UNIVERSITE SCIENTIFIQUE ET MEDICALE
et INSTITUT NATIONAL POLYTECHNIQUE
de GRENOBLE

MATHEMATIQUES APPLIQUEES ET INFORMATIQUE

Laboratoire associé au CNRS n°7

B.P. 53 - 38041 GRENOBLE cédex France

STATIC VERIFICATION OF DYNAMIC TYPE PROPERTIES OF VARIABLES

Patrick COUSOT and Radhia COUSOT

RR. 25 Novembre 1975

RAPPORT DE RECHERCHE

STATIC VERIFICATION OF DYNAMIC TYPE PROPERTIES OF VARIABLES

-:-:-

Patrick COUSOT and Radhia COUSOT

ABSTRACT : For 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 <u>nil</u>, etc...
We present a general algorithm allowing most of these certifications to be done at compile time. The static analysis of a program consists of an abstract evaluation of the program (NAUR, SINTZOFF, KILDALL, WEGBREIT, KARR) which computes, by successive approximations, an abstract context at every program point. A context is a set of pairs (identifier, abstract value). An abstract value denotes a set of execution values or properties of it, satisfying a number of dynamic conditions. An abstract interpretor defines the sequencing of abstract evaluation through the different paths of the program, and builds a union of contexts resulting from the jonction of these paths. The elementary interpretation of basic operations of the language and the choice of abstract values are left unspecified. They depend upon the specific dynamic properties of variables which have been chosen for being extracted from the program. The correctness and termination of the abstract evaluation algorithm, rely only on the algebraic structure of the abstract values set, and on some properties of abstract basic operations. Several abstract evaluations can then be defined for a program, in order to eliminate redundant tests, verify correct uses of operations, choose types or organization of data structures in the case of very high level languages or provide diagnostic information.

Authors' Address : Laboratoire d'Informatique (D. 319)
UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE
Boîte Postale n° 53
38041 GRENOBLE-Cédex - France -

STATIC VERIFICATION OF DYNAMIC TYPE PROPERTIES OF VARIABLES

-:-:-:-

Patrick COUSOT* and Radhia COUSOT**

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 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 [8], 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], etc. We present an abstract interpretation of programs, which permits the verification of most dynamic type properties at compile time. We illustrate the technique by the determination of range informations for integers 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.

---

* Research Assistant - CNRS

** Contract IRIA-SESORI 75-035

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" @ :

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

Example :

$S \subset \bar{Z}$, $\quad @(S) = \left[\begin{array}{ll} \text{MIN}\,(x), & \text{MAX}(y) \\ x \in S & y \in S \end{array}\right]$

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

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 :

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

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

$\forall (s_1, s_2) \subset V_c^2$:

$$@(s_1 \cup s_2) = @(s_1) \;\bar{\cup}\; @(s_2)$$

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 undefined abstract value.

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

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

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

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

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 :

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

Finally, for the abstract evaluation of loops, the problem arises of terminating the computation of abstract values. For that purpose, an operation has been defined, called widening, and noted $\bar{\nabla}$:

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

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>]

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. $(\neg\{s_o \bar{<} s_1 \bar{<} \ldots \bar{<} s_n \bar{<} \ldots\})$.

Example :

$$\square \ \bar{\nabla}\ [1, 10] = [1, 10]$$
$$[1, 10]\ \bar{\nabla}\ [1, 11] = [1, +\infty]$$
$$[1, +\infty]\ \bar{\nabla}\ [0, 12] = [-\infty, +\infty]$$

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 $\mathcal{C}$ of abstract contexts is such that :

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

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

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

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

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

$C(i) \equiv \underline{\text{if}}\ (\exists v \in (V_a-\{\square\})\ |\ (i, v) \in C)\ \underline{\text{then}}\ v\ \underline{\text{else}}\ \square\ \underline{\text{fi}}.$

Example :

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

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

In particular, we note Φ the empty abstract context.

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 :

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

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

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

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

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

it can be shown that this is equivalent to

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

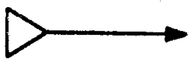
From the algebraic structure $(V_a, \bar{\cup}, \bar{\nabla}, \bar{\leq})$ defined on abstract values, we can show that the same algebraic structure holds for $(\mathcal{C}, \bar{\bar{\cup}}, \bar{\bar{\nabla}}, \bar{\bar{\leq}})$. In particular, $\bar{\bar{\cup}}$ is associative, commutative, idempotent and Φ is its zero element, $\bar{\bar{\leq}}$ is a partial ordering on $\mathcal{C}$, $C_1 \bar{\bar{\cup}} C_2 \bar{\bar{\leq}} C_1 \bar{\bar{\nabla}} C_2$ for every (C_1, C_2), and there are no infinite strictly increasing sequence $S_o \bar{\bar{<}} S_1 \bar{\bar{<}} \ldots$ $\ldots \bar{\bar{<}} S_n \bar{\bar{<}} \ldots$ of abstract contexts of the form

$S_o = \Phi, S_1 = S_o \bar{\bar{\nabla}} C_1, \ldots, S_n = S_{n-1} \bar{\bar{\nabla}} C_n, \ldots$ for arbitrary abstract contexts $C_1, \ldots, C_n, \ldots$.

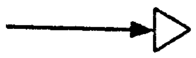
4 - PROGRAMS -

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

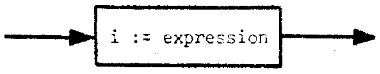
A single entry node :



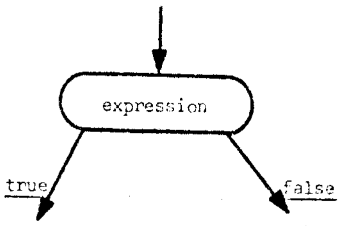
exit nodes :



assignment nodes :

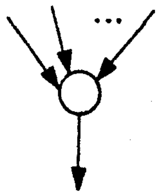


test-nodes :

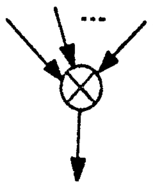


We will assume that the evaluation of expressions in assignment and test nodes have no side-effect.

simple junction nodes :

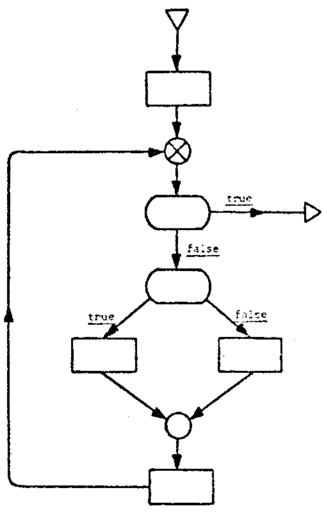


loop junction nodes :



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.

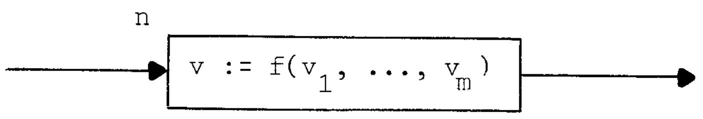
Example :



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 fonction $\underline{\mathfrak{J}}$ defines for any assignment or test program unit n and input context C, an output context $\underline{\mathfrak{J}}(n, C)$, (or two when n is a test node). The application $\underline{\mathfrak{J}}$ 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 :



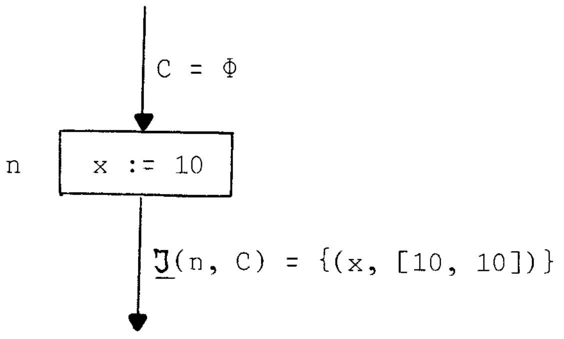
where $(v, v_1, \ldots, v_m) \in I^{m+1}$ and $f(v_1, \ldots, v_m)$ is an expression of the language depending on the variables $v_1, \ldots, v_m$. Then :

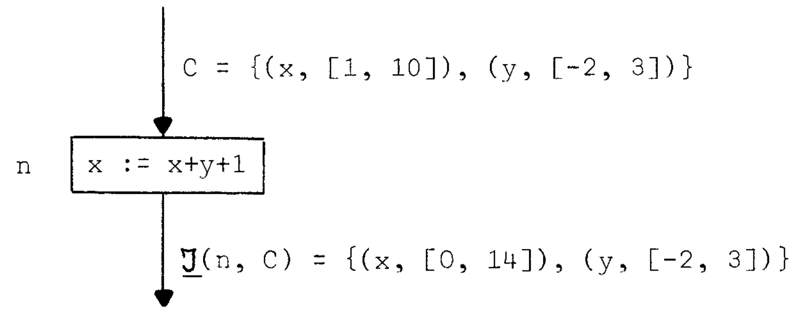
$\{\forall C \in \mathcal{C}, \forall i \in I, i \neq v \Rightarrow \underline{\mathfrak{J}}(n, C)(i) = C(i)\}$ and

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

The first condition expresses that the evaluation of the expression $f(v_1, \ldots, v_m)$ 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 $f(v_1, \ldots, v_m)$ when the values $(\nu_1, \ldots, \nu_m)$ of $(v_1, \ldots, v_m)$ are chosen in the input context C.

Examples :




C = {(x, [1, 10]), (y, [-2, 3])}
n
x := x+y+1
$\underline{\mathfrak{J}}$(n, C) = {(x, [0, 14]), (y, [-2, 3])}

In the case of that specific application an interval arithmetic [7] is used for defining $\underline{\mathfrak{J}}(n, C)$:

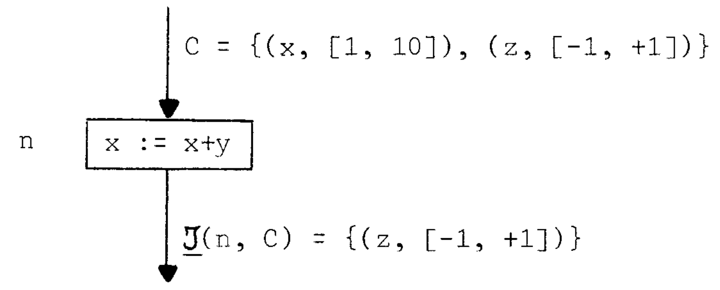
$C(x) = [1, 10]$

$C(y) = [-2, 3]$

$C(x+y) = [1+(-2), 10 + 3] = [-1, 13]$

$C(1) = [1, 1]$

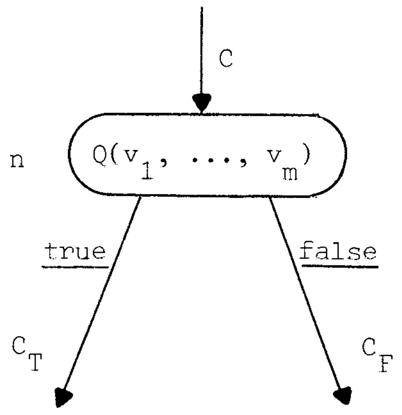
$C[x+y+1] = [-1, 13] + [1, 1] = [0, 14]$


C = {(x, [1, 10]), (z, [-1, +1])}
n
x := x+y
$\underline{\mathfrak{J}}$(n, C) = {(z, [-1, +1])}

We have $C(y) = \Box$, so that the expression $x + y$ is undefined, therefore $\underline{\mathfrak{J}}(n, C)(x) = \Box$.

5.2 - The elementary abstract interpretation $\underline{\mathfrak{J}}(n, C)$ 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 :



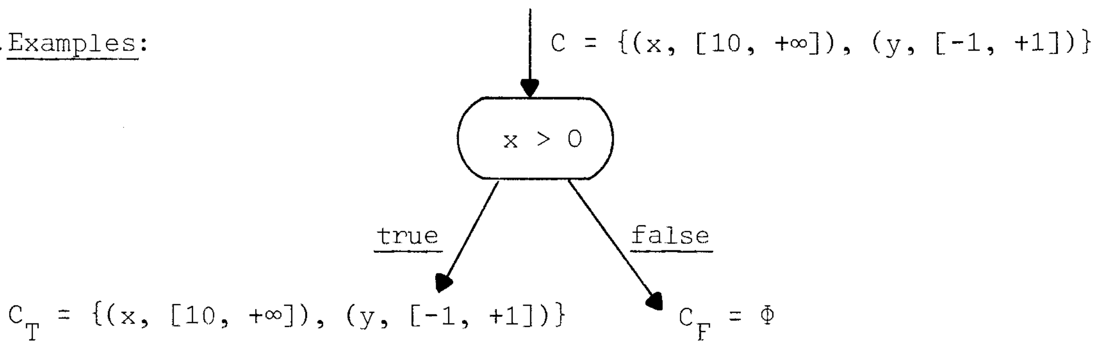
where $Q(v_1, \ldots, v_m)$ is a boolean expression without side-effect depending on the variables $v_1, \ldots, v_m$. Then we define $\underline{\mathfrak{J}}(n, C) = (C_T, C_V)$ such that, $\forall i \in I$

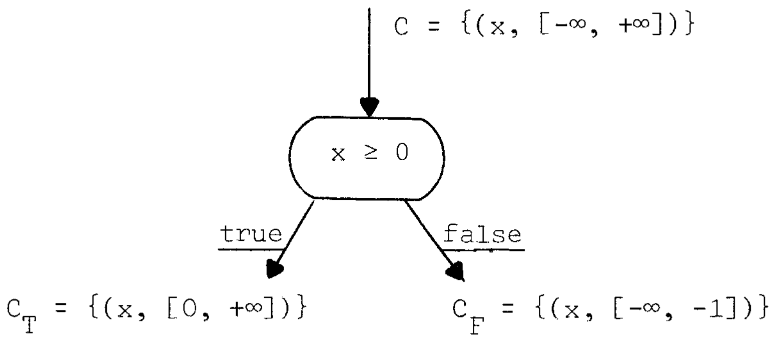
$$C_T(i) = @(\{\iota \mid (\iota \in \gamma(C(i))) \wedge (\exists(\nu_1, \ldots, \nu_m) \in \gamma(C(v_1)) \times \ldots \times \gamma(C(v_m)) \mid Q(\nu_1, \ldots, \nu_m))\})$$

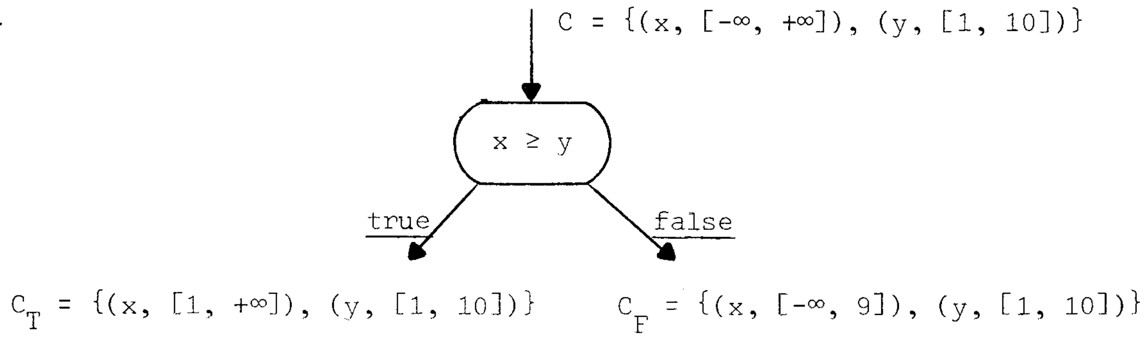
$$C_F(i) = @(\{\iota \mid (\iota \in \gamma(C(i))) \wedge (\exists(\nu_1, \ldots, \nu_m) \in \gamma(C(v_1)) \times \ldots \times \gamma(C(v_m)) \mid \neg Q(\nu_1, \ldots, \nu_m))\})$$

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".

<u>Examples</u>:


$C = \{(x, [10, +\infty]), (y, [-1, +1])\}$
$x > 0$
true
false
$C_T = \{(x, [10, +\infty]), (y, [-1, +1])\}$
$C_F = \Phi$

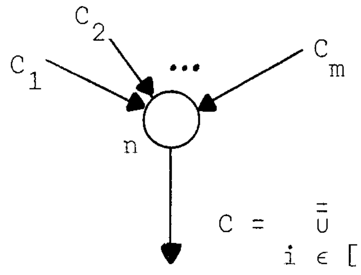

$C = \{(x, [-\infty, +\infty])\}$
$x \geq 0$
true
false
$C_T = \{(x, [0, +\infty])\}$
$C_F = \{(x, [-\infty, -1])\}$


$C = \{(x, [-\infty, +\infty]), (y, [1, 10])\}$
$x \geq y$
true
false
$C_T = \{(x, [1, +\infty]), (y, [1, 10])\}$
$C_F = \{(x, [-\infty, 9]), (y, [1, 10])\}$

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 ot 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 empty context Φ.

For a node $n \in N_{sj}$ (the set of simple junction nodes), we have :

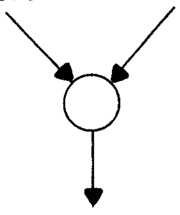


$$C = \underset{i \in [1,\, m]}{\overset{=}{\cup}} C_i$$

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

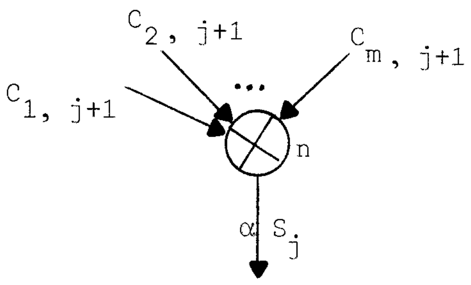
Example :

$C_1 = \{(x, [1, 1])\}$ $\qquad$ $C_2 = \{(x, [2, 2])\}$



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

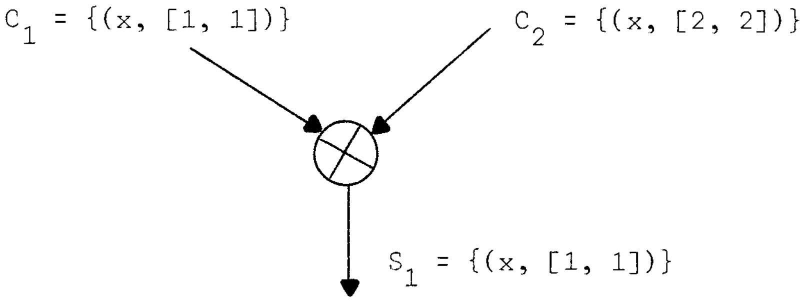
5.4 - In order that the abstract interpretation terminate correctly, we need something analogous to the induction step used in the automatic verification of programs with loops. This is provided at the loop junction nodes by the widening of contexts, as follows :



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

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

<u>Example</u> :



$$\begin{aligned} S_2 &= S_1 \;\bar{\bar{\nabla}}\; (C_1 \;\bar{\bar{\cup}}\; C_2) \\ &= S_1 \;\bar{\bar{\nabla}}\; \{(x, [1, 1] \;\bar{\cup}\; [2, 2])\} \\ &= \{(x, [1, 1])\} \;\bar{\bar{\nabla}}\; \{(x, [1, 2])\} \\ &= \{(x, [1, 1] \;\bar{\nabla}\; [1, 2])\} \\ &= \{(x, [1, +\infty\ ])\} \end{aligned}$$

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 - <u>ABSTRACT INTERPRETOR</u> -

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 $\underline{\mathfrak{J}}$ is an increasing function for $\overset{=}{<}$ in $\mathfrak{C}$. It can be shown that :

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

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 :

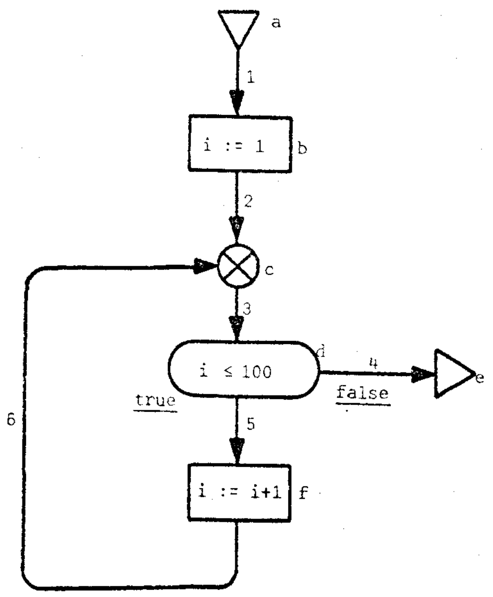
```
procedure abstract interpretation (graph) ;
begin
    for each arc of graph do local context (arc) := $\Phi$ repeat ;
    execution paths := {entry-arc (entry-node (graph))} ; junctions := $\emptyset$ ;
    while (execution paths $\neq \emptyset$) do
        while (execution paths $\neq \emptyset$) do
            {choose an execution path}
            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), $\mathfrak{J}$(node,
                                              local context (input arc))) ;

                test node →
                    ($C_T$, $C_F$) := $\mathfrak{J}$(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 $\cup$ {node} ;
                exit node → ;
            end ;
        repeat ;
        for each junction node of junctions do
            output context := $\bar{\bar{\cup}}$ local context (input arc)
                              input arc $\in$ entry-arc (junction node)

            if $\neg$(output context $\bar{\bar{\leq}}$ 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)) $\bar{\bar{\nabla}}$
                    output context) ;
                end ;
            fi ;
        repeat ;
        junctions := $\emptyset$ ;
    repeat ;
    return ;
    procedure assign output context (output arc, output context) ;
        if $\neg$(output context $\bar{\bar{\leq}}$ local context (output arc)) then
            local context(output arc) := output context ;
            execution paths := execution paths $\cup$ {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 identifier i of the program, and every actual execution of that program :

- if C(i) = $\square$, 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.

7 - EXAMPLES -

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				{1}	∅
2	b	①	∅	{i, [1, 1]}	{2}	∅
3	c	2	{i, [1, 1]}		{ }	{c}
4	c	2	{i, [1, 1]}			
				{i, [1, 1]}	{3}	∅
		6	∅			
5	d	3	{i, [1, 1]}	{i, [1, 1]}	{5}	∅
6	f	5	{i, [1, 1]}	{i, [2, 2]}	{6}	∅
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, +∞]}	{3}	∅
9	d	③	{i, [1, +∞]}	{i, [101, +∞]}	{4	∅
					,	
				{i, [1, 100]}	5}	
10	e	④	{i, [101 +∞]}		{5}	∅
11	f	⑤	{i; [1, 100]}	{i, [2, 101]}	{6}	∅
12	c	6	{i, [2, 101]}		{ }	{c}
13	c	②	{i, [1, 1]}	{i, [1, +∞] $\bar{\nabla}$ ([1, 1] $\bar{\cup}$ [2, 101])}		
		⑥	{2, [2, 101]}	$\bar{\leq}$ {i, [1, +∞]}. 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.

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, +∞]), (upb, [-∞, 100]), (m, [-∞, +∞])}

if upb < lwb then
    {(lwb, [1, +∞]), (upb, [-∞, 100]), (m, [-∞, +∞])}
      unsuccessfull search ;
fi ;

{(lwb, [1, 100]), (upb, [1, 100]), (m, [-∞, +∞])}
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 ;
```

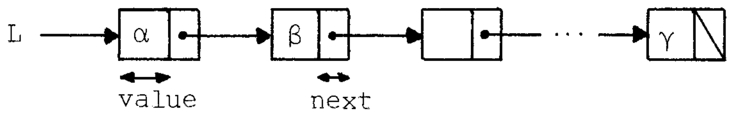
In PASCAL or LIS like languages, where lwb, upb and m should have been declared of type 1 .. 101, 0 .. 100 and 1 .. 100, dynamic tests for assignments to these variables or bounds tests for access to array R are statically shown to be useless.

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

$$\begin{tikzcd}
 & \text{dubious} & \\
\text{not-nil} \arrow[ur] & & \text{nil} \arrow[ul] \\
 & \square \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

In the case of a finite number of abstract values, the widening $\bar{\nabla}$ is taken to be $\bar{\cup}$.

The problem consists in finding the K^{th} 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 the empty list, and that of a list of length K-1.

8 - CONCLUSION -

Several analysis can be defined for a program, using the general abstract interpretation algorithm briefly reported here [2]. One can quote as examples, elimination of redundant tests, verification of correct uses of operations, supplying of diagnostic informations, choice of types or organization of data structures in the case of very high level languages, etc.

9 - BIBLIOGRAPHY -

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

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

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

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

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

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

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

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

[9] - B. WEGBREIT
"Property extraction in well-founded property sets"
Harvard University, Cambridge, Mass., Feb. 1973

[10] - N. WIRTH
"The programming language PASCAL"
Acta Informatica, 1, 1971, pp. 35 - 63

-:-:-:-:-

Authors' Address :

P. COUSOT, R. COUSOT
Laboratoire d'Informatique (D 319)
Boîte Postale 53
38041 GRENOBLE Cédex
France