\ Store & = & Ident \rightarrow SVal_\perp \\ Ref & = & Ident \rightarrow Ident_\perp \end{array}$$

For each stack $\Sigma \in Stack$ of height n we use the notation $\Sigma_{\downarrow k} = \langle P_k, c_k, \langle \sigma_k, \mu_k \rangle \rangle$, with $k \in [0, n]$.

Definition 6 *During a call of procedure P_{n+1} from procedure P_n , the context $\langle \sigma_{n+1}, \mu_{n+1} \rangle$ of the new activation appended to the stack is defined by : $\forall x \equiv P_{n+1}$*

$$\begin{array}{lcl} \sigma_{n+1}(x) & = & (x \in Value) \rightarrow \Pi_{n+1}(x), \perp \\ \mu_{n+1}(x) & = & (x \in Ref) \rightarrow \Pi_{n+1}(x), \perp \end{array}$$

otherwise $\sigma_{n+1}(x) = \mu_{n+1}(x) = \perp$.

It is then clear that the nature of an identifier x at the level k in the stack is:

- If $\mu_k(x) \in Ident$ then x is a call-by-address formal parameter, and $\mu_k(x)$ is the actual parameter used in the call from P_{k-1} to P_k.
- If $\mu_k(x) = \perp$ then if $\sigma_k(x) \in SVal$, then x is either a local variable or a call-by-value parameter, otherwise x is not defined in P_k.

We define the function $Address : I\!N \times Ident \rightarrow (I\!N \times Ident)_\perp$ giving the location of a variable in the stack by:

$$\begin{array}{lcl} Address(\langle n, x \rangle) & = & x \not\equiv P_n \rightarrow (n = 0) \rightarrow \perp, \; Address(\langle \Lambda(n), x \rangle), \\ & & x \in Ref \rightarrow (n = 0) \rightarrow \perp, \; Address(\langle n - 1, \mu_n(x) \rangle), \\ & & \langle n, x \rangle \end{array}$$

where $\Lambda(k)$ is the *access link*, that is the stack index of the most recent activation of $P_k^\bullet$:

$$\Lambda(k) = max\ \{i \in [0, k-1]\ : P_i = P_k^\bullet\}$$

This access link is used to access non local variables on the stack, and is implicitly defined in our language for we do not allow procedural parameters[2]. The lexical scope rule used in Pascal says that, in order to be called by procedure P_k, procedure P_{k+1} must be accessible to P_k. A run-time stack must thus verify: $\forall k \in [0, n-1]\ : P_{k+1}^\bullet \leq P_k$. It can therefore be proven that the access link is well defined for any run-time stack.

In order to access non local names on the stack, we also define the set $\mathcal{L}(n) = \{n, \Lambda(n), \Lambda^2(n), \ldots, 0\}$ of the stack indexes corresponding to the activations of P_n and of its enclosing procedures[3]. For any enclosing procedure $P \leq P_n$ we can then define $\Lambda(n, P)$ as being the unique element in $\mathcal{L}(n)$ such that $P = P_{\Lambda(n,P)}$.

The value of a variable $x \leq P_n$ is now defined as $Value(\Sigma, x) = \sigma_k(y)$, where $\langle k, y\rangle = Address(\langle n, x\rangle)$, and we define $Update(\Sigma, x, v) = \Sigma\,[\sigma_k[y \mapsto v]/\sigma_k]$.

Returning from a procedure call simply consists in popping the most recent activation from the stack.

5 An equivalent operational semantics

5.1 Procedure call semantics

The domains used for this new semantics are listed below. In order not to use heavy notations, we shall refer, when necessary, to the former domains and functions using primes.

$$\begin{array}{lcl} Stack & = & Act^* \\ Act & = & Proc \times Control \times Context \\ Context & = & Env \times Out \times Store \\ Env & = & Ident \rightarrow Loc \\ Store & = & Loc \rightarrow SVal_\perp \\ Out & = & Loc \rightarrow Loc \\ Loc & = & \mathcal{P}(Ident) \end{array}$$

For each stack $\Sigma \in Stack$ we use the notation $\Sigma_{\downarrow k} = \langle P_k, c_k, \langle \varepsilon_k, \omega_k, \sigma_k\rangle\rangle$ where the third component of a local context $\omega_k \in Out$ is a new store-like function used to manage call-by-address parameters.

The environment ε_n is local to procedure P_n, so we have to define the environment $\mathrm{E}_n : Ident \rightarrow (I\!N \times Loc)_\perp$ binding any identifier accessible to P_n to its location:

$$\forall x \in Ident, \forall k \in [0, n]\ :\ \mathrm{E}_k(x) = \ (x \not\equiv P_k)\ \rightarrow\ \begin{array}{c}(k = 0)\\ \langle k, \varepsilon_k(x)\rangle\end{array}\ \rightarrow\ \perp,\ \mathrm{E}_{\Lambda(k)}(x),$$

[2] See [ASU 86] Section 7.3 for more details about the access link.

[3] See figure 2 for an example.

When $\mathrm{E}_k(x) \neq \perp$, it should be clear that $\mathrm{E}_k(x) = \langle i, \varepsilon_i(x)\rangle$, where $i = \Lambda(k, x^\bullet)$. We can now define the local environment ε_{n+1} pushed on top of the stack during a call from procedure P_n to procedure P_{n+1} by:

$$\begin{array}{rcll} \varepsilon_{n+1}(x) & = & (x \in \mathit{Local}) & \rightarrow \{x\}, \quad (1) \\ & & (x \in \mathit{Value}) & \rightarrow \{x\}, \quad (2) \\ & & (\lambda < P_{n+1}) & \rightarrow \lambda, \quad (3) \\ & & & \{y \in \mathit{Alias} : (y \equiv P_{n+1}) \wedge (\mathrm{E}_n(\Pi_{n+1}(y))_{\downarrow 2} = \lambda)\} \quad (4) \\ & & & \text{where } \lambda = \mathrm{E}_n(\Pi_{n+1}(x))_{\downarrow 2} \end{array}$$

(and $\varepsilon_{n+1}(x) = \emptyset$ if $x \not\equiv P_{n+1}$)

We say that a call-by-address formal parameter $x \equiv P_{k+1}$ aliases a location λ if the environment E_k of the calling procedure P_k binds the actual parameter $\Pi_{k+1}(x)$ to an address of the form $\langle i, \lambda\rangle$, where $i \in \mathcal{L}(k)$.

Case (3) refers to the situation where the formal parameter x aliases a location created by an enclosing procedure of P_{n+1}. A location is said to be created by procedure P_{n+1} in cases (1), (2), and (4). In case (3), the location is said to be *handed* to P_{n+1}. In case (4), the location λ aliased by x has been created by a procedure P_k not accessible to P_{n+1}. One can consider λ as being "hidden" in the stack. The only way to update this location is to assign a call-by-address formal parameter that belongs to the location $\varepsilon_{n+1}(x)$ defined in case (4), considered as a set. It can be shown by induction that for any stack:

Theorem 7 *For any location λ such that $\varepsilon_k^{-1}(\lambda) \neq \emptyset$, either $\lambda \subseteq \mathit{Local}$, $\lambda \subseteq \mathit{Value}$ or $\lambda \subseteq \mathit{Alias}$.*

Whenever $\varepsilon_k^{-1}(\lambda) \neq \emptyset$, λ will be called a *pseudo-location* if $\lambda \subseteq \mathit{Alias}$ and an *actual location* otherwise.

Theorem 8 *Let λ be a location such that $\varepsilon_{n+1}^{-1}(\lambda) \neq \emptyset$ and $\lambda \subseteq \mathit{Alias}$. Then:*

$$\exists k \in \mathcal{L}(n+1) - \{0\} : (\lambda \equiv P_k \wedge \forall (x, y) \in \lambda^2 : E_{k-1}(\Pi_k(x)) = E_{k-1}(\Pi_k(y)))$$

The integer k will be noted $\Lambda(n+1, \lambda)$ by analogy with the previous definition of Λ. Theorem 8 and Definition 5 enable the following definitions of ω_{n+1} and σ_{n+1}:

$$\begin{array}{rcll} \omega_{n+1}(\lambda) & = & (\lambda \equiv P_{n+1}) \wedge (\varepsilon_{n+1}^{-1}(\lambda) \neq \emptyset) \wedge (\lambda \subseteq \mathit{Alias}) & \rightarrow \overline{\mathrm{E}_n \circ \Pi_{n+1}}(\lambda)_{\downarrow 2}, \ \emptyset \\ \sigma_{n+1}(\lambda) & = & (\lambda \equiv P_{n+1}) \wedge (\varepsilon_{n+1}^{-1}(\lambda) \neq \emptyset) \wedge (\lambda \subseteq \mathit{Value}) & \rightarrow \overline{\Pi_{n+1}}(\lambda), \ \perp \end{array}$$

5.2 Stack structure

The interpretation of a pseudo-location λ in the local context $\langle \varepsilon_k, \omega_k, \sigma_k\rangle$ is the following:

- If $\omega_k(\lambda) = \lambda'$ then $\sigma_k(\lambda)$ is meaningless, λ' is the location of the calling procedure P_{k-1} (or one of its enclosing procedures) aliased by λ, and the identifiers in λ' are not accessible to the enclosing procedures of P_k. Moreover, λ is the set of call-by-address formal parameters

```
program Stack;
   var   g : integer;
   procedure Q(var  q1, q2, q3 : integer);
      procedure P(var  p1, p2 : integer);
         var   p3 : integer;
      begin
{4}      Q(p3, p3, q3);
      end;
   begin
{5}   ...
{3}   P(q1, q2)
      ...
   end;
   procedure R(var  r1, r2, r3 : integer);
   begin
{2}   Q(r1, r2, r3)
   end;
   procedure S(var s1 : integer);
      var   s2 : integer;
   begin
{1}   R(s1,s2,s2)
   end;
begin
{0}   S(g)
end.
```

Procedure	Ctrl	Identifier x	$\lambda = \varepsilon_k(x)$	$\omega_k(\lambda)$
Prog.Q	5	q_1 q_2	$\{q_1, q_2\}$	$\{p_3\}$
		q_3	$\{q_3\}$	$\{q_3\}$
Prog.Q.P	4	p_1 p_2	$\{q_1, q_2\}$	$\emptyset$
		p_3	$\{p_3\}$	$\emptyset$
Prog.Q	3	q_1 q_2	$\{q_1, q_2\}$	$\{p_3\}$
		q_3	$\{q_3\}$	$\{q_2, q_3\}$
Prog.Q.P	4	p_1	$\{g\}$	$\emptyset$
		p_2	$\{q_2, q_3\}$	$\emptyset$
		p_3	$\{p_3\}$	$\emptyset$
Prog.Q	3	q_1	$\{g\}$	$\emptyset$
		q_2 q_3	$\{q_2, q_3\}$	$\{r_2, r_3\}$
Prog.R	2	r_1	$\{g\}$	$\emptyset$
		r_2 r_3	$\{r_2, r_3\}$	$\{s_2\}$
Prog.S	1	s_1	$\{g\}$	$\emptyset$
		s_2	$\{s_2\}$	$\emptyset$
Prog	0	g	$\{g\}$	$\emptyset$

Run-time stack

$\mathcal{L}(7) = \{7,0\}$ $\quad$ $\mathcal{L}(3) = \{3,0\}$
$\mathcal{L}(6) = \{6,5,0\}$ $\quad$ $\mathcal{L}(2) = \{2,0\}$
$\mathcal{L}(5) = \{5,0\}$ $\quad$ $\mathcal{L}(1) = \{1,0\}$
$\mathcal{L}(4) = \{4,3,0\}$ $\quad$ $\mathcal{L}(0) = \{0\}$

Access links

Figure 2: Run-time stack structure

of the called procedure P_k bound to λ by the environment ε_k. We shall see that the pseudo-location λ can be used as a temporary to store the value of these formal parameters during the execution of the procedure body.

- If $\omega_k(\lambda) = \emptyset$ then λ is still a pseudo-location, but it was created earlier by an enclosing procedure of P_k. Identifiers bound to λ by ε_k can be seen as pure lexical aliases of the identifiers in λ (considered as a set).

An example of this stack structure is given in figure 2. Identifier names are written without their path prefix for simplicity.

The location given by the environment E_n is an actual location (i.e. a location used to store a value) only for local variables and call-by-value parameters, so we need to define a function $Locate : (IN \times Loc) \rightarrow (IN \times Loc)_\perp$ giving the actual location to which call-by-address parameters are really bound, by descending the pseudo-location string:

$$\begin{array}{rcl} Locate(\langle k, \lambda \rangle) & = & \lambda \not\equiv P_k \quad \rightarrow \quad (k = 0) \quad \rightarrow \quad \perp, \quad Locate(\langle \Lambda(k), \lambda \rangle) \\ & & \lambda \subseteq Alias \quad \rightarrow \quad (k = 0) \quad \rightarrow \quad \perp, \quad Locate(\langle k-1, \omega_k(\lambda) \rangle) \\ & & \qquad\qquad\quad \langle k, \lambda \rangle \end{array}$$

Definition 9 *The address of an identifier x is defined by:* $Address(\langle n, x\rangle) = Locate(E_n(x))$

The *Value* and *Update* functions are easily defined from *Address* as in the former semantics.

5.3 Equivalence of the two semantics

In the former semantics, μ_k defined a set of trees over *Ident*, the roots of which were the local/global variables or the call-by-value formal parameters being aliased by the other call-by-address formal parameters in the tree. The idea behind the latter semantics is to *lift* these trees in order to gather call-by-address formal parameters of the same procedure having the same address into nodes — being called locations —, while preserving the structure of each tree. A comparison between these two kinds of trees is given in figure 3. The actuals locations are drawn in a gray rectangle, whereas the pseudo-locations are drawn in a black rectangle. With this in mind, one can prove by induction the following important theorem.

Theorem 10 *For any variable $x \leq P_n$:*

$$Address'(\langle n, x\rangle) = \langle k, y\rangle \Longleftrightarrow Address(\langle n, x\rangle) = \langle k, \{y\}\rangle$$

This theorem shows that the actual locations are exactly the same in the two semantics. Therefore, it is not too difficult to see that the two semantics are equivalent.

5.4 A new procedure call semantics

The main theorem about the latter semantics is the following.

Theorem 11 *Let $x \neq y$ be two variables, $x \leq P_n$ and $y \leq P_n$. Then:*

$$Address(\langle n, x\rangle) = Address(\langle n, y\rangle) \Longleftrightarrow E_n(x) = E_n(y)$$

The meaning of this theorem is that two variables accessible to the current procedure are aliases if and only if the environment binds them to the same location (or pseudo-location). In other words, one does not have do go down deep into the stack to know whether or not two identifiers share the same actual location. The reduced aliasing tree has lifted this information up to the current procedure activation.

This implies that *for a given stack context* we can freely change the call-by-address binding into a kind of call-by-value-result binding without modifying the semantics of the language. Note that since the switch from one binding pattern to another is context-sensitive, it does not mean that call-by-address and call-by-value-result are equivalent. Pseudo-locations that were previously used as pointers to other locations will now be used as temporaries during the execution of the procedure body. Thanks to Theorem 11, there will be no memory conflict. Aliased locations will be updated when returning from the called procedure.

6 Abstracting stack sets

In order to perform an abstract interpretation of our language, we must define an *abstraction function*[4] $\alpha : (\mathcal{P}(Stack), \subseteq, \cup, \cap) \rightarrow (\overline{Stack}, \sqsubseteq, \sqcup, \sqcap)$ giving, for each stack set S, a safe approxi-

[4] See [Cousot 81].

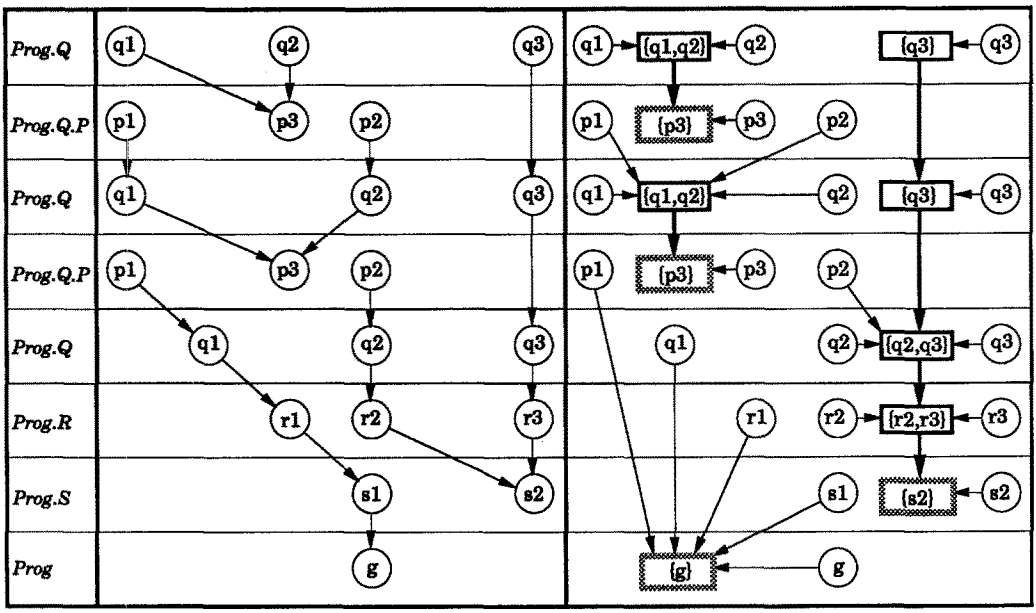

Figure 3: Stack structure in the two semantics

mation of this set. This function must be isotone and continuous, that is:

$$\forall (x,y) \in \mathcal{P}(Stack)^2 : \quad x \subseteq y \Longrightarrow \alpha(x) \sqsubseteq \alpha(y)$$
$$\forall X \subseteq \mathcal{P}(Stack) : \qquad \alpha(\bigcup X) = \bigsqcup \alpha^*(X)$$

The idea behind the abstraction of a stack structure is to merge procedure calls having the same shape. This shape will be defined later, but one of its characteristics should obviously be the partition of the identifiers accessible to that procedure into sets of identifiers sharing the same location in the stack.

All the information about the call to procedure P_k is contained in the substack $\overline{\Sigma}_k$ which is the stack built using procedure activations corresponding to the stacks indexes in $\mathcal{L}(k)$ (i.e. activations accessible to procedure P_k). Some activations are however accessible to both P_k and $P_{k'}$, namely thoses corresponding to the indexes in $\mathcal{L}(k) \cap \mathcal{L}(k')$ (which always contains 0). On the contrary, some activations of a substack $\overline{\Sigma}_k$ are hidden, that is they do not belong to any of the substacks $\overline{\Sigma}_{k+1}, \ldots, \overline{\Sigma}_n$ of the procedures called by P_k. These activations contain the values of the local variables that will have to be restored when returning from the called procedure P_{k+1}. The substack $\overline{\Sigma}_{k+1}$ thus defines the shape of the procedure call and gives the values of the identifiers, but gives no information about the return point and about the values of the local variables of its calling procedures (except of course if $P_k < P_{k+1}$). A stack abstraction could therefore be the main substack $\overline{\Sigma}_n$ plus the abstraction of the set of all the other substacks, used to modelize the "history" of the stack. But the fact that the value of some locations are duplicated in the different substacks can be a problem to insure the semantic correctness of the

analysis. That is the reason why we are going to "dissect" the stack into substacks using the sets $\mathcal{L}_n(k)$ which are the (possibly empty) sets of indexes corresponding to the hidden activations of procedure P_k.

Theorem 12 *The sets $\mathcal{L}_n(k)$ and $\mathcal{L}_n^*(k)$ are defined by:*

$$\begin{array}{rlcl} & \mathcal{L}_n(n) & = & \mathcal{L}(n) \\ & \mathcal{L}_n^*(n) & = & \mathcal{L}_n(n) \\ \forall k \in [1,n]: & \mathcal{L}_n(k-1) & = & \mathcal{L}(k-1) - \mathcal{L}_n^*(k) \\ & \mathcal{L}_n^*(k-1) & = & \mathcal{L}_n(k-1) \cup \mathcal{L}_n^*(k) \end{array}$$

The structure of the $\mathcal{L}_n(k)$ sets is essential to the semantic correctness of the abstraction. The key point is that once the activation of level k has been "hidden" by its called procedure P_{k+1} (i.e. when $\Lambda(k+1) < k$), then no further procedure can access directly this hidden activation, that is : $\forall i \in [k+1, n] : \Lambda(i) \neq k$. Moreover, one can prove the following theorem.

Theorem 13 *Let $k < n$ and $k' < n$ be such that $P_k = P_{k'}$ and $c_k = c_{k'}$ (where c_k is the control point stored in the stack at level k). The sets $\mathcal{L}(k)$ and $\mathcal{L}(k')$ can be written:*

$$\begin{array}{rcl} \mathcal{L}(k) & = & \{k = l_M > \cdots > l_0 = 0\} \\ \mathcal{L}(k') & = & \{k' = l'_M > \cdots > l'_0 = 0\} \end{array}$$

Then either $\mathcal{L}_n(k) = \mathcal{L}_n(k') = \emptyset$ or there exists an index $m \leq M$ such that:

$$\begin{array}{rcl} \mathcal{L}_n(k) & = & \{k = l_M > \cdots > l_m\} \\ \mathcal{L}_n(k') & = & \{k' = l'_M > \cdots > l'_m\} \end{array}$$

The meaning of this theorem is that whenever a given procedure calls the same procedure from the same control point ($c_k = c_{k'}$), its hidden activations are the same. For any k in $[0, n]$ we are now going to define access functions that will be used to abstract the substack $\overline{\Sigma}_k$.

Definition 14 *Let us call $Ident_k$ the set of identifiers x being accessible to P_k (i.e. $x \leq P_k$), and Loc_k the set of locations such that $\lambda \leq P_k$. Then for any identifier x, and for any location λ we define:*

$$\begin{array}{rcll} \bar{\varepsilon}_k(x) & = & (x \notin Ident_k) & \rightarrow \emptyset, \quad \varepsilon_{\Lambda(k,x^\bullet)}(x) \\ \bar{\omega}_k(\lambda) & = & (\lambda \notin Loc_k) & \rightarrow \emptyset, \quad \omega_{\Lambda(k,\lambda)}(\lambda) \\ \bar{\sigma}_k^n(\lambda) & = & (\lambda \notin Loc_k) \vee (\Lambda(k,\lambda) \notin \mathcal{L}_n(k)) & \rightarrow \perp, \quad \sigma_{\Lambda(k,\lambda)}(\lambda) \end{array}$$

The definitions of $\bar{\varepsilon}_k$ and $\bar{\omega}_k$ are not surprising. The test $\Lambda(k,\lambda) \notin \mathcal{L}_n(k)$ in the definition of $\bar{\sigma}_k^n$ means that if a location is also accessible to a called procedure, then the store should be set to $\perp$, which means that all information about the content of this location is lost. Remember that we want to dissect the stack and hence we do not want the values of the locations to be duplicated in the different substacks. This restriction will be justified later when talking about the semantic correctness of the abstract primitives.

For each index k we can now define an abstract substack as $\langle g_k, v_k^n \rangle$. The first element $g_k = \langle P_k, c_k, h_k, \bar{\varepsilon}_k, \bar{\omega}_k \rangle$ is called the *generalized control point.* It contains the control point itself

but also describes the structure of the stack that is, first, the partition of $Ident_k$ into sets of identifiers sharing the same location, and second, the locations that will have to be updated when returning from the current procedure call. This generalized control point can be seen as a characteristic of the stack used to *merge* similar procedure activations when unfolding the call graph during the analysis. This merging will be defined below using an upper closure operator.

The third element h_k of the generalized control point is called the *history* of the stack and is a *finite* abstraction of the control stack $\langle c_0, \ldots, c_{k-1} \rangle$. Different definitions can be used[5] and we can choose for instance $h_i = \langle c_{k-\theta}, \ldots, c_{k-1} \rangle$, with $\theta \geq 0$ being called the *degree* of the approximation. The parameter θ can be used to increase the precision of the analysis by forcing the duplication of procedure P_k.

At last, we have $v_k^n = \eta(\{\bar{\sigma}_k^n\})$, where η is an abstraction function from the lattice $(\mathcal{P}(Loc \rightarrow SVal_\perp), \cup)$ onto the lattice $(\overline{Store}, \vee)$. It is usual in abstract interpretation to abstract the lattice $\mathcal{P}(Loc \rightarrow SVal)$. When Loc is finite, which is the case here, this lattice is isomorphic to $\mathcal{P}((SVal)^m)$, where $m = |Loc|$, and can be abstracted by $\overline{SVal}^m$. Well-known examples of such approximations for integer variables are the constant lattice, the integer range lattice, the linear inequalities lattice (see [CH 78]), the arithmetical congruences lattice (see [Granger 89]), and the linear congruences lattice (see [Granger 90]).

Our problem here is that some elements $\bar{\sigma}_k^n$ are such that the set $\mu_k^n = Loc - (\bar{\sigma}_k^n)^{-1}(\{\perp\})$ is not equal to Loc. In this case, the abstract store v_k^n can be considered as being an element of $\overline{SVal}[\mu_k^n]$ which is isomorphic to $\overline{SVal}^m$, where $m = |\mu_k^n|$. It is then obvious that $\overline{Store}$ can be represented by the lattice $\prod_{\mu \in \mathcal{P}(Loc)} \overline{SVal}[\mu]$.

However, one can show using Theorem 13 that if $g_k = g_{k'}$ then $\mu_k^n = \mu_{k'}^n$. Consequently, if we want to merge abstract substacks having the same generalized control point (using the $\rho_\vee$ operator defined hereunder), we can use in practice $\overline{Store} = (\sum_{\mu \in \mathcal{P}(Loc)} \overline{SVal}[\mu])_\perp$.

Definition 15 *Let G be any set and $\vee$ be a join operator over a lattice V. We define the function $\rho_\vee$ over the lattice $\mathcal{P}(G \times V)$ by:*

$$\rho_\vee(\{\langle g_i, v_i \rangle\}_{i \in I}) = \{\langle g, \bigvee_{g_i = g} v_i \rangle : g \in \{g_i\}_{i \in I}\}$$

It is clear that $\rho_\vee$ is an upper closure and consequently $\rho_\vee(\mathcal{P}(G \times V))$ is known to be a lattice for the join operator $\hat{\vee}$ defined by:

$$\widehat{\bigvee_{i \in I}} S_i = \rho_\vee(\bigcup_{i \in I} S_i)$$

This lattice is also isomorphic to the lattice : $G \rightarrow V$. Therefore, if $r \in \rho_\vee(\mathcal{P}(G \times V))$, the value v such that $\langle g, v \rangle \in r$ will be noted $r(g)$.

Using this upper closure, we can now define the abstraction of a single stack using the function $\alpha_0 : Stack \rightarrow \overline{Control} \times \overline{Return}$, where $\overline{Return} = \mathcal{P}(\overline{Control} \times \overline{Store})$, defined by:

$$\alpha_0(\Sigma) = \langle g_0, \rho_\vee(\{\langle g_k, v_k^n \rangle\}_{k \in [0,n]}) \rangle$$

[5] See for instance the *call string approach* in [Sharir 81] for examples of such approximations.

The abstraction function $\alpha : \mathcal{P}(Stack) \to \overline{Stack}$ is simply defined by extending α_0 to $\mathcal{P}(Stack)$ and then normalizing using $\rho_{\hat{\vee}} : \alpha = \rho_{\hat{\vee}} \circ \alpha_0^*$. $\overline{Stack}$ is thus a subset of $\mathcal{P}(\overline{Control} \times \overline{Return})$ and is a lattice for the join operator $\sqcup = \hat{\vee}$. An element of $\overline{Stack}$ will be noted $\langle g, r \rangle$.

7 Abstract primitives

We are now going to abstract the *Call* and *Return* primitives. Let $\langle g, r \rangle$ be the current abstract stack. We use the following notations:

$$\begin{array}{rcl} g & = & \langle P, c, h, \bar{\varepsilon}, \bar{\omega} \rangle \\ r & = & \{\langle g_i, v_i \rangle\}_{i \in I} \\ v & = & r(g) \in \overline{SVal}[\mu] \end{array}$$

The abstract store $v = v[\mu]$ thus corresponds to the current generalized control point g, and μ is the set of locations accessible to procedure P. In order to deal with the locations created during procedure calls, we will use the function $\Delta^+_{\mu^-,\mu^+} : \overline{SVal}[\mu^-] \to \overline{SVal}[\mu^+]$ which takes and abstract store defined over μ^- and inserts the new locations in $\mu^+ - \mu^-$ with undefined values. The function $\Delta^{\lambda^-}_{\mu^-,\mu^+}$ does the same but assigns the value of the location λ^- to the new locations. On the opposite, the function $\Delta^-_{\mu^+,\mu^-} : \overline{SVal}[\mu^+] \to \overline{SVal}[\mu^-]$ forgets every information about the locations in $\mu^+ - \mu^-$. A formal definition is given below.

Definition 16 *For any sets $\mu^- \subseteq \mu^+$, let $\mathcal{T}(\mu^\pm)$ be an upper approximation of the lattice $\mathcal{P}(\mu^\pm \to SVal)$, $\alpha^\pm$ and $\gamma^\pm$ being the abstraction and meaning functions. The functions $\Delta^+_{\mu^-,\mu^+}$, $\Delta^-_{\mu^+,\mu^-}$, and $\Delta^{\lambda^-}_{\mu^-,\mu^+}$ are defined by:*

$$\Delta^+_{\mu^-,\mu^+}(P) = \alpha^+(\{\sigma \in (\mu^+ \to SVal) : \sigma_{|\mu^-} \in \gamma^-(P)\})$$
$$\Delta^-_{\mu^+,\mu^-}(P) = \alpha^-(\{\sigma_{|\mu^-} : \sigma \in \gamma^+(P)\})$$
$$\Delta^{\lambda^-}_{\mu^-,\mu^+}(P) = \alpha^+(\{\sigma \in (\mu^+ \to SVal) : \sigma_{|\mu^-} \in \gamma^-(P) \wedge \forall \lambda^+ \in (\mu^+ - \mu^-) : \sigma(\lambda^+) = \sigma(\lambda^-)\})$$

7.1 Procedure calls

Let us call $\langle g', r' \rangle$ the abstract stack after the procedure call. The generalized control point $g' = \langle P', c'_0, h', \bar{\varepsilon}', \bar{\omega}' \rangle$ is easily determined using the formal parameter binding function Π, and the environment $\bar{\varepsilon}$. We call μ' the set of locations accessible to P'. We have $\mu' = \mu'_S \cup \mu'_V \cup \mu'_A \cup \mu'_L$, where $\mu'_S = \mu_S$ contains the locations shared between procedure P and procedure P', namely the locations accessible to $P'^\bullet$ (for $P'^\bullet \leq P$ by construction), and μ'_V, μ'_A and μ'_L are the locations that are local to P' and respectively belong to *Value*, *Alias* and *Local*.

The second element r' is defined by:

$$r' = \rho_\vee(r[\langle g, \tilde{v} \rangle / \langle g, v \rangle] \cup \{\langle g', v' \rangle\})$$

The first abstract store $\tilde{v}$ is what remains of the abstract store v after the call to procedure P'. Remember that all the shared locations in μ_S must be "erased" to be consistent with our abstraction α. This is achieved through the use of Δ^-_{μ,μ_H}, where $\mu_H = \mu - \mu_S$ is the set of "hidden" locations of procedure P.

In order to simplify the theory, we only allow actual parameters to be identifiers, even for call-by-value formal parameters[6]. We can therefore model the call by a function π binding formal parameters locations in $\mu'_B = \mu'_S \cup \mu'_A \cup \mu'_V$ to locations in μ. This function π is one to one from μ'_A onto μ_H by construction, and is the identity over $\mu'_S = \mu_S$. We thus define $\bar{\mu}'_V$ as being any of the largest subsets of μ'_V for which π is one to one over $\bar{\mu}'_B = \mu'_S \cup \mu'_A \cup \bar{\mu}'_V$.

The value v' is the abstract store of procedure P' after the call. It is obtained from $v[\mu]$ in three steps. The first step consists in substituting in μ the aliased locations $\lambda \in \pi(\bar{\mu}'_B)$ by their alias $\pi^{-1}(\lambda)$ and then by erasing the locations in μ that are not aliased by using $\Delta^-_{\mu,\pi(\bar{\mu}'_B)}$.

The second step consists in the insertion of the locations in $\mu'_V - \bar{\mu}'_V = \mu'_B - \bar{\mu}'_B$. For every location $\lambda \in \bar{\mu}'_B$, let us call δ the set of locations in $\mu'_V - \bar{\mu}'_V$ bound to $\pi(\lambda)$ by π. Their insertion is achieved by using $\Delta^{\lambda}_{\bar{\mu}'_B,\bar{\mu}'_B \cup \delta}$.

At last, the third step consists in the insertion of the locations in μ'_L by using $\Delta^+_{\mu'_B,\mu'}$.

The elements of r' corresponding to generalized control points different from g and g' remain unchanged. This can be justified by the following theorem which shows that only the substack $\overline{\Sigma}_n$ is altered during a procedure call.

Theorem 17 *For any index k in $[0, n-1]$: $\mathcal{L}_{n+1}(k) = \mathcal{L}_n(k)$*

7.2 Returning from procedure calls

We are now going to return from the call to procedure P'. First of all, we must notice that there might be several abstract substacks in r' leading to g' when called, for the history of these substacks has been abstracted. Let us choose one of them and call it $\langle g, v \rangle$. The abstract stack after returning will be:

$$\langle g^+, r'[\langle g^+, v^+ \rangle / \langle g', v' \rangle] \rangle$$

where $\langle g^+, v^+ \rangle$ is the new current abstract substack, and $g^+ = g[c^+/c]$, c^+ being the control point following c in the source code. The new abstract store $v^+ = v_1 \wedge v'_3$ is defined as follows.

The abstract store $v_1 = \Delta^+_{\mu - \pi(\mu'_A),\mu} \circ \Delta^-_{\mu,\mu-\pi(\mu'_A)}(v)$ is simply the store that was pushed on the stack, from which every information about the aliased locations in $\pi(\mu'_A)$ has been removed. Only the values of the "hidden locations" remain.

The abstract store v'_3 represents the values of the aliased locations after returning from the called procedure P'. It is obtained from v' in three steps. First, $v'_1 = \Delta^-_{\mu',\mu'_S \cup \mu'_A}(v')$ has lost every information about the call-by-value formal parameters and local variables of P'. Then by substituting the aliases $\lambda' \in \mu'_A$ by their aliased location $\pi(\lambda')$ in P, we get v'_2. And at last we get v'_3 by inserting the missing "hidden" locations, with undefined values, using $\Delta^+_{\mu_S \cup \pi(\mu'_A),\mu}$.

8 Implementation

Based on the previous theoretical work a system called SYNTOX has been developed. It consists of 450 Kbytes of *C* source code and has an optional user-friendly interface running under X/Windows. Its purpose is to analyze the value range of integer variables in Pascal programs.

[6]This transformation is always possible using temporary variables.

Any program can be analyzed, provided that it does not use the '@' operator (*address of*), and that it does not use procedurals parameters. Jumps to local labels are supported, as well as *long jumps* — that is jumps to labels outside the current procedure — although this feature has not been covered in the present paper.

The system performs forward and backward analyses until it reaches a fixpoint. Each analysis performs a widening and a narrowing pass, which is necessary when using a very large lattice such as the integer range lattice. We didn't say however how the widening operation is achieved when talking about the abstraction method. This is a problem though, for the resolution strategy used in SYNTOX is not to solve a system of semantic equations, but rather to dynamically generate new equations. An algorithm has been developed, but it is beyond the scope of the present paper and will be exposed later.

Assertions such as {% x in [0,10] %} can be inserted into the Pascal source code to impose that x be in the range $[0, 10]$ at the corresponding control point. This assertion is taken into account during the analysis. The forward analysis provides information on how an assertion propagates to the end of the program, whereas the backward analysis determines *necessary* — but not sufficient — conditions to be satisfied at the beginning of the program to insure that this assertion is satisfied at run time. As a consequence, the backward analysis determines necessary conditions for the program to terminate (provided of course that the state at the end of the program is not equal to $\perp$). Examples are given below.

The results given by SYNTOX could be used by compilers to eliminate most of the run time tests on array bounds. Such an approach has been taken by a company such as *Alsys* which, by using an *intraprocedural* analysis, has been able to eliminate almost 80 per cent of the run time tests generated by its *Ada* compiler.

9 Examples

MacCarthy's 91 function provides an good example of the power of backward analysis. If the assertion {% x <= 101 %} is inserted in the Pascal source code at control point {1}, SYNTOX will prove that $x = 91$ at control point {2}. But on the contrary, if the assertion {% x = 91 %} is inserted at control point {2}, SYNTOX will "back propagate" the condition that $x \in [lo, 101]$ at control point {1}. This kind of reasonning about programs properties can be very helpful for program debugging. The analysis time for this example was roughly 4.2 seconds user time on a Sun 3/60. Two forward analyses and one backward analysis were done before reaching a fixpoint, each analysis including a widening pass and a narrowing pass.

Another example, freely adapted from [Banning 79] is given in figure 4. For each procedure, we have given every possible partition of the set of identifiers into sets of identifiers sharing the same location. For each pseudo-location, the right arrow indicates the location being aliased. It can be seen that even a very simple program can generate very complicated sets of aliases. The analysis time for this example was roughly 10 seconds.

```
program Aliases;
   var   x, y : integer;
   procedure P1(var  x1, y1 : integer);
   begin
      if x1 < y1 then      x1 := 5
      else                 P1(y1, x1)
   end;
   procedure P2(var  x2 : integer);
      var     y2, z2 : integer;
      procedure P3(var  x3 : integer);
         var      y3 : integer;
      begin
         if z2 = y then    P2(x3)
         else              P2(y);
         y3 := y2
      end;
   begin
      if y2 = 5 then       P3(y)
      else if y2 > 5 then  P3(x2);
      P1(x2, x); y2 := x2
   end;
   procedure P0;
      var     x0 : integer;
   begin
      P2(x); P2(x0); P1(x, y);
   end;
begin
   P0
end.
```

{x}	{y}	{x0}

Procedure P0

{x, y1}	{y}	{x1} →	{y1}
{x, x1}	{y}	{y1} →	{x1}
{x, y1}	{y}	{x1} →	{x2}
{x, x1, y1}			{y}
{x, x1}			{y, y1}
{x, y1}			{y, x1}

Procedure P1

{x}	{y}	{x2} →	{x2}	{y2}	{z2}
{x}	{y}	{x2} →	{x0}	{y2}	{z2}
{x}	{y, x2}		{y2}		{z2}
{x, x2}	{y}		{y2}		{z2}

Procedure P2

{x}	{y}	{x2, x3} →	{x2}	{y2}	{z2}	{y3}
{x}	{y, x3}	{x2} →	{x2}	{y2}	{z2}	{y3}
{x}	{y}	{x2, x3} →	{x0}	{y2}	{z2}	{y3}
{x}	{y, x3}	{x2} →	{x0}	{y2}	{z2}	{y3}
{x, x2, x3}		{y}	{y2}	{z2}		{y3}
{x}	{y, x2, x3}		{y2}	{z2}		{y3}
{x, x2}	{y, x3}		{y2}	{z2}		{y3}

Procedure P3

Figure 4: Partitionning into alias sets

10 Conclusion

Several attempts have been made to provide a framework for the abstract interpretation of languages with procedures and recursivity. Patrick Cousot in [Cousot 78] studied the case of recursive procedures, but with a language that used the call-by-value-result parameter passing scheme, and that did not have nested procedures. Other people have developed either "ad-hoc algorithms" or abstract interpretations to find aliases of variables in Pascal programs ([Banning 79]), or in higher order functional languages with recursivity and call-by-reference, but without nested procedures ([Demers 87]). They did not propose however a framework for the abstract interpretation of their languages. The reason why they could not do so is that, as we saw, the main problem to be solved is not to determine the set of aliases of a variable, but to determine all the possible partitions of the variables lexically visible from a procedure into variables sharing the same location.

We proposed in this article a framework for designing easily customizable abstract interpretations of Pascal-like languages. This customization is a great improvement over the preceding works, for there is no need to provide a soundness theorem for any new abstract interpretation. The abstraction method we proposed is highly "tunable", as emphasized in [Jones 82],

and can be designed to get an abstract interpretation of the desired precision. Moreover, the operational approach we have taken should make the implementation of semantic analyzers for Pascal reasonably easy.

A natural extention to the present work is to study the full abstract interpretation of the Pascal language, with procedural parameters. Theoretical results ([Clarke 77, Clarke 84]) show that it should be somewhat more difficult.

References

[Abramsky 87] Samson Abramsky and Chris Hankin : "Abstract Interpretation of Declarative Languages", Ellis Horwood Books in Computing Science (1987)

[ASU 86] Aho, Sethi, Ullman : "Compilers — Principles, Techniques and Tools", Addison-Wesley Publishing Company (1986)

[Banning 79] John P. Banning : "An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables" in *Proc. of the 6th ACM Symp. on POPL* (1979)

[Clarke 77] E.M. Clarke "Programming languages constructs for which it is impossible to obtain good Hoare axiom systems" in *Proc. of the 4th ACM Symp. on POPL*, pp 129-147 (1977)

[Clarke 84] E.M. Clarke "The characterization problem for Hoare logic" in *Phil. Trans. R. Soc. Lond.*, pp 423-440

[Cousot 77] Patrick and Radhia Cousot : "Abstract Interpretation : a unified lattice model for static analysis of programs by construction of approximative fixpoints" in *Proc. of the 4th ACM Symp. on POPL* (1977)

[Cousot 78] Patrick and Radhia Cousot : "Static determination of dynamic properties of recursive procedures" in *Formal Description of Programming Concepts*, North Holland Publishing Company, pp 237-277 (1978)

[Cousot 81] Patrick Cousot : "Semantic foundations of program analysis" in Muchnick and Jones Eds., *Program Flow Analysis, Theory and Applications*, Prentice-Hall (1981)

[CH 78] Patrick Cousot and Nicolas Halbwachs : "Automatic discovery of linear constraints among variables of a program", in *Proc. of the 5th ACM Symp. on POPL*, pp 84-97 (1978)

[Demers 87] Alan J. Demers, Anne Neirynck, Prakash Panangaden : "Computation of Aliases and Support Sets" in *Proc. of the 14th ACM Symp. on POPL* (1987)

[Granger 89] Philippe Granger : "Static analysis of arithmetical congruences", in *International Journal of Computer Mathematics*, pp 165-190 (1989)

[Granger 90] Philippe Granger : "Static analysis of linear congruences among variables of a program", L.I.X. Internal Report (to appear)

[Jones 82] Neil Jones and Steven Muchnick : "A Flexible Approach to Interprocedural Data Flow Analysis and Programs with Recursive Data Structures", in *Proc. of the 9th ACM Symp. on POPL* (1982)

[Plotkin 81] Gordon D. Plotkin : "A structural approach to operational semantics" Aarhus University Internal Report, Denmark, (September 1981)

[Sharir 81] Micha Sharir, Amir Pnueli : "Two Approaches to Interprocedural Data Flow Analysis" in Muchnick and Jones Eds., *Program Flow Analysis, Theory and Applications*, Prentice-Hall (1981)

[Venkatesh 89] G.A. Venkatesh : "A Framework for Construction and Evaluation of High-level Specifications for Program Analysis Techniques", in *Proc. of SIGPLAN '89 Conference on Programming Language Design and Implementation* (1989)