STATIC VERIFICATION OF DYNAMIC TYPE PROPERTIES OF VARIABLES

Patrick COUSOT and Radhia COUSOT
ABSTRACT: For high level languages, compile time type verifications are usually incomplete, and dynamic coercion 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, etc... We present a general algorithm allowing most of these verifications to be done at compile time. The static analysis of a program consists of an abstract evaluation of the program (RUN, CONTEXT,スーペリア, PROPERTY, ...) which computes, by successive approximations, an abstract context at every program point. An abstract value denotes a set of execution values or properties of it, satisfying a number of dynamic conditions. An abstract interpreter defines the sequencing of abstract evaluation through the different paths of the program, and builds a union of contexts resulting from the junction 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
38041 GRENOBLE Cedex - France -
LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication elsewhere and has been issued as a Research Report for early dissemination of its contents. As a courtesy to the intended publisher, it should not be widely distributed until after the date of outside publication.

Copies may be requested from :
Service des Polygraphes
Laboratoire d'Informatique
U.S.M.G.
Boîte Postale 53
38041 GRENOBLE Cédez 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 usually incomplete.
In most examples given here, \( \mathbb{V}_c \) will be the set of integers and \( \mathbb{V}_a \) the set of intervals of integers. If \( \mathbb{V}_c = \mathbb{Z} \) 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 \( a \in \mathbb{Z}, b \in \mathbb{Z} \) and \( a \leq b \).

The correspondence between a set of concrete values and an abstract value, is established by the "abstraction function" \( \varnothing : \) :

\[
\begin{array}{ccc}
\mathbb{V}_c & \xrightarrow{\varnothing} & \mathbb{V}_a \\
V & \Rightarrow & V \\
\end{array}
\]

Example :
\[
S \subseteq \mathbb{Z}, \quad \varnothing(S) = \left[ \min (x), \max (y) \right] \\
\text{ where } x \in S, \quad y \in S
\]

Another function, \( \gamma \), gives the concrete form of an abstract value :

\[
\begin{array}{ccc}
\mathbb{V}_a & \xleftarrow{\gamma} & \mathbb{V}_c \\
V & \Leftarrow & V \\
\end{array}
\]

Example :
\[
\gamma([a, b]) = \{ x \mid (x \in \mathbb{Z}) \land (a \leq x \leq b) \}
\]

The functions \( \varnothing \) and \( \gamma \) are defined such that they verify :

\[
(\forall s \in \mathbb{V}_c, s \subseteq \gamma(\varnothing(s)))
\]

and

\[
(\forall v \in \mathbb{V}_a, v = \varnothing(\gamma(v))).
\]

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 :

\[
\begin{array}{ccc}
\mathbb{V}_a \times \mathbb{V}_a & \xrightarrow{\bar{\cup}} & \mathbb{V}_a \\
\text{ and } & \Rightarrow & \text{ and } \\
\end{array}
\]

Example :
\[
[a_1, b_1] \bar{\cup} [a_2, b_2] = [\min(a_1, a_2), \max(b_1, b_2)]
\]

The abstraction function \( \varnothing \) is assumed to be an homomorphism from \((\mathbb{V}_c, \cup)\) into \((\mathbb{V}_a, \bar{\cup})\) :

\[
\varnothing(s_1, s_2) \in \mathbb{V}_c^2 : \quad \varnothing(s_1 \cup s_2) = \varnothing(s_1) \bar{\cup} \varnothing(s_2)
\]
This implies that \( \tilde{u} \) has the associativity, commutativity and idempotency properties, and that the zero element \( \emptyset \) of \( \tilde{u} \) is also \( \emptyset(\emptyset) \) where \( \emptyset \) is the empty set. \( \emptyset \) is called the undefined abstract value.

Corresponding to the inclusion \( \subseteq \) of sets of concrete values, the abstract evaluation uses the inclusion \( \subseteq \) of abstract values, which is defined by:

\[
V(v_1, v_2) \in V_a^2
\]

\[
\{v_1 \leq v_2\} \iff \{v_1 \tilde{u} v_2 = v_2\}
\]

and

\[
\{v_1 < v_2\} \iff \{(v_1 \leq v_2) \land (v_1 \neq v_2)\}
\]

From this definition and the hypothesis on \( \tilde{u} \), \( \subseteq \) can be shown to be a partial ordering, and \( \emptyset \) is included in every abstract value.

**Example:**

\[
\{[a_1, b_1] \leq [a_2, b_2]\} \iff \{(a_2 \leq a_1) \land (b_2 \geq b_1)\}
\]

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 \( \tilde{\nu} \):

\[
V_a \times V_a \xrightarrow{\tilde{\nu}} V_a
\]

**Example:**

\[
[a_1, b_1] \tilde{\nu} [a_2, b_2] =
\]

\[
[\begin{array}{l}
\text{if } a_2 < a_1 \text{ then } -\infty \text{ else } a_1, \\
\text{if } b_2 > b_1 \text{ then } +\infty \text{ else } b_1
\end{array}]
\]

For every particular abstract evaluation, \( \tilde{\nu} \) must be defined such that:

- \( V(v_1, v_2) \in V_a^2 \), \((v_1 \tilde{u} v_2) \leq (v_1 \tilde{\nu} v_2)\) and
- every infinite sequence \( s_0, s_1, \ldots, s_n, \ldots \) of the form \( s_0 = \emptyset \),
\( s_1 = s_0 \tilde{\nu} v_1, \ldots, s_n = s_{n-1} \tilde{\nu} v_n, \ldots \) (where \( v_1, v_2, \ldots, v_n, \ldots \) are arbitrary abstract values), is not strictly increasing. (\( \neg(s_o \prec s_1 \prec \ldots \prec s_n \prec \ldots)\).
Example:

\[ \emptyset \bar{\bar{v}} [1, 10] = [1, 10] \]
\[ [1, 10] \bar{\bar{v}} [1, 11] = [1, +\infty] \]
\[ [1, +\infty] \bar{\bar{v}} [0, 12] = [-\infty, +\infty] \]

so that, in that case, the length of the sequences \(s_0, s_1, \ldots, s_n, \ldots\)

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 \(C\) of abstract contexts
is such that:

\[ I \times (V_a - \{\emptyset\}) \]
\[ C \in 2 \]

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

\[ \{\forall C \in C : \forall (i, j) \in I^2, \forall (v, u) \in (V_a - \{\emptyset\})^2, \]
\[ \{(i, v) \in C \land (j, u) \in C \land (i, v) \neq (j, u)\} \Rightarrow \{i \neq j\}. \]

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

\[ C(i) = \text{if } (\exists v \in (V_a - \{\emptyset\}) \mid (i, v) \in C) \text{ then } v \text{ else } \emptyset \text{ fi.} \]

Example:

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

In particular, we note \(\emptyset\) the empty abstract context.

The union \(C \bar{\bar{v}} C'\) of two contexts \(C\) and \(C'\), will be used for expressing, for
elementary, the context resulting from conditional statements. The widening
\(C \bar{\bar{v}} C'\) of contexts, will be used in loops. They are defined using the union
and widening of abstract values:

\[ C_1 \bar{\bar{v}} C_2 = \{(i, v) \mid (i \in I) \land (v \in (V_a - \{\emptyset\})) \land (v = C_1(i) \bar{\bar{v}} C_2(i))\} \]
$C_1 \triangleright C_2 = \{(i, v) \mid (i \in I) \land (v \in (V_\alpha - \{\}) \land (v = C_1(i) \triangleright C_2(i))\}$

We can show, for every identifier $i$, that:

$$(C_1 \triangleright C_2)(i) = C_1(i) \triangleright C_2(i)$$

and $(C_1 \triangleright C_2)(i) = C_1(i) \triangleright C_2(i)$

**Example:**

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

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

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

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

As before, we define the inclusion $\lessdot$ of contexts by

$$\{c, \lessdot, r\} \iff \{c, \triangleright, c_1 = c_2\}$$

and $\{c, \lessdot, c_2\} \iff \{c, \triangleright, C_1 \triangleright C_2\}$

It can be shown that this is equivalent to

$$\{c, \lessdot, c_2\} \iff \forall i \in I, c_1(i) \lessdot c_2(i)$$

From the algebraic structure $(V_\alpha, \triangleright, \triangleright, \lessdot, \lessdot)$ defined on abstract values, we can show that the same algebraic structure holds for $(C, \triangleright, \triangleright, \lessdot, \lessdot)$. In particular, $\lessdot$ is associative, commutative, idempotent and $\mathcal{F}$ is its zero element, $\lessdot$ is a partial ordering on $C$, $C_1 \triangleright C_2 \lessdot C_1 \triangleright C_2$ for every $(C_1, C_2)$, and there are no infinite strictly increasing sequence $S_0 \lessdot S_1 \lessdot \ldots$ $\ldots \lessdot S_n \lessdot \ldots$ of abstract contexts of the form

$S_0 = \mathcal{F}, S_1 = S_0 \triangleright C_1, \ldots, S_n = S_{n-1} \triangleright C_n, \ldots$ 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 function $\mathcal{J}$ defines for any assignment or test program unit $n$ and input context $C$, an output context $\mathcal{J}(n, C)$, (or two when $n$ is a test node). The application $\mathcal{J}$ 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:

$$v := f(v_1, \ldots, v_m)$$

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:

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

$$\forall C \in \mathfrak{C}, \mathcal{J}(n, C)(v) = \emptyset \left[ \{f(v_1, \ldots, v_m) | (v_1, \ldots, v_m) \in \gamma(C(v_1)) \times \ldots \times \gamma(C(v_m)) \} \right]$$

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 $(v_1, \ldots, v_m)$ of $(v_1, \ldots, v_m)$ are chosen in the input context $C$.

Examples:

$$C = \emptyset$$

$$n \xrightarrow{x := 10}$$

$$\mathcal{J}(n, C) = \{(x, [10, 10])\}$$
In the case of that specific application an interval arithmetic [7] is used for defining \( \mathcal{J}(n, C) \):

\[
C(x) = [1, 10] \\
C(y) = [-2, 3] \\
C(x+y) = [1+(-2), 10 + 3] = [-1, 13] \\
C(1) = [1, 1] \\
C[x+y+1] = [-1, 13] + [1, 1] = [0, 14]
\]

We have \( C(y) = \emptyset \), so that the expression \( x + y \) is undefined, therefore \( \mathcal{J}(n, C)(x) = \emptyset \).

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

\( \forall n \in \mathbb{N}_T \) (the set of test nodes), \( n \) is of the form:

\[\begin{array}{c}
C \\
\end{array}\]

\[\begin{array}{c}
\mathcal{O}(v_1, \ldots, v_m) \\
\end{array}\]

\[\begin{array}{c}
\text{true} \\
C_T \\
\end{array}\]

\[\begin{array}{c}
\text{false} \\
C_F \\
\end{array}\]
where \(Q(v_1, ..., v_m)\) is a boolean expression without side-effect depending on the variables \(v_1, ..., v_m\). Then we define \(J(n, C) = (C_T, C_F)\) such that, \(\forall i \in I\)

\[
C_T(i) = \bigcap \{1| \gamma(C(i)) \land (3(v_1, ..., v_m) \in \gamma(C(v_1)) \times ... \times \gamma(C(v_m)) \land Q(v_1, ..., v_m)) \} \\
C_F(i) = \bigcap \{1| \gamma(C(i)) \land (\exists(v_1, ..., v_m) \in \gamma(C(v_1)) \times ... \times \gamma(C(v_m)) \land \neg Q(v_1, ..., v_m)) \}
\]

On the true edge for example, the abstract value of a variable \(i\) is the abstraction of the set of values \(1\) chosen in the input context \(C\), for which the evaluation of the predicate \(Q\) in context \(C\) may yield the value "true".

**Examples:**

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

**Example 1:**

- \(C_T = \{(x, [10, +\infty]), (y, [-1, +1])\}\)
- \(C_F = \emptyset\)

\[
C = \{(x, [-\infty, +\infty])\}
\]

**Example 2:**

- \(C_T = \{(x, [0, +\infty])\}\)
- \(C_F = \{(x, [-\infty, -1])\}\)

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

**Example 3:**

- \(C_T = \{(x, [1, +\infty]), (y, [1, 10])\}\)
- \(C_F = \{(x, [-\infty, 9]), (y, [1, 10])\}\)
In the case of that specific application, the treatment of conditional 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 interpreter 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-parallelly, 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 $\emptyset$.

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

\[
C_1 \quad C_2 \quad \ldots \quad C_m
\]

\[
C_{n,1} \quad C_{n,2} \quad \ldots \quad C_{n,m}
\]
If the $j^{th}$ pass on a junction node $n \in N_j$ (loop junction nodes) has associated the context $S_j$ to the output arc $\alpha$ of that node (or $S_0$ has been initialized to the empty context), then the context associated to $\alpha$ on the $j+1^{th}$ pass, will be:

$$S_{j+1} = S_j \mathcal{V} \left( \bigcup_{i \in [1,m]} \mathcal{V} \left( C_i, j+1 \right) \right)$$

**Example:**

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

$S_1 = \{(x, [1, 1])\}$

$S_2 = S_1 \mathcal{V} (C_1 \mathcal{V} C_2)$

$= S_1 \mathcal{V} \{(x, [1, 1] \mathcal{V} [2, 2])\}$

$= \{(x, [1, 1])\} \mathcal{V} \{(x, [1, 2])\}$

$= \{(x, [1, 1] \mathcal{V} [1, 2])\}$

$= \{(x, [1, \infty])\}$

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 interpreter starts with the empty context $\emptyset$ 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-parallelly, 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 $\mathcal{J}$ is an increasing function for $\preceq$ in $\mathcal{C}$. It can be shown that:

$$\{C' \preceq C\} \Rightarrow \{\forall n, \mathcal{J}(n, C') \preceq \mathcal{J}(n, C)\}$$

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 interpreter is now stated:
procedure abstract interpretation (graph) ;
begin
  for each arc of graph do local context (arc) := $\emptyset$ 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), $\mathcal{U}$ (node, local context (input arc))) ;
        test node ->
          $(C_n, C_f) := \mathcal{U}$ (node, local context (input arc)) ;
          assign output context (true-exit-arc (node), $C_f$) ;
          assign output context (false-exit-arc (node), $C_n$) ;
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 \( \alpha \), then for every identifier \( i \) of the program, and every actual execution of that program:

- if \( C(i) = \emptyset \), then \( i \) is never initialized on arc \( \alpha \), or \( \alpha \) is on a dead path.
- if \( C(i) \) is some abstract value \( v \), then the concrete value of \( i \) on arc \( \alpha \) within every execution path \( p \) containing \( \alpha \) 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:

```
\[ a \]
\[ b \]
\[ c \]
\[ d \]
\[ e \]
\[ f \]
```

The following table shows the analysis of this program graph, with the specific abstract interpretation we have chosen as example in the paper:
<table>
<thead>
<tr>
<th>step</th>
<th>node</th>
<th>input arc</th>
<th>local context (input arc)</th>
<th>output context</th>
<th>execution paths</th>
<th>junctions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 a</td>
<td>b</td>
<td>c</td>
<td>1, [1, 1]</td>
<td>1, [1, 1]</td>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td>2 b</td>
<td>c</td>
<td>2</td>
<td>1, [1, 1]</td>
<td>1, [1, 1]</td>
<td>(2)</td>
<td></td>
</tr>
<tr>
<td>3 c</td>
<td>2</td>
<td>6</td>
<td>1, [1, 1]</td>
<td>1, [1, 1]</td>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td>4 d</td>
<td>3</td>
<td>6</td>
<td>1, [1, 1]</td>
<td>1, [1, 1]</td>
<td>(4)</td>
<td></td>
</tr>
<tr>
<td>5 f</td>
<td>5</td>
<td>6</td>
<td>1, [1, 1]</td>
<td>1, [2, 2]</td>
<td>(5)</td>
<td></td>
</tr>
<tr>
<td>6 e</td>
<td>5</td>
<td>6</td>
<td>1, [1, 1]</td>
<td>1, [2, 2]</td>
<td>(6)</td>
<td></td>
</tr>
<tr>
<td>7 d</td>
<td>3</td>
<td>6</td>
<td>1, [1, 1]</td>
<td>1, [1, 1]</td>
<td>(7)</td>
<td></td>
</tr>
<tr>
<td>8 c</td>
<td>2</td>
<td>6</td>
<td>1, [1, 1]</td>
<td>1, [1, 1]</td>
<td>(8)</td>
<td></td>
</tr>
<tr>
<td>9 e</td>
<td>5</td>
<td>6</td>
<td>1, [1, 1]</td>
<td>1, [2, 2]</td>
<td>(9)</td>
<td></td>
</tr>
<tr>
<td>10 f</td>
<td>5</td>
<td>6</td>
<td>1, [1, 1]</td>
<td>1, [2, 101]</td>
<td>(10)</td>
<td></td>
</tr>
<tr>
<td>11 c</td>
<td>6</td>
<td>6</td>
<td>1, [1, 1]</td>
<td>1, [2, 101]</td>
<td>(11)</td>
<td></td>
</tr>
<tr>
<td>12 c</td>
<td>2</td>
<td>6</td>
<td>1, [1, 1]</td>
<td>1, [1, 1]</td>
<td>(12)</td>
<td></td>
</tr>
<tr>
<td>13 c</td>
<td>3</td>
<td>6</td>
<td>1, [1, 1]</td>
<td>1, [1, 1]</td>
<td>(13)</td>
<td></td>
</tr>
</tbody>
</table>

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.
In PASCAL or LIS like languages, where lwb, upb and m should have been declared of type 1..101,0100 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:

\[
\text{dubious} \leq \text{not-nil} \leq \text{nil} 
\]

In the case of a finite number of abstract values, the widening V is taken to be L. The problem consists in finding the Kth value of a linear linked list L:

\[
\text{value next} 
\]

The intended solution, with its analysis is the following:

\[
\{(K, i, -, -), (1, !!ri''ii)} \]

if \[ i < 0 \text{ then return } i; \]

\[
cursor :: L; 
\]

\[
(\{(K, [1, +\*]), (cursor, dubious),,,\} 
\]

if cursor : nil then

\[
K :: K - 1; 
\]

\[
(\{(ii, [1, +\*], (curs,>r, dubiorrs),,,\} 
\]

else

\[
(\{(K, t1, +\*]), (curscr, not-nji), ...} 
\]

LUP 

\[
\text{to} \] cursor :: next {cur:sor) 
\]

\[
(\{..., (u.so, dubious), ..,\} 
\]

\[
(..., r-r, +-l), (cursor, dubious), (L, dubious) \}; 
\]

\[
IBl ... v.llue (crirsor) . .. 
\]
It is shown, at line [a] that "cursor" in \texttt{next(cursor)}", is not a \texttt{nil} pointer, and it has been taken account of the fact that the function \texttt{next} delivers a \texttt{nil} or \texttt{not-nil} pointer. On the other hand, "cursor" might be \texttt{nil} at line [b], 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 -

"Computer proofs of limit theorems"

"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

"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

"A unified 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