programmation

Proceedings
of the 2nd international symposium on Programming
edited by B. Robinet
Paris
April, 13 - 15 1976

Actes du 2e colloque international sur la programmation
direction B. Robinet
Paris
13 - 15 avril 1976

DUNOD
phase recherche informatique

STATIC DETERMINATION OF DYNAMIC PROPERTIES OF PROGRAMS

Patrick COUSOT* and Radhia COUSOT**

Université Scientifique et Médicale de Grenoble

1 - INTRODUCTION -

In high level languages, compile time type verifications are usualy incomplete, and dynamic coherence checks must be inserted in object code. For example, in PASCAL one must dynamically verify that the values assigned to subrange type variables, or index expressions lie between two bounds, or that pointers are not nil, ... We present here a general algorithm allowing most of these certifications to be done at compile time. The static analysis of programs we do consists of an abstract evaluation of these programs, similar to those used by NAUR for verifying the type of expressions in ALGOL 60 [6] , by SINTZOFF for verifying that a module corresponds to its logical specification [9] , by KILDALL for global program optimization [5] , by WEGBREIT for extracting properties of programs [9] , by KARR for finding affine relationships among variables of a program [4] , by SCHWARTZ for automatic data structure choice in SETL [8] ,...

The essential idea is that, when doing abstract evaluation of a program, "abstract" values are associated with variables instead of the "concrete" valees used while actually executing. The basic operations of the language are interpreted accordingly and the abstract interpretation then consists in a transitive closure mechanism. One may consider abstract values belonging to no finite sets, but the properties of the transitive closure algorithm are chosen such that the abstract interpretation stabilizes after finitely many steps,

---

* Attaché de Recherche au CNRS, Laboratoire Associé N°7.
** This work was supported by IRIA-SESORI under grant 75-035.

which implies that it can be fully worked out at compile time. The elementary interpretation of the basic operations of the language and the choice of abstract values depend upon the specific dynamic properties which one wants to extract from the program. Provided that this choice of abstract values and basic operations satisfies the general framework we specify in this paper, correctness and termination of the abstract evaluation algorithm are guaranteed [2]. For simplicity purposes we illustrate the method, by the determination of range information associated with integer variables in high level languages such as PASCAL [10] or LIS [3].

2 - ABSTRACT VALUES -

The abstract evaluation of a program is a "symbolic" interpretation of this program, using abstract values instead of concrete or execution values. An abstract value denotes a set of concrete values, (defined in extension) or properties of such a set (intensive definition), satisfying a number of dynamic conditions. Let V_c be the set of concrete values and V_a the set of abstract values.

In most examples given here, V_c will be the set of integers and V_a the set of intervals of integers. If $V_c = \bar{Z}$ is the set of integers (between the limits $-\infty$ and $+\infty$) used in a programming language, the intervals of integers will be denoted $[a, b]$ where $a \in \bar{Z}$, $b \in \bar{Z}$ and $a \leq b$.

The correspondance between a set of concrete values and an abstract value, is established by the "abstraction function" @ :

$$2^{V_c} \xrightarrow{@} V_a$$

(this notation states that @(S) must be defined for any S of 2^{V_c})

Example :

$S \subseteq \bar{Z}$, $\quad @(S) = \left[\underset{x \in S}{\mathrm{MIN}}\,(x), \underset{y \in S}{\mathrm{MAX}}(y)\right]$

$@(\{-1,5,3\}) = [-1,5]$

$@(\{3,4,5,\ldots\}) = [3,+\infty]$

Another function, γ, gives the concrete form of an abstract value :

$$V_a \xrightarrow{\gamma} 2^{V_c}$$

Example :

$$\gamma([a, b]) = \{x \mid (x \in \bar{Z}) \wedge (a \leq x \leq b)\}$$

The functions @ and γ are defined such that they verify :

$$(\forall s \in 2^{V_c}, s \subseteq \gamma(@(s)))$$

and $$(\forall v \in V_a , v = @(\gamma(v))).$$

Corresponding to the union $\cup$ of sets of concrete values, the union $\bar{\cup}$ of abstract values must also be defined for every particular abstract evaluation :

$$V_a \times V_a \xrightarrow{\bar{\cup}} V_a$$

Example :

$$[a_1, b_1] \bar{\cup} [a_2, b_2] = [MIN(a_1, a_2), MAX(b_1, b_2)]$$

The abstraction function @ is assumed to be a morphism from $(2^{V_c}, \cup)$ into $(V_a, \bar{\cup})$:

$$\forall(s_1, s_2) \subseteq V_c^2, \quad @(s_1 \cup s_2) = @(s_1) \bar{\cup} @(s_2)$$

This implies that $\bar{\cup}$ has the associativity, commutativity and idempotency properties, and that the zero element $\square$ of $\bar{\cup}$ is also $@(\emptyset)$ where $\emptyset$ is the empty set. $\square$ is called the null abstract value.

Corresponding to the inclusion $\subseteq$ of sets of concrete values, the abstract evaluation uses the inclusion $\bar{\leq}$ of abstract values, which is defined by :

$\forall(v_1, v_2) \in V_a^2,$

$$\{v_1 \bar{\leq} v_2\} \Leftrightarrow \{v_1 \bar{\cup} v_2 = v_2\}$$

and $$\{v_1 \bar{<} v_2\} \Leftrightarrow \{(v_1 \bar{\leq} v_2) \wedge (v_1 \neq v_2)\}$$

From this definition and the hypothesis on $\bar{\cup}$, $\bar{\leq}$ can be showned to be a partial ordering, and $\square$ is included in every abstract value.

Example :

$$[a_1, b_1] \bar{\leq} [a_2, b_2] \iff \{(a_2 \leq a_1) \wedge (b_2 \geq b_1)\}$$

$[-2, 10] \bar{\leq} [-3, 12]$ but $[-2, 10]$ and $[0, 12]$ are not comparable.

V_a is a complete $\bar{\cup}$-semi-lattice under the partial ordering $\bar{\leq}$.

Example :

The strictly increasing infinite chain $\square$, $[1, 1]$, $[1, 2]$, $[1, 3]$, ... has an upper bound which is $[1, +\infty]$.

Finally, for the abstract evaluation of loops, the problem arises of computing the limit of strictly increasing infinite chains, in a finite number of steps. For that purpose, an operation has been defined, called widening, denoted $\bar{\nabla}$:

$$V_a \times V_a \xrightarrow{\bar{\nabla}} V_a$$

For every particular abstract evaluation, $\bar{\nabla}$ must be defined such that :

- $\forall (v_1, v_2) \in V_a^2, \{(v_1 \bar{\cup} v_2) \bar{\leq} (v_1 \bar{\nabla} v_2)\}$ and

- every infinite sequence $s_o, s_1, \ldots, s_n, \ldots$ of the form $s_o = \square$, $s_1 = s_o \bar{\nabla} v_1, \ldots, s_n = s_{n-1} \bar{\nabla} v_n, \ldots$, (where $v_1, v_2, \ldots, v_n, \ldots$ are arbitrary abstract values), is not strictly increasing.

Example :

$[a_1, b_1] \bar{\nabla} [a_2, b_2] =$

[<u>if</u> $a_2 < a_1$ <u>then</u> $-\infty$ <u>else</u> a_1 <u>fi</u>,

<u>if</u> $b_2 > b_1$ <u>then</u> $+\infty$ <u>else</u> b_1 <u>fi</u>]

$$\square \,\bar{\nabla}\, [1, 10] = [1, 10]$$

$$[1, 10] \,\bar{\nabla}\, [1, 11] = [1, +\infty]$$

$$[1, +\infty] \,\bar{\nabla}\, [0, 12] = [-\infty, +\infty]$$

so that, in that case, the length of the sequences $s_o, s_1, \ldots, s_n, \ldots$ which are strictly increasing is less or equal to 4.

3 - ABSTRACT CONTEXTS -

The abstract evaluation of a program computes by successive approximations an abstract context at every program point. An abstract context is a set of pairs (i, v) which expresses that the identifier i has the abstract value v at some program point. Then, in every actual execution of the program, the objects accessed by i will be in the set $\gamma(v)$ at that program point.

If I denotes the set of identifiers (after the syntactical conflicts of identifiers in the program have been resolved), the set $\mathfrak{C}$ of abstract contexts is such that :

$$\mathfrak{C} \subset 2^{I \times (V_a - \{\square\})}$$

and the pairs in a given context differ from one another in their identifiers :

$\{\forall C \in \mathfrak{C}, \forall (i, j) \in I^2, \forall (v, u) \in (V_a - \{\square\})^2,$

$\{(i, v) \in C \wedge (j, u) \in C \wedge (i, v) \neq (j, u)\} \Rightarrow \{i \neq j\}\}.$

We note C(i) the value of an identifier i in a context C, it is defined by

$C(i) =$ <u>if</u> $(\exists v \in (V_a - \{\square\}) \mid (i, v) \in C)$ <u>then</u> v <u>else</u> $\square$ <u>fi</u>.

<u>Example</u> :

$C = \{(x, [1, 10]), (y, [-\infty, 0])\}$;

$C(x) = [1, 10]$; $C(z) = \square$;

In particular, we note Φ the null abstract context so that, from the above definition : $\{\forall i \in I, \Phi(i) = \square\}$.

The union $C \bar{\bar{\cup}} C'$ of two contexts C and C', will be used for expressing, for example, the context resulting from conditionnal statements. The widening $C \bar{\bar{\nabla}} C'$ of contexts, will be used in loops. They are defined using the union and widening of abstract values :

$$C_1 \bar{\bar{\cup}} C_2 = \{(i, v) \mid (i \in I) \wedge (v \in (V_a - \{\square\})) \wedge (v = C_1(i) \bar{\cup} C_2(i))\}$$

$$C_1 \bar{\bar{\nabla}} C_2 = \{(i, v) \mid (i \in I) \wedge (v \in (V_a - \{\square\})) \wedge (v = C_1(i) \bar{\nabla} C_2(i))\}$$

We can show, for every identifier i, that :

$$(C_1 \bar{\bar{\cup}} C_2)\ (i) = C_1(i) \bar{\cup} C_2(i)$$

and $(C_1 \bar{\bar{\nabla}} C_2)\ (i) = C_1(i) \bar{\nabla} C_2(i)$

Example :

$$C_1 = \{(x, [1, 10]), (y, [-\infty, 0])\} ;$$

$$C_2 = \{(x, [0, 5]), (z, [1, +\infty])\} ;$$

$$C_1 \bar{\bar{\cup}} C_2 = C_2 \bar{\bar{\cup}} C_1 = \{(x, [0, 10]), (y, [-\infty, 0]), (z, [1, +\infty])\}$$

$$C_1 \bar{\bar{\nabla}} C_2 = \{(x, [-\infty, 10]), (y, [-\infty, 0]), (z, [1, +\infty])\}$$

$$C_2 \bar{\bar{\nabla}} C_1 = \{(x, [0, +\infty]), (y, [-\infty, 0]), (z, [1, +\infty])\}$$

As before, we define the inclusion $\bar{\bar{\leq}}$ of contexts by

$$\{C_1 \bar{\bar{\leq}} C_2\} \Leftrightarrow \{C_1 \bar{\bar{\cup}} C_2 = C_2\}$$

and $\{C_1 \bar{\bar{<}} C_2\} \Leftrightarrow \{(C_1 \bar{\bar{\leq}} C_2) \wedge (C_1 \neq C_2)\}$

it can be shown that this is equivalent to

$$\{ C_1 \bar{\bar{\leq}} C_2 \} \Leftrightarrow \{ \forall i \in I, C_1(i) \bar{\leq} C_2(i) \}$$

$\mathcal{C}$ is a complete $\bar{\bar{\cup}}$-semi-lattice under the partial ordering $\bar{\bar{\leq}}$.
In addition, for every contexts C_1, C_2 we have :

$$C_1 \bar{\bar{\cup}} C_2 \bar{\bar{\leq}} C_1 \bar{\bar{\nabla}} C_2$$

and there are no infinite strictly increasing chains $S_0 \bar{\bar{<}} S_1 \bar{\bar{<}} \dots \bar{\bar{<}} S_n \bar{\bar{<}} \dots$ of abstract contexts of the form $S_0 = \Phi$, $S_1 = S_0 \bar{\bar{\nabla}} C_1, \dots, S_n = S_{n-1} \bar{\bar{\nabla}} C_n, \dots$ for arbitrary abstract contexts $C_1, \dots, C_n, \dots$

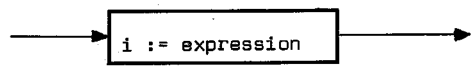
4 - PROGRAMS -

As a first approximation of programs, we will use finite flowcharts. They are built from the following elementary program units :

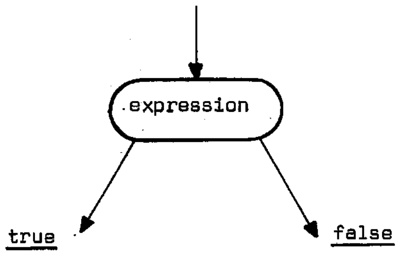
A single entry node :

exit nodes :

assignment nodes :

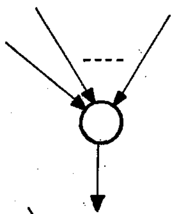


test-nodes :

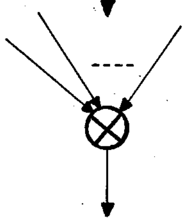


(To avoid the choice of a particular programming language, we will assume that the evaluation of expressions in assignment and test nodes have no side-effect).

simple junction nodes :

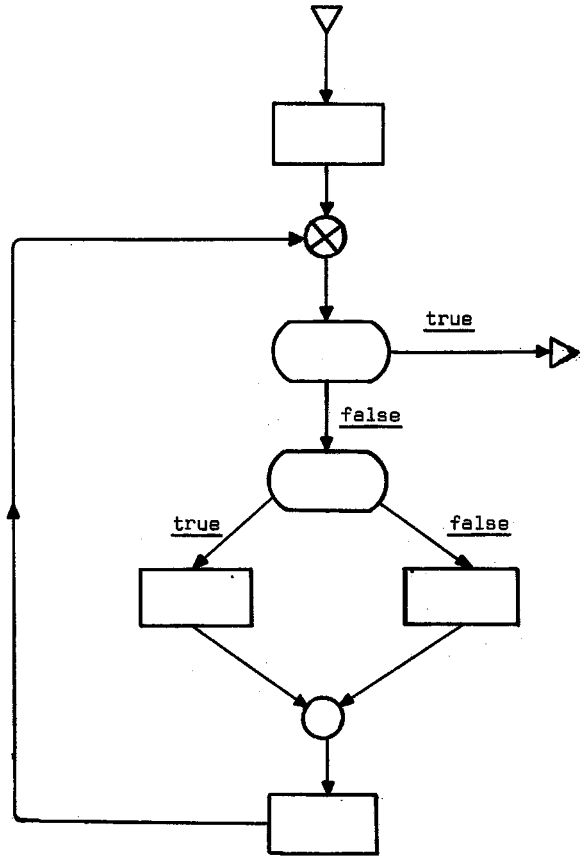


loop junction nodes :



Only connected flowcharts are considered and there is at least one path from the unique entry node to every node of the flowchart. With these conditions, every cycle in the flowchart contains at least one simple or loop junction node. Additionally, a preliminary graph theoretic analysis of the flowchart has been performed, choosing which of the junction nodes are loop junction nodes, so that every cycle contains at least one loop junction node, and that the total number of loop junction nodes is minimal.

Example :



5 - ELEMENTARY ABSTRACT INTERPRETATION OF BASIC PROGRAM UNITS -

For every particular application of the abstract evaluation algorithm, we must provide a definition of the evaluation of basic program units. The function $\underline{\mathfrak{I}}$ defines for any assignment or test program unit n and input context C, an output context $\underline{\mathfrak{I}}(n, C)$, (or two when n is a test node). The application $\underline{\mathfrak{I}}$ must be a correct "abstraction" of the actual execution of program unit n. It may be defined as follows :

5.1. - $\forall n \in N_a$ (the set of assignment nodes), n is of the form :

$$\xrightarrow{\quad n \quad} \boxed{v := f(v_1, \dots, v_m)} \longrightarrow$$

where $(v, v_1, \dots, v_m) \in I^{m+1}$ and $f(v_1, \dots, v_m)$ is an expression of the language depending on the variables $v_1, \dots, v_m$. Then :

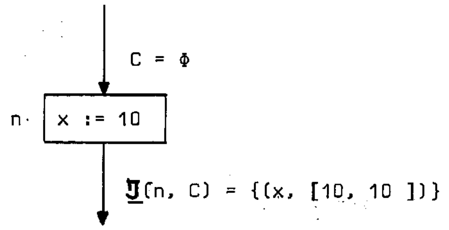
$\{\forall C \in \mathcal{C}, \forall i \in I, i \neq v \Rightarrow \underline{\mathfrak{I}}(n, C)(i) = C(i)\}$ and

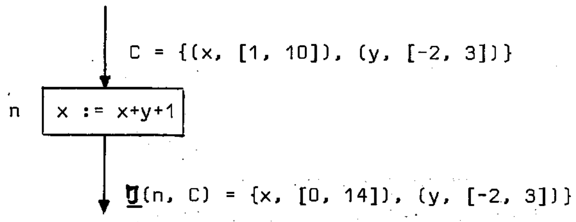
$\{\forall C \in \mathcal{C}, \underline{\mathfrak{I}}(n, C)(v) = @[\{f(\nu_1, \dots, \nu_m) \mid (\nu_1, \dots, \nu_m) \in \gamma(C(v_1)) \times \dots \times \gamma(C(v_m))\}]\}$

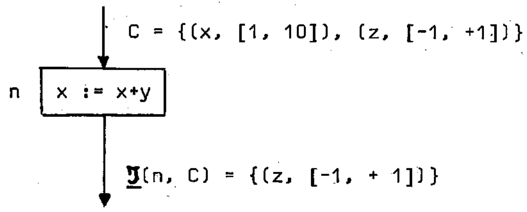
The first condition expresses that the evaluation of the expression $f(v_1, \dots, v_m)$ has no side effect, and the second one that the value of v in the output context, is the abstraction of the set of values of the expression $f(v_1, \dots, v_m)$ when the values $(\nu_1, \dots, \nu_m)$ of $(v_1, \dots, v_m)$ are chosen in the input context C.

(In order to avoid language dependent problems, we make the general assumption that all functions and predicates are "naturally extended" in such a way that they are undefined whenever at least one of their arguments is undefined (i.e. is $\square$)).

Examples :



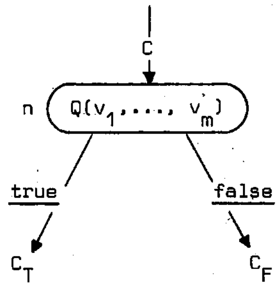




We have C(y) = □, so that the expression x + y is undefined, therefore $\mathfrak{I}$(n, C)(x) = □.
In the case of that specific application, an interval arithmetic [7] is used for defining $\mathfrak{I}$(n, C).

5.2. - The elementary abstract interpretation $\mathfrak{I}$(n, C) of a test node n, in input context C results in two output contexts C_T and C_F associated with the true and false edges respectively :

$\forall$ n $\in$ N_T(the set of test nodes), n is of the form :



where $Q(v_1,\ldots, v_m)$ is a boolean expression without side-effect depending on the variables $v_1,\ldots, v_m$. Then we define $\mathfrak{I}(n, C) = (C_T, C_V)$ such that :

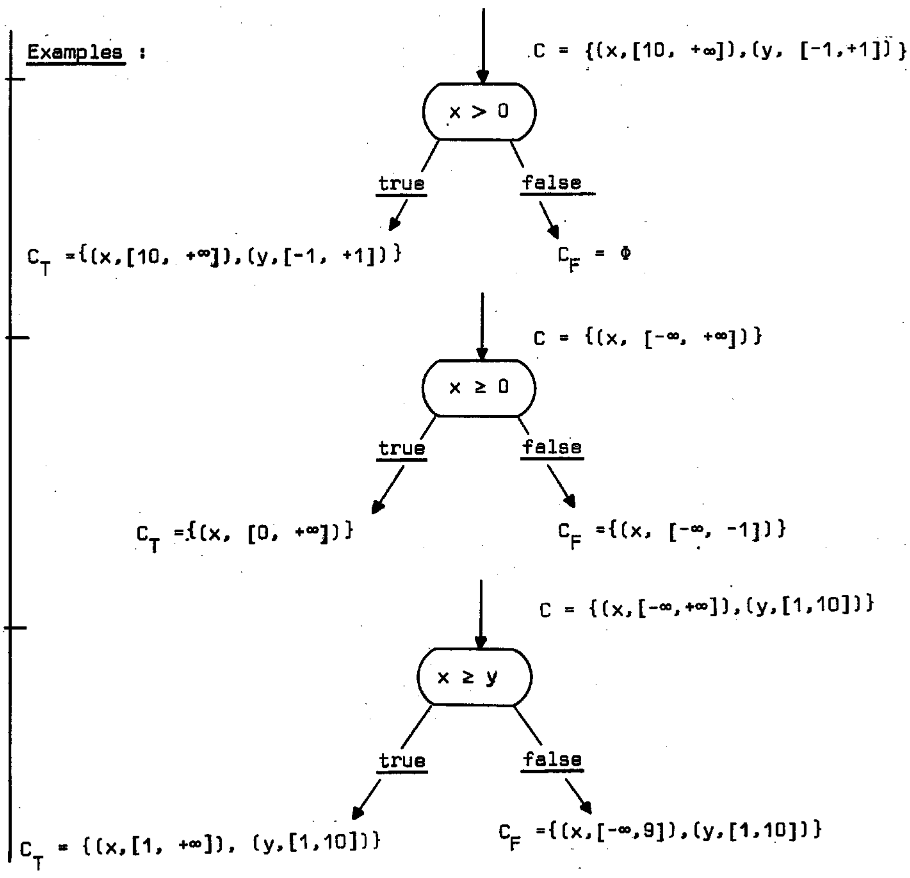
$\forall\, i \in I$,

$$C_T(i) = @(\{\iota \mid (\iota \in \gamma(C(i))) \wedge (\exists\, (v_1,\ldots,v_m) \in \gamma(C(v_1)) \times \ldots \times \gamma(C(v_m)) \mid Q(v_1,\ldots,v_m))\})$$

$$C_F(i) = @\{\iota \mid (\iota \in \gamma(C(i))) \wedge (\exists\, (v_1,\ldots,v_m) \in \gamma(C(v_1)) \times \ldots \times \gamma(C(v_m)) \mid \neg Q(v_1,\ldots,v_m))\})$$

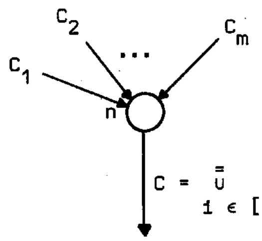
On the true edge for example, the abstract value of a variable i is the abstraction of the set of values ι chosen in the input context C, for which the evaluation of the predicate Q in context C may yield the value "true".

Examples :



In the case of that specific application, the treatment of conditionnal statements has to designate whether a given variable belongs to a certain interval on the real line, our approach is similar to that of [1].

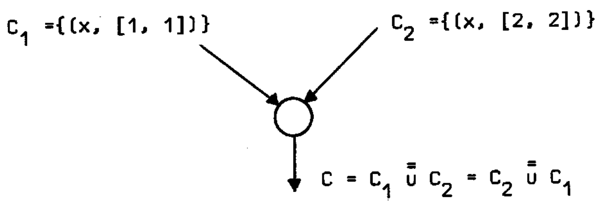
5.3. - When the abstract interpretor follows an execution path until reaching a test node, this may give rise to two execution paths. Each of the two paths will be executed pseudo-parallely, until reaching an exit node, in which case the execution of that path ends, or a junction node, in which case the pseudo-parallel execution paths are synchronized. In order to compute the output context of a junction node, we must have first computed the input contexts of the input edges which may be reached by an execution path. The unreachable input edges have their associated contexts initialized to the null context Φ. For a simple junction node n, we have :



$$C = \overset{=}{\cup}_{i \in [1, m]} C_i$$

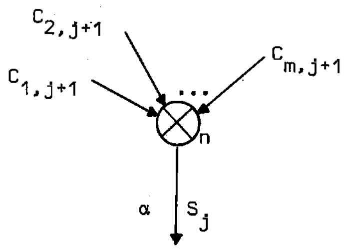
(the use of this generalized notation results from the commutativity and associativity of $\overset{=}{\cup}$).

<u>Example</u> :



$$C = C_1 \overset{=}{\cup} C_2 = C_2 \overset{=}{\cup} C_1$$
$$=\{(x, [1, 1] \bar{\cup} [2, 2])\}$$
$$=\{(x, [1, 2])\}$$

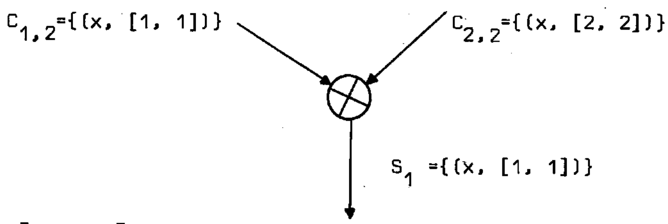
5.4. - In order that the abstract interpretation terminate correctly, we need something analogous to the induction step used in the automatic verification of programs with loops. This is provided at the loop junction nodes by the widening of contexts, as follows :



If the $j^{\underline{th}}$ pass on a loop junction node n has associated the context S_j to the output arc α of that node (or S_0 has been initialized to the null context), then the context associated to α on the $j+1^{\underline{th}}$ pass, will be :

$$S_{j+1} = S_j \;\bar{\bar{\nabla}}\Big(\bar{\bar{\cup}}_{i\,\in[1,m]}\; C_{i,j+1}\Big)$$

<u>Example</u> :



$$\begin{aligned}
S_2 &= S_1 \;\bar{\bar{\nabla}}\; (C_{1,2} \;\bar{\bar{\cup}}\; C_{2,2}) \\
&= S_1 \;\bar{\bar{\nabla}}\; \{(x, [1, 1] \;\bar{\cup}\; [2, 2])\} \\
&= \{(x, [1, 1])\} \;\bar{\bar{\nabla}}\; \{(x, [1, 2])\} \\
&= \{(x, [1, 1] \;\bar{\nabla}\; [1, 2])\} \\
&= \{(x, [1, +\infty])\}
\end{aligned}$$

Note that the widening at the loop junction nodes introduces a loss of information. However it will be shown on examples that the tests behave as filters. Furthermore, for a PASCAL like language, one can first use the bounds given in the declaration of x, before widening to "infinite" limits.

6 - ABSTRACT INTERPRETOR -

The abstract interpretor starts with the empty context Φ on all arcs. For each of the different types of nodes, we have described a transformation which specifies the context(s) for the output arc(s) of the node, in term of the context(s) associated to input arc(s) to the node, and where relevant, the contents of the node. The algorithm essentially performs applications of these transformations until all contexts are stabilized , i.e. the application of a transformation at any node results in no change in the contexts of its output arcs. The distinct execution paths are followed pseudo-parallely, with synchronization on junction nodes.

During abstract evaluation, it should be noted that it is useless to go on along one path when the output context C' of a node is included in the context C already associated with the arc out of that node. This results from the fact that the elementary interpretation $\mathfrak{I}$ is an increasing function for $\overset{=}{\leq}$ in $\mathfrak{C}$.
It can be shown that :

$$\{C' \overset{=}{\leq} C\} \Rightarrow \{\forall\ n,\ \underline{\mathfrak{I}}(n,C') \overset{=}{\leq} \underline{\mathfrak{I}}(n,\ C)\}$$

The proof of termination of the abstract evaluation comes from the fact that on one hand the sequence of contexts associated with the output arc of each loop junction node form a strictly ascending chain which cannot be infinite and, on the other hand, that every loop contains a loop junction node.

The general abstract interpretor is now stated :

(As shown in [2] , it is convenient to mark the edges of the graph, in order to cause the abstract interpretation to consider each arc of the program graph at least once. For simplicity, this has not been done here).

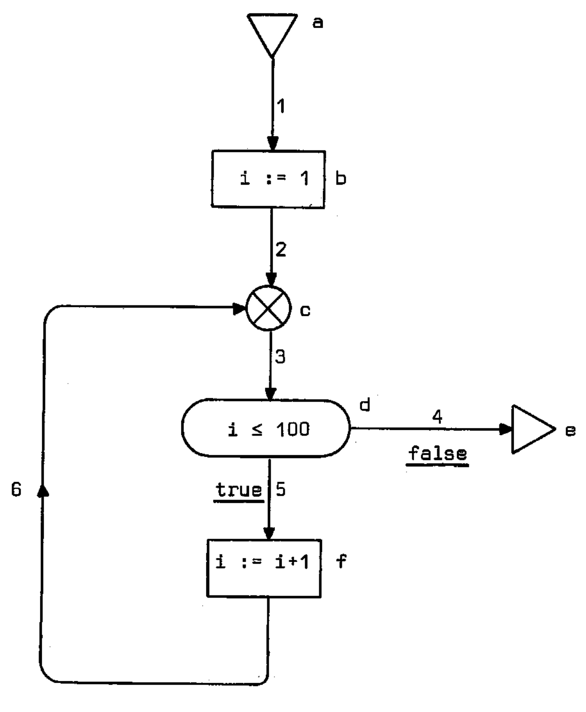
```
procedure abstract interpretation (graph) ;
begin
|    for each arc of graph do local context (arc) := Φ repeat ;
|    execution paths := {exit-arc (entry-node (graph))} ; junctions := ∅ ;
|    while (execution paths ≠ ∅) do
|         while (execution paths ≠ ∅) do
|         |    input arc := choose (execution paths) ;
|         |    execution paths := execution paths - {input arc} ;
|         |    node := final-end (input arc) ;
|         |    case node of
|         |    ├─  assignment node→
|         |    |   assign output context (exit-arc (node),
|         |    |                  ℑ(node, local context (input arc))) ;
|         |    ├─  test node→
|         |    |       (C_T, C_F) := ℑ(node, local context (input arc)) ;
|         |    |   assign output context (true-exit-arc (node), C_T) ;
|         |    |   assign output context (false-exit-arc (node), C_F) ;
|         |    ├─  simple or loop junction node→  junctions := junctions ∪
|         |    |                                        {node} ;
|         |    ├─  exit node → ;
|         |    end ;
|         repeat ;
|         for each junction node of junctions do
|         |    output context : = ∪̄ local context (input arc);
|         |                       input arc ∈ entry-arcs(junction node)
|         |    if ¬(output context ≦ local context (exit-arc(junction node)))
|         |    |   then
|         |    |   case junction node of
|         |    |   ├─  simple junction node →
|         |    |   |   assign output context (exit-arc (junction node),
|         |    |   |                          output context) ;
|         |    |   ├─  loop junction node →
|         |    |   |   assign output context (exit-arc (junction node),
|         |    |   |   local context (exit-arc (junction node)) ∇̄
|         |    |   |   output context) ;
|         |    |   end ;
|         |    fi ;
|         repeat ;
|         junctions := ∅ ;
|    repeat ;
return ;
|    procedure assign output context (output arc, output context) ;
|         if ¬ (output context ≦ local context (output arc)) then
|         |    local context (output arc) := output context ;
|         |    execution paths := execution paths ∪ {output arc} ;
|         fi ;
end ;
```

In [2] we have shown that the algorithm terminates, even when analyzing non-terminating programs, and that it is correct : if C' is the final abstract context associated with an arc α, then for every identifer i of the program, and every actual execution of that program :

- if C(i) = □, then i is never initialized on arc α, or α is on a dead path.
- if C(i) is some abstract value v, then the concrete value of i on arc α within every execution path p containing α belongs to $\gamma(v)$, under the condition that i has been correctly initialized on path p.

Example :

Our first example is very simple in order to illustrate the technique :



The following table shows the analysis of this program graph, with the specific abstract interpretation we have chosen as example in the paper :

step	node	input arc	local context (input arc)	output context	execution paths	junctions
1	a			Φ	{1}	$\emptyset$
2	b	1	Φ	{i,[1,1]}	{2}	$\emptyset$
3	c	2	{i, [1,1]}		{ }	{c}
4	c	2	{i, [1,1]}			
				{i, [1,1]}	{3}	$\emptyset$
		6	Φ			
5	d	3	{i, [1,1]}	{i, [1,1]}	{5}	$\emptyset$
6	f	5	{i, [1,1]}	{i, [2,2]}	{6}	$\emptyset$
7	c	6	{i, [2,2]}		{ }	{c}
8	c	2	{i, [1,1]}	{i, [1,1] $\bar{\nabla}$[1,2]}		
		6	{i, [2,2]}	={i, [1,$+\infty$]}	{3}	$\emptyset$
9	d	3	{i, [1,$+\infty$]}	{i,[101,$+\infty$]}	{4,	$\emptyset$
				{i, [1,100]}	5}	
10	e	4	{i, [101,$+\infty$]}		{5}	$\emptyset$
11	f	5	{i, [1,100]}	{i, [2,101]}	{6}	$\emptyset$
12	c	6	{i, [2,101]}		{ }	{c }
13	c	2	{i, [1,1]}	{i, [1,$+\infty$]$\bar{\nabla}$ ([1,1] $\bar{\cup}$ [2,101])}		
		6	{2, [2,101]}	$\leqq$ {i, [1,$+\infty$]}.end.		

After processing the flowchart, the final context on each arc is listed in the table opposite the circled nodes. Note that the results are approximate,which is a consequence of the undecidability of the problem of finding exact domains for the variables at each program point.

7 - TWO STRATEGIES FOR ABSTRACT INTERPRETATION -

The first strategy we have presented until now, consists in associating with every edge of the graph an increasing chain of contexts, starting from a first approximation which is the null context.

One can proceed in a reverse way, by associating with every edge of the graph, a decreasing chain of contexts, starting from the universal context.

The abstract evaluation will then use the partial ordering $\bar{\geq}$, which is defined by :

$$\forall\ (v_1,v_2) \in V_a^2,\ \{v_1 \bar{\geq} v_2\} \Leftrightarrow \{v_1 \bar{\cup} v_2 = v_1\} \Leftrightarrow \{v_2 \bar{\leq} v_1\}$$

The universal abstract value $\# = @(V_c)$, is used to define the universal context ψ, such that :

$$\{\forall\ i \in I,\ \psi(i) = \#\}$$

Instead of widening, we use the "narrowing" of abstract values denoted $\bar{\Delta}$, which must be defined such that :

- $V_a \times V_a \xrightarrow{\bar{\Delta}} V_a$
- $\forall\ (v_1,v_2) \in V_a^2,\ \{v_1 \bar{\geq} v_2\} \Rightarrow \{v_1 \bar{\geq} v_1\ \bar{\Delta}\ v_2 \bar{\geq} v_2\}$
- every infinite sequence $s_o, s_1, \ldots, s_n, \ldots$ of the form

  $s_o = \#,\ s_1 = s_o\ \bar{\Delta}\ v_1, \ldots,\ s_n = s_{n-1}\ \bar{\Delta}\ v_n, \ldots$

(where $v_1, v_2, \ldots, v_n, \ldots$ are arbitrary abstract values) is not strictly decreasing.

<u>Example</u> :

$$\# = @(V_c) = [-\infty,+\infty]$$

$$[a_1,b_1]\ \bar{\Delta}\ [a_2,b_2] =$$

$$[\underline{\text{if}}\ a_1 = -\infty\ \underline{\text{then}}\ a_2\ \underline{\text{else}}\ \text{MIN}\ (a_1,a_2),$$

$$\underline{\text{if}}\ b_1 = +\infty\ \underline{\text{then}}\ b_2\ \underline{\text{else}}\ \text{MAX}\ (b_1,b_2)]$$

$$[-\infty,+\infty]\ \bar{\Delta}\ [-\infty,101] = [-\infty,101]$$

$$[-\infty,101]\ \bar{\Delta}\ [0,100] = [0,101]$$

$$[0,100]\ \bar{\Delta}\ [0,99] = [0,100]$$

For this new strategy, the abstract interpretor is obtained from the previous one, with $\bar{\psi}$, $\bar{\bar{\geq}}$, $\bar{\bar{\Delta}}$ instead of Φ, $\bar{\bar{\leq}}$ and $\bar{\bar{\nabla}}$ respectively. (In this case too, one must ensure that the abstract interpretor considers every edge of the program graph at least once).

Example :

The following table shows the analysis of the program graph given at paragraph 6 , with this new strategy :

step	node	input arc	local context (input arc)	output context	execution paths	junctions
1	a			ψ	{1}	∅
2	b	1	ψ	{i, [1,1]}	{2}	∅
3	c	2	{i, [1,1]}		∅	{c}
4	c	2	{i, [1,1]}			
				ψ	{3}	∅
		6	ψ			
5	d	3	ψ	{i, [101,+∞]}	{4,	∅
				{i, [-∞,100]}	5}	
6	e	4	{i, [101,+∞]}		{5}	∅
7	f	5	{i, [-∞,100]}	{i, [-∞,101]}	{6}	∅
8	c	6	{i, [-∞,101]}		∅	{c}
9	c	2	{i, [1,1]}			
				{i, [-∞,+∞] $\bar{\Delta}$ [-∞,101]}		
		6	{i, [-∞,101]}	={i, [-∞,101]}	{3}	∅
10	d	3	{i, [-∞,101]}	{i, [101,101]}	{4}	
				{i, [-∞,100]} = {i, [-∞,100]}. end.		
11	e	4	{i, [101,101]}		∅	∅

Generally as noticed in the above example, the two strategies don't come out with the same result. If the first strategy leads to the abstract value v_1 for identifer i, and the second one gives v_2, the final result may be choosen as $@(\gamma(v_1) \cap \gamma(v_2))$, which is a better result than those obtained separately. However a best solution, is to start with the strategy which associates with every edge of the graph an increasing chain of contexts and to use next the resulting contexts for initialization when applying the other strategy.

8 - EXAMPLES -

The final result obtained for our first example is :

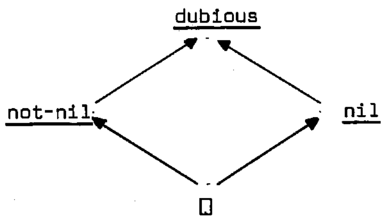
```
{i,□}
 i := 1 ;
{i, [1,1]}
 L : {i, [1,101]}
 if i ≤ 100 then
      {i,[1,100]}
       i := i+1 ;
      {i,[2,101]}
       go to L ;
 else
    {i,[101,101]}
     stop ;
 fi ;
```

The next example is the binary search of a given key K in a table R of 100 elements whose keys are in increasing order. The result of the program analysis is the following :

```
lwb := 1 ; upb := 100 ;
{(lwb, [1,1]), (upb, [100, 100])}
L : {(lwb, [1,101]), (upb, [0,100]), (m, [1,100])}
if upb < lwb then
     {(lwb, [1,101]), (upb, [0,100]), (m, [1,100])}
       unsuccessfull search ;
fi ;
{(lwb, [1,100]), (upb, [1,100]), (m, [1,100])}
m := (upb + lwb) ÷ 2 ;
{(lwb, [1,100]), (upb, [1,100]), (m, [1,100])}
if K = R(m) then
       successfull search ;
elsif K < R(m) then
       upb := m - 1 ;
      {(lwb, [1,100]), (upb, [0,99]), (m, [1,100])}
else
       lwb := m + 1 ;
       {(lwb, [2, 101]), (upb, [1,100]), (m, [1,100])}
fi ;
{(lwb, [1,101]), (upb, [0,100]), (m, [1,100])}
go to L ;
```

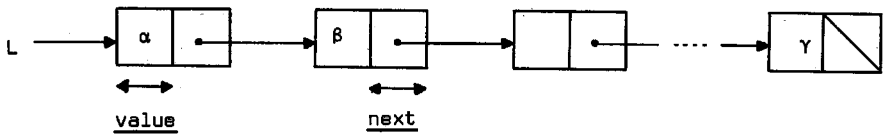
In PASCAL or LIS like languages, where lwb, upb and m have been declared of type 1 ..101, 0 .. 100 and 1 .. 100, dynamic tests for assignments to these variables or bounds tests for access to array R are statically shown to be useless.

The last example is dedicated to detection of incorrect access to records through nil pointers. There are four abstract values ☐, nil, not-nil, dubious with the following ordering :



In the case of an abstract $\bar{\cup}$-semi-lattice satisfying the maximal chain condition (every strictly increasing chain is finite), the widening $\bar{\nabla}$ is taken to be $\bar{\cup}$. This is the case in that example because there is a finite set of abstract values.

The problem consists in finding the Kth value of a linear linked list L :



The intended solution, with its analysis is the following :

```
        {(K, [-∞,+∞]), (L, dubious)}
        if K ≤ 0 then stop fi ;
        cursor := L ;
     E :{(K, [1,+∞]), (cursor, dubious), (L, dubious)}
        if K ≠ 1 then
            {(K, [2,+∞]), (cursor, dubious),...}
            K := K - 1 ;
            {(K, [1,+∞],(cursor, dubious),...}
            if cursor = nil then
                stop ;
            else
                {(K,[1,+∞]),(cursor, not-nil),...}
[α]             cursor := next (cursor);
                {..., (cursor, dubious),...}
            fi ;
        {(K, [1,+∞]), (cursor, dubious), (L, dubious)}
        go to E ;
        fi ;
        {(K, [1,1]), (cursor, dubious), (L, dubious)};
[β] ... value (cursor)...
```

It is shown, at line [α] that "cursor" in "next(cursor)", is not a nil pointer, and it has been taken account of the fact that the function next delivers a nil or not-nil pointer. On the other hand, "cursor" might be nil at line [β], and from this diagnostic information, the programmer should be able to discover that he has forgotten the case of a list of length K - 1.

9 - "DUAL" ABSTRACT INTERPRETATIONS -

The abstract interpretations we have presented until now determine for every program point a "maximal" property, common to all possible runs of that program. A "minimal" property may be obtained by two dual algorithms using the intersection in place of union in the two former abstract interpretations.

-10 - CONCLUSION -

The abstract interpretation algorithm briefly reported here may be extended to other control structures. This has be done for subprograms, including recursive ones, and we are developping the case of semi-coroutines, coroutines, backtracking, etc.

More general assertions about programs may be determined at compile time with the above method of program analysis, by choosing abstract values which are more elaborated than those given in this paper. For example, abstract values for integers may be chosen as intervals, lists of disjoined intervals, intervals with symbolic bounds, lists of disjoined symbolic intervals ...

It is important to note that, because of the undecidable problems we are faced with, our results are valid but fundamentally not perfect. However it should be clear that this incompleteness is acceptable for the verification of correct uses of data and operations, for the supply of diagnostic informations, for program optimization, for choosing types or organizations of data structures in very high level languages, etc.

More generally, types are used in high-level languages for specifying some logical properties of data and their memory representation. As presented in this paper the abstract value concept, allows us to specify and statically check additionnal dynamic properties. Under the light of this work, we are now extending the classical type approach, in order to catch the dynamic aspects obtained by abstract interpretation, and thus provide a better data specification framework in programming languages.

11 - BIBLIOGRAPHY -

[1] BLEDSOE, W.W. - BOYER, R.S.
"Computer proofs of limit theorems"
Proceedings of the I.J.C.A.I., 586-600 (1971)

[2] COUSOT, P. - COUSOT, R.
"Vérification statique de la cohérence dynamique des programmes"
Laboratoire d'Informatique, U.S.M.G., Grenoble.
Research report of contract IRIA-SESORI 75-035, (Sept. 1975)

[3] ICHBIAH, J.D. - RISSEN , J.P. - HELIARD, J.C. - COUSOT, P.
"The system implementation language LIS"
CII - TR 4549 E/EN - (Dec.1974)

[4] KARR, M.
"On affine relationships among variables of a program"
Massachusetts computer associates, inc.
CA - 7402 - 2811, (1974)

[5] KILDALL, G.A.
"A unifed approach to global program optimization"
Conf. Record of ACM symposium on principles of programming languages,
Boston, 194-206 (1973)

[6] NAUR, P.
"Checking of operand types in ALGOL compilers"
BIT 5, 151÷163 (1965)

[7] RIS, F.N.
"Tools for the analysis of interval arithmetic"
RC 5305, IBM T.J. Watson Research Center , (1975).

[8] SCHWARTZ, J.T.
"Automatic data structure choice in a language of very high level"
Second ACM Symposium on Principles of Programming Languages,
Palo Alto, California, 36-40 (Jan. 1975)

[9] SINTZOFF, M.
"Vérifications d'assertions pour les fonctions utilisables comme valeurs et affectant des variables extérieures"
proving and improving programs, Ed. G. HUET, G. KAHN,
IRIA, 11-27 (1975)

[10] WEIGBREIT , B.
"Property extraction in well-founded property sets"
IEEE transactions on software engineering,
Vol SE-1, NO.3, 270-285 (Sept.1975)

[11] WIRTH , N.
"The programming language PASCAL"
Acta Informatica 1, 35-63 (1971)

=:=:=:=:=:=:=

Authors' Address :

P.COUSOT, R.COUSOT
Laboratoire d'Informatique (D.319)
Boîte postale 53
38 041 GRENOBLE CEDEX
FRANCE.