STATIC DETERMINATION OF DYNAMIC PROPERTIES OF GENERALIZED TYPE UNIONS

Patrick COUSOT Radhia COUSOT

DATA TYPES / TYPE CHECKING

Among other purposes, the type of a value serves to staticly determine which operations may be applied to that value.

operation → $\underline{op}(x : t)$
value → $\alpha : t'$
type checking :

$$\{\underline{op}(\alpha) \text{ is "meaningful"}\} \Leftrightarrow \{t' = t\}$$

$$t' \subset t$$

This rule is not obeyed in pratice (ALGOL 68, PASCAL ....) :

- Static type checking is not sufficient :

  Dereferencing X↑....
  - X is necessarily of pointer type
  - not sufficient if X is <u>nil</u>

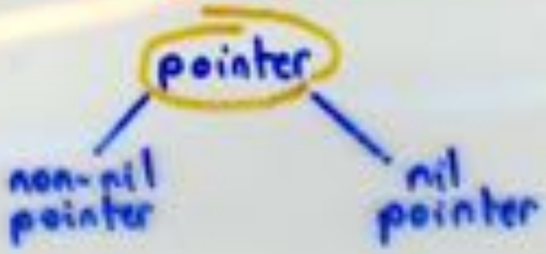
- Type checking is not compile-time :

  ```
  I, J : 1..100 ;
  T : array 1..100 of ...
  ```

  Array access T[I+J] ?

COMPILE-TIME TYPE SYSTEM
&
RUN-TIME TYPE SYSTEM

- implicit run-time types are more refined than the explicit compile-time types provided by classical languages :



- run-time type checking is strong (it provides a necessary and sufficient condition for operations to be applicable to values).

- run-time type checking is often redundant :

```
if x ≠ nil then
      x↑...
      ...
fi;
```

→ ...

The run-time type system should be used at compile-time

that is :

1 _ Classical data types should be refined :

type = hierarchy of sub-types

2 _ Strong static type checking should be the rule :

One should determine for each occurrence of a variable what range of sub-types its values may assume during execution.

- Why isn't it the case ? (→2)
- What are the possible remedies ?

Two approaches :

2.1 ⇒ "compiler design" approach :
Perform a global analysis of the program for local type determination.

2.2 ⇒ "language design" approach :
Design syntactic constructs for local type determination.

COMPILER DESIGN APPROACH
(Example : pointer handling)

Searching a linked linear list

- global type constrains (declarations) :

<u>type</u> cell = <u>record</u> next : <u>pointer</u> cell ; .... <u>end</u>
L : <u>pointer</u> cell ;
...
<u>type</u> plex = <u>record</u> next : <u>pointer</u> plex ; ... <u>end</u>

```
      pt := L;
1 ---
      while pt ≠ nil do
2 ---
          pt := pt↑.next;
3 ---
      od;
4 ---
```

Type analysis :

- type system :

- local type constrains :

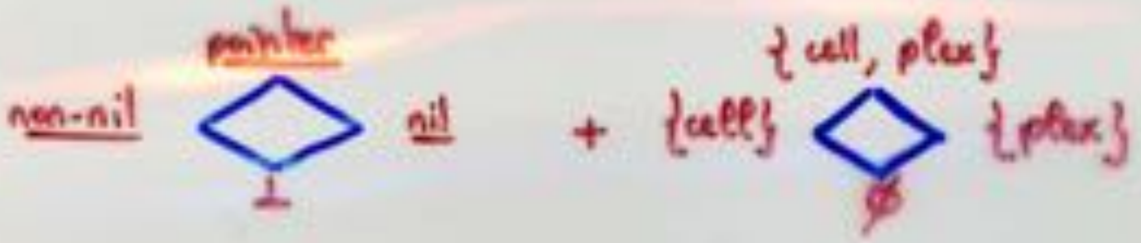
$$\begin{cases} pt_1 = \langle \underline{\text{pointer}}, \{cell\} \rangle \\ pt_2 = (pt_1 \ \underline{\text{or}} \ pt_3) \ \underline{\text{and}} \ \underline{\text{non-nil}} \\ pt_3 = NEXT\uparrow(pt_2) \\ \quad\;\; = \langle \underline{\text{pointer}}, pt_2[e] \rangle \\ pt_4 = (pt_1 \ \underline{\text{or}} \ pt_3) \ \underline{\text{and}} \ \underline{\text{nil}} \end{cases}$$

— local type determination

consists in solving the system of equations :

$$\begin{cases} pt_1 = \langle \underline{\text{pointer}}, \{\text{cell}\} \rangle \\ pt_2 = (pt_1 \ \underline{\text{or}} \ pt_3) \ \underline{\text{and}} \ \underline{\text{non-nil}} \\ pt_3 = \langle \underline{\text{pointer}}, pt_2[\ell] \rangle \\ pt_4 = (pt_1 \ \underline{\text{or}} \ pt_3) \ \underline{\text{and}} \ \underline{\text{nil}} \end{cases}$$

resolution by successive approximations :



— $pt_1 = pt_2 = pt_3 = pt_4 = \langle \perp, \emptyset \rangle$

— $pt_1 = \langle \underline{\text{pointer}}, \{\text{cell}\} \rangle$

$pt_2 = (\langle \underline{\text{pointer}}, \{\text{cell}\} \rangle \ \underline{\text{or}} \ \langle \perp, \emptyset \rangle) \ \underline{\text{and}} \ \underline{\text{non-nil}}$

$= (\langle \underline{\text{pointer}}, \{\text{cell}\} \rangle) \ \underline{\text{and}} \ \underline{\text{non-nil}}$

$= \langle \underline{\text{non-nil}}, \{\text{cell}\} \rangle$

$pt_3 = \langle \underline{\text{pointer}}, \{\text{cell}\} \rangle$

$pt_4 = (\langle \underline{\text{pointer}}, \{\text{cell}\} \rangle \ \underline{\text{or}} \ \langle \underline{\text{pointer}}, \{\text{cell}\} \rangle) \ \underline{\text{and}} \ \underline{\text{nil}}$

$= \langle \underline{\text{nil}}, \text{cell} \rangle$

— $pt_1 = \langle \underline{\text{pointer}}, \{\text{cell}\} \rangle$

$pt_2 = (\langle \underline{\text{pointer}}, \{\text{cell}\} \rangle \ \underline{\text{or}} \ \langle \underline{\text{pointer}}, \{\text{cell}\} \rangle) \ \underline{\text{and}} \ \underline{\text{non-nil}}$

$= \langle \underline{\text{non-nil}}, \{\text{cell}\} \rangle$

$pt_3 = \langle \underline{\text{pointer}}, \{\text{cell}\} \rangle$

$pt_4 = \langle \underline{\text{nil}}, \{\text{cell}\} \rangle$

— local type checking : pt not nil at point ℓ .

LANGUAGE DESIGN APPROACH
(Example : pointer handling)

Searching a linked linear list 

```
1 --.-  pt := L;
        do pt in
           - non-nil cell pt' →  2 ¦ pt := pt'.next ¦ 3 ;
           - nil cell         →    ¦ exit ;
        od;                        4
5 --.-
```

- local type constrains :

$$\begin{cases} pt_1 = \langle \underline{pointer}, \{cell\} \rangle \\ pt_2 = (pt_1 \text{ or } pt_3) \ \underline{and} \ \langle \underline{non\text{-}nil}, \{cell\} \rangle \\ pt'_2 = \langle \underline{non\text{-}nil}, cell \rangle \\ pt_3 = \langle \underline{pointer}, pt'_2[e] \rangle \\ pt'_3 = pt'_2 \\ pt_4 = (pt_1 \text{ or } pt_3) \ \underline{and} \ \langle \underline{nil}, \{cell\} \rangle \\ pt_5 = pt_4 \end{cases}$$

- local type checking :
  - no need to solve the equations. $(pt'_2 \neq \underline{nil})$
  - the features necessitating information propagation are eliminated.

pointer := **nil** implicitly propagated type = pointer

~~**non-nil** := **pointer**~~ forbidden since the implicitly propagated type is non-nil.

OUR POINT OF VIEW :

- The implicit run-time data types should be made available to the programmer :

Ex : binary tree traversal

```
type node = record left, right : pointer node ; ... end ;
tree : pointer node ;
stack : stack of non-nil node ;

empty(stack);
do
   if tree ≠ nil then
        push tree on stack;
        tree := tree.left;
   elif stack ≠ empty then
        pop tree from stack;
        tree := tree.right;
   else exit;
   fi;
od;
```

- The type of an object should be considered to be a local property (a subtype of the globally declared type of the object).

  Ex: Collections

- Type information should be propagated by some "closure" mechanism, with the aid of local subtype discriminating constructs (for dealing with undecidable cases where subtype discovery is "impossible")

  Ex: integer subrange type

- Introducing the need for COLLECTIONS

The type of an object defines our that object relates to other objects.

Ex : Representation of sequences of integers.

- fixed length n

```
S1, S2 : array 1..n of integer ;
```

→ no sharing between the two sequences S_1 and S_2

- variable length

```
type cell = record
                value : integer ;
                next  : ↑ cell ;
            end;
S1, S2 : ↑ cell ;
```

→ operations on S_1 may have side effects on S_2 (because of possible sharings)

COLLECTIONS (≃ EUCLID)

Sets of disjoined dynamic variables

```
var c1 : collection of elem1
type elem1 = record
                 var next : ↑c1
                 ....
             end
var s1 : ↑c1
```

```
var c2 : collection of elem2
type elem2 = record
                 var next : ↑c2
                 ....
             end
var s2 : ↑c2
```

s_1 and s_2 have different types → the list handling algorithms must be written twice.

→ type parameterization is necessary

EUCLID'S COLLECTION ≃

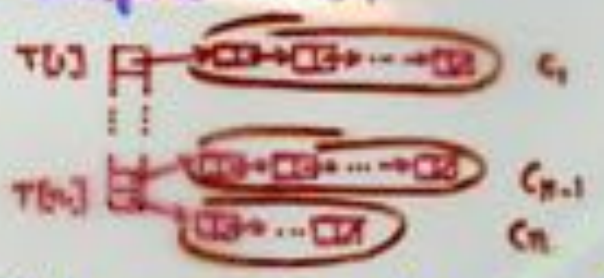
```
type elem (c: collection of elem(c)) =
    record
        var next : ↑c
        ....
    end;
{ var c1 : collection of elem (c1)
{ var s1 : ↑c1
{ var ce : collection of elem (ce)
{ var se : ↑ce
```

parameterized list handling algorithms :

```
procedure insert (c : collection of elem(c),
                  var s : ↑c,
                  ...)
    ...
```

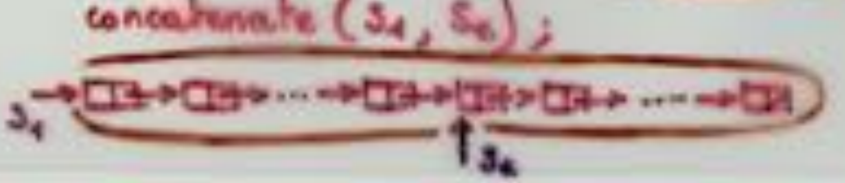
Defects :

- The number of collections is determined at compile time.



- Collections are declared globally and cannot be locally refined.

concatenate (s_1, s_2);



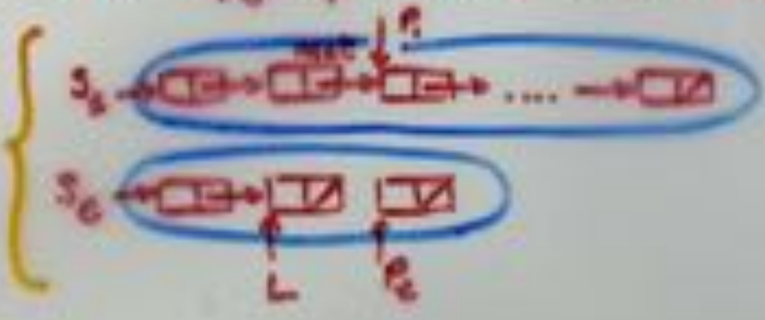
COMPILER DISCOVERY OF LOCAL COLLECTIONS

Ex: copy of a linked linear list

```
procedure copy (S1 : list ; var S2 : list);
    var P1, P2, L : list ;
begin
      P1 := S1 ;   S2 := nil;   L := nil ;

      while P1 ≠ nil do

          new P2 ;   P2↑.next := nil ;

→         if L = nil then

              S2 := P2;

          else

              L↑.next := P2 ;

          fi;

          L := P2 ;   P1 := P1.next;

      od;

end ;
```

COMPILER DISCOVERY OF LOCAL COLLECTIONS

Ex : copy of a linked linear list



/S1 P1 /S2 L P2/

```
procedure copy (S1 : list ; var S2 : list);
    var P1, P2, L : list ;
begin /S1 S2 / P1 P2 L/
    P1 := S1 ;  S2 := nil ;  L := nil ;
    /S1 P1 / S2 / P2 / L /
    while P1 ≠ nil do
        /S1 P1 / P2 S2 L/
        new P2 ;  P2↑.next := nil ;
→       /S1 P1 / P2 / S2 L/
        if L = nil then
            /S1 P1 / P2 / S2 / L/
            S2 := P2 ;
            /S1 P1 / P2 S2 / L/
        else
            /S1 P1 / P2 / S2 L/
            L↑.next := P2 ;
            /S1 P1 / P2 S2 L/
        fi ;
        /S1 P1 / P2 S2 L /
        L := P2 ;  P1 := P1.next ;
        /S1 P1 / P2 S2 L /
    od ;
    /S1 / P1 / P2 S2 L/
end ;
```

LOCAL CONSTRAINS ON COLLECTIONS.

- $C_1 = /x\, x_1 \ldots x_n / y_1 \ldots y_m /$

  x := nil

  if x = nil then ...

  new x

  $C_2 = /x/x_1 \ldots x_n / y_1 \ldots y_m /$

  = extract (x, C_1)

- $$\begin{tikzcd} C_1 = /xy/tw/zu/ \arrow[dr] & & C_2 = /yz/u/w/ \arrow[dl] \\ & C_1 \bar{\cup} C_2 = /xyzu/tw/ & \end{tikzcd}$$

- $C_1 = /x\, x_1 \ldots x_n / y\, y_1 \ldots y_m / z_1 \ldots z_k /$

  x := y

  $C_2 = /x_1 \ldots x_n / x\, y\, y_1 \ldots y_m / z_1 \ldots z_k /$

  = extract $(x, C_1) \bar{\cup} /x\, y/$

- $C_1 = /x\, x_1 \ldots x_n / y\, y_1 \ldots y_m / z_1 \ldots z_k /$

  x↑.next := y

  $C_2 = /x\, x_1 \ldots x_n\, y\, y_1 \ldots y_m / z_1 \ldots z_k /$

  $C_2 = C_1 \bar{\cup} /xy/$

  (Deep modifications in the list structure are not observed).

LOCAL CONSTRAINS ON COLLECTIONS (EXAMPLE)

(16

```
procedure copy (S1 : list ; var S2 : list);
    var P1, P2, L : list;
begin
0 ——
    P1 := S1 ;  S2 := nil ;  L := nil ;
1 ——
    while P1 ≠ nil do
2 ——
        new P2 ;   P2↑.next := nil ;
3 ——
        if L = nil then
4 ——
            S2 := P2;
5 ——
        else
6 ——
            L↑.next := P2;
7 ——
        fi;
8 ——
        L := P2;   P1 := P1.next;
9 ——
    od;
10 ——
end;
```

$C_0 = /S_1\ S_2\ /\ P_1\ P_2\ L/$

$C_1 = \underline{ext}(L, \underline{ext}(S_2, \underline{ext}(P_1, C_0)\ \bar{\cup}\ /P_1\ S_1/))$

$C_2 = C_1\ \bar{\cup}\ C_9$

$C_3 = \underline{ext}(P_2, C_2)$

$C_4 = \underline{ext}(L, C_3)$

$C_5 = \underline{ext}(S_2, C_4)\ \bar{\cup}\ /S_2\ P_2/$

$C_6 = C_3$

$C_7 = C_6\ \bar{\cup}\ /L\ P_2/$

$C_8 = C_5\ \bar{\cup}\ C_7$

$C_9 = \underline{ext}(L, C_8)\ \bar{\cup}\ /L\ P_2/$

$C_{10} = \underline{ext}(P_1, C_1\ \bar{\cup}\ C_9)$

SOLVING THE EQUATIONS

- $C_0 = C_1 = \dots = C_{10} = //$
- $C_0 = /S_1 S_2 / P_1 P_2 L/$

  $C_1 = \underline{\text{ext}}(L, \underline{\text{ext}}(S_2, \underline{\text{ext}}(P_1, C_0) \bar{\cup} /P_1 S_1/))$

  $= \underline{\text{ext}}(L, \underline{\text{ext}}(S_2, /S_1 S_2/P_1/P_2 L/ \bar{\cup} /P_1 S_1/))$

  $= \underline{\text{ext}}(L, \underline{\text{ext}}(S_2, /P_1 S_1 S_2/P_2 L/)$

  $= /P_1 S_1/S_2/P_2/L/$

  $C_2 = C_1 \bar{\cup} C_9 = C_1 \bar{\cup} // = C_1$

  $= /P_1 S_1/S_2/P_2/L/$

  $C_3 = \underline{\text{ext}}(P_2, C_2)$

  $= /P_1 S_1/S_2/P_2/L/$

  $C_4 = \underline{\text{ext}}(L, C_3)$

  $= /P_1 S_1/S_2/P_2/L/$

  $C_5 = \underline{\text{ext}}(S_2, C_4) \bar{\cup} /S_2 P_2/$

  $= /P_1 S_1/S_2 P_2/L/$

  $C_6 = C_3 = /P_1 S_1/S_2/P_2/L/$

  $C_7 = C_6 \bar{\cup} /L P_2/$

  $= /P_1 S_1/S_2/P_2 L/$

  $C_8 = C_5 \bar{\cup} C_7 = /P_1 S_1/S_2 P_2/L/ \bar{\cup} /P_1 S_1/S_2/P_2 L/$

  $= /P_1 S_1/S_2 P_2 L/$

  $C_9 = \underline{\text{ext}}(L, C_8) \bar{\cup} /L P_2/$

  $= /P_1 S_1/S_2 P_2 L/$

- $C_2 = C_1 \cup C_9$

  $= /P_1 S_1/S_2/P_2/L/ \bar{\cup} /P_1 S_1/S_2 P_2 L/$

  $= /P_1 S_1/S_2 P_2 L/$

SOLVING THE EQUATIONS (Continued)

$C_2 = /P_1 S_1 / S_2 P_2 L/$

$C_3 = \underline{\mathrm{ext}}(P_2, C_2)$

$= /P_1 S_1 / P_2 / S_2 L/$

$C_4 = \underline{\mathrm{ext}}(L, C_3)$

$= /P_1 S_1 / P_2 / S_2 / L/$

$C_5 = \underline{\mathrm{ext}}(S_2, C_4) \, \overline{\cup} \, /S_2 P_2)$

$= /P_1 S_1 / P_2 S_2 / L/$

$C_6 = C_3$

$= /P_1 S_1 / P_2 / S_2 L/$

$C_7 = C_6 \, \overline{\cup} \, /L P_2/$

$= /P_1 S_1 / P_2 S_2 L/$

$C_8 = C_5 \, \overline{\cup} \, C_7$

$= /P_1 S_1 / P_2 S_2 / L/ \, \overline{\cup} \, /P_1 S_1 / P_2 S_2 L/$

$= /P_1 S_1 / P_2 S_2 L/$

$C_9 = \underline{\mathrm{ext}}(L, C_8) \, \overline{\cup} \, /L P_2/$

$= /P_1 S_1 / P_2 S_2 L/$

$C_{10} = \underline{\mathrm{ext}}(P_1, C_1 \, \overline{\cup} \, C_9)$

$= \underline{\mathrm{ext}}(P_1, /S_1 P_1 / S_2 / P_2 / L/ \, \overline{\cup} \, /P_1 S_1 / P_2 S_2 L/)$

$= \underline{\mathrm{ext}}(P_1, /S_1 P_1 / P_2 S_2 L/)$

$= /S_1 / P_1 / P_2 S_2 L/$

A next step would prove stabilization.

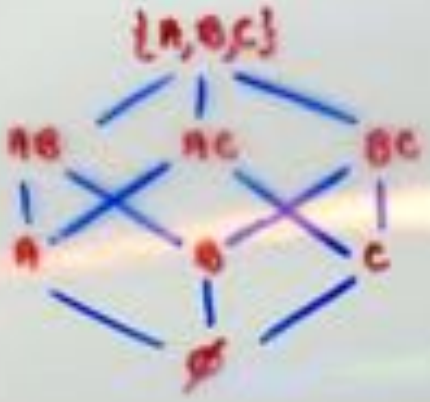
Final result : lists S_1 and S_2 are disjoined.

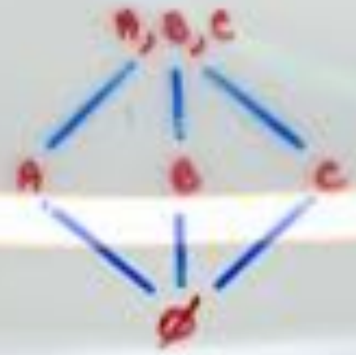
- Refinement

Records with variants

Associate with each collection the set of tags of all records in the collection.

Ex : 3 variants A, B, C





- Comparison with EUCLID :

Euclid's language design approach is more complicated and cannot lead to thin and local results

However

- Enforce a programming discipline
- Euclid's compilers can use the same program analysis technique.

A last example showing the need for LOCAL SUBTYPE DISCRIMINATING SYNTACTIC CONSTRUCTS.

integer subrange type.

```
1 — i := 1;
2 — while i ≤ 1000 do
3 —     i := i+1;
4 — od;
```

System of equations:

$$i_1 = 1..1$$
$$i_2 = (i_1 \cup i_3) \cap -\infty..1000$$
$$i_3 = i_2 + 1..1$$
$$i_4 = (i_1 \cup i_3) \cap 1001..+\infty$$

The global declaration:

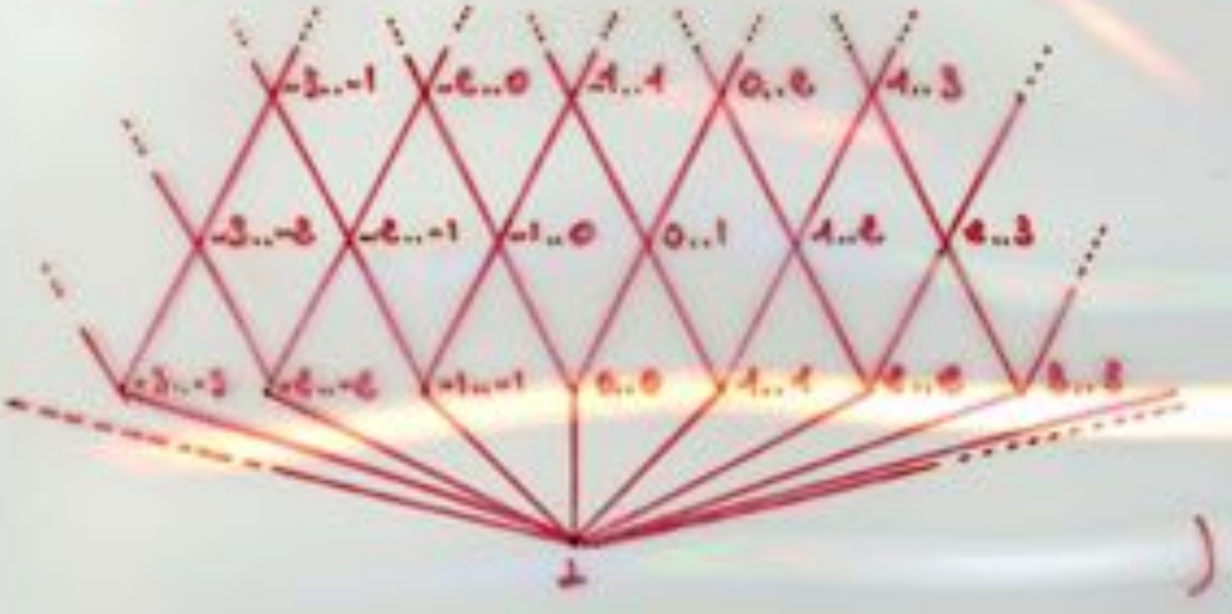
<u>var</u> i : 1..1001;

is nearly useless, since it only add an inequation:

$$i_1 \cup i_2 \cup i_3 \cup i_4 \subseteq 1..1001$$

compiler design approach

the problem of solving the equations is now <u>undecidable</u>. (the type system is infinite :



)

Only approximate solutions can be automaticly computed.

Ex :
```
{$1 \le i \le n$}
do
  — $i = 1$          → exit ;
  — $(i \bmod 2) = 0$ → $i := i/2$ ;
  — $(i \bmod 2) = 1$ → $i := 3i + 1$ ;
od;
```

In a very large number of cases the most beautiful equation solving methods will not be able to discover a good approximation of the exact local subranges of integer variables whereas the user will be able to provide this information. Then automatic verification will eventually be possible.

language design approach :

the solution of the equations is provided by the the user :

```
                                  3               4
                                  |               |
        i := 1;
1 —     do i in
2 —       — 1..1000 i' →  i := i' + 1 ;
          — 1001..1001 →  exit ;
        od;               |
                          5
```

and the 'closure' analysis of the program can be used to eliminate redundant tests.

$$\begin{aligned} i_2 &= i_1 \cup i_4 \\ &= i_1 \cup (i_3 + 1..1) \\ &= 1..1 \cup (1..1000 + 1..1) \\ &= 1..1 \cup 2..1001 \\ &= 1..1001 \end{aligned}$$

so that you can optimize in

```
i:=1;
do
  - i=1001 → exit;
  - otherwise → i:= i+1;
od;
```

In difficult cases the user will not be able to provide the solution, or its solution cannot be verified :

→ run time tests.

CONCLUSION

- A type system provides an abstract representation of sets of states of variables which can be used to simulate program execution.
- A type is generally the union of subtypes, which should be available to the user.
- Global type declarations are necessary but not sufficient.
- One must be able to locally determine the subtype of each variable occurence.
- This involves solving a system of equations.
- The language designer has
  - .to specify how to built that system of equations for each particular program,
  - .to specify a method for solving these equations
- This gives the language designer a criterion for deciding whether the programmer's aid will be necessary to guess an approximation of the exact solution.

- In case the programmer's aid is needed, the language designer has
  - . to design linguistic constructs which will provide this information.
  - . to design a static verifier of correctness of these information, as well as a run-time type checking mechanism in case static verification is impossible.

PROBLEM :

The implicit type system used in particular programs is much more refined than the type system provided by languages

→ In part the subject of that conferece