PROCEEDINGS OF AN ACM CONFERENCE ON LANGUAGE DESIGN FOR RELIABLE SOFTWARE

edited by
David B. Wortman

Sheraton-Crabtree Inn
Raleigh, North Carolina
March 28-30, 1977
STATIC DETERMINATION OF DYNAMIC PROPERTIES
OF GENERALIZED TYPE UNIONS
Patrick Cousot* and Radhia Cousot**
Laboratoire d'Informatique, U.S.M.G., BP. 53
38041 Grenoble Cedex, France

Abstract. The classical programming languages such as PASCAL or ALGOL 60 do not provide full data type security. Run-time errors are not precluded on basic operations. Type safety necessitates a refinement of the data type notion which allows subtypes. The compiler must also be able to ensure that basic operations are applicable. This verification consists in determining a local subtype of globally declared variables or constants. This may be achieved by improved compiler capabilities to analyze the program properties or by language constructs which permit the expression of these properties. Both approaches are discussed and illustrated by the problems of access to records via pointers, access to variants of record structures, determination of disjoint collections of linked records, and determination of integer subrange. Both approaches are complementary and a balance must be found between what must be specified by the programmer and what must be discovered by the compiler.

Key words and phrases : Type safety, type unions, subtype, data type, system of equations, type verification/discovery, error detection capabilities, abstract interpretation of programs, secure use of pointers/variants of record structures, domains/collections, integer subrange type, ALGOL 68, EUCLID, PASCAL.

CR categories : 4.12, 4.13, 4.2, 5.4.

1. Introduction

The type of an object defines how that object relates to other objects and which actions may be applied to it. Unfortunately the classical type systems of ALGOL 60[1973], PASCAL[1974], ALGOL 68 [1975] ... do not convey enough information to determine statically whether a given action applied to a value will be meaningful. For example, in ALGOL 60 the type procedure does not include the type of acceptable parameters, in ALGOL 68 the type reference ignores the fact that a reference may be dummy, in PASCAL type unions (variants of record structures) are unsafe because of the possibility of erring on the current alternative of the union. In all these languages the problem of subscript range is not safely treated by the type concept. Likewise, the classical type systems define only loose relationships between objects. For example, in PASCAL, a pointer to a record must be considered as potentially designating any record of a given type. One cannot express the fact that two linked linear lists of the same type do not intermix. Finally, the rules of the language or the programming discipline accepted by the programmer are not statically enfor-
ced by the compilers, so that run-time checks are
the widely used remedy. However these expensive run-
time checks are usually turned off before the "last"
programming error has been discovered.

In the interest of increased reliability of soft-
ware products, the language designer may reply upon:

- The design of a refined and safe type system, which
necessitates linguistic constructs which propaga-
tate strong type properties. The rules of the lan-
guage must then be checkable by a mere textual
scan of programs (e.g. ALGOL 68[1975] and EUCLID
[1976] provide a secure use of type unions). This
language design approach may degenerate to large
and baroque programming languages.

- The design of a refined compiler which performs
a static treatment of programs and provides im-
proved error-detection capabilities. The language
then remains simple and flexible, but security is
offered by compiler verifications (e.g. EUCLID
legality assertions which the compiler generates
for the verifier). This compiler design approach
may degenerate into futururistic and mysterious au-
tomatic program verifiers.

We illustrate the two approaches by means of examples:
The compiler techniques we propose for the static ana-
lysis of programs have a degree of sophistication
comparable to program optimization techniques ran-
ning security offered by full typing (within a suitable
linguistic framework to properly propagate strong
type properties), and the simplicity offered by the
flexible (but incomplete) classical type systems.

2. Nil and Non-nil Pointers

Among the objections against the use of point-
ers are the facts that they can lead to serious ty-
pe violations (PL/1) and that they may be left dangle-
ing. One can take care of these objections, by gua-
рантее that the type of the object pointed at (PASCAL
[1974] except for variant of records), and ensuring
that pointers point only to explicitly allocated
heap cells (disjoint from variable cells) which re-
main allocated until they are no longer accessible
(PASCAL[1974] when "dispose" is not used). However
a pointer may always have the nil value which points
to no element at all; this is a source of frequent
errors.

The type of a value may be viewed as a static
summary of the meaningful operations on that value.
However the operations prescribed by a syntactically
valid construct are not always dynamically mean-
ful. This is the case when dereferencing a pointer
value which happens to be nil.
The pointer type notion must then be refined so
that one can distinguish:
expensive.

- Safe language design, with strong typing i.e. a type system which ensures that any operation prescribed by a syntactically valid construct will always be dynamically meaningful. This type scheme must distinguish between nil and non-nil pointer types, disallow type violations (i.e. forbid the type of an object to be changed from the type "nil" or non-nil pointer, to the type "non-nil pointer") and syntactically check the correct use of operations (i.e. authorize dereferencing for non-nil pointers only).

- Compile time checks, to recognize the safe use of a type scheme which is too tolerant. We illustrate now this last strategy.

2.1 Static Correctness Check of Access to Records via Pointers

Consider the simple problem of searching for a record with value "n" in a linked linear list L:

\[ \begin{array}{c}
\text{L} \\
\text{value} \\
\to \text{a} \\
\to \text{b} \\
\to \text{c} \\
\to \text{w} \\
\end{array} \]

The PASCAL solution is given by PASCAL[1974] (p. 64) as follows:

\[
\begin{align*}
(1) & \quad \text{pt := L; b := true;} \\
(2) & \quad \{P1\} \\
(3) & \quad \text{while (pt <> nil) and b do} \\
(4) & \quad \{P2\} \\
(5) & \quad \text{if pt.t.value = n then} \\
(6) & \quad \quad b := \text{false} \\
(7) & \quad \{P3\} \\
(8) & \quad \text{else} \\
(9) & \quad \{P4\} \\
(10) & \quad \text{pt := pt.t.next;} \\
(11) & \quad \{P5\};
\end{align*}
\]

The above piece of program is correct with regard to accesses to records via pointers, since pt is not nil when dereferenced at line (5) and (10). This fact is established by the programmer using a simple propagation algorithm from the test of line (3). This reasoning is easily mechanized as follows: associate invariants P1, P2, P3, P4 and P5 to points (2), (4), (7), (9) and (11) respectively.

According to the semantics of the programming language PASCAL (Hoare and Wirth[1973]), these invariants are related as defined by the subsequent system of equations:

\[
\begin{align*}
(1) & \quad P1 = (pt = L) \land (b = true) \\
(2) & \quad P2 = (P1 \lor P5) \land ((pt <> \text{nil}) \land b) \\
(3) & \quad P3 = (P2 \land (pt.t.value = n)) \land (b = false) \\
(4) & \quad P4 = P2 \land (pt.t.value <> n) \\
(5) & \quad P5 = P3 \lor (3 pt.' | s^{0}_{pt.t} [P4] \land pt = pt'.next)
\end{align*}
\]

(Equation (5) has been deliberately oversimplified, see Dembinski and Schwartz[1975]).

Since in general it is undecidable to find a solution to systems such as the one above, we must consider simplifications (to the prejudice of the precision of our results). For that purpose we will ignore the existence of the boolean variable b, of the fields "value" in records of the linear list, and thus focusing on pointers. Moreover, we will consider only the pointer variable pt, and the following predicates on pt:

\[
\begin{align*}
\text{pt = nil, pt <> nil, (pt = \text{nil}) or (pt <> \text{nil})}
\end{align*}
\]

respectively denoted by \text{nil}, \text{non-nil}, \text{T}. These predicates form a complete lattice whose HASSE’s diagram is:

\[
\begin{array}{c}
\text{T} \\
\text{nil} \\
\text{non-nil} \\
\end{array}
\]

Where \text{1} is used to denote the fact that nothing is known about the variable pt.

Since we are only considering an oversimplified subset of the set of predicates, our system of equations can be simplified accordingly:

\[
\begin{align*}
(1') & \quad P1 = T \\
(2') & \quad P2 = (P1 \lor P5) \land \text{non-nil} \\
(3') & \quad P3 = P2 \\
(4') & \quad P4 = P2 \\
(5') & \quad P5 = P4 \lor T
\end{align*}
\]

(In equation (1) we consider \text{(pt = L)} since L may
be an empty or non-empty linear list, we get \( pt = \text{nil} \) or \( pt \neq \text{nil} \) denoted \( T \), in equation (5) we only consider the fact that the function 'next' (when defined) delivers a (nil or non-nil) pointer value which is assigned to \( pt \).

Our system of equations is of the form:

\[
\langle P_1, P_2, P_3, P_4, P_5 \rangle = F(\langle P_1, P_2, P_3, P_4, P_5 \rangle)
\]

where \( F \) is an order-preserving application from the complete lattice \( L^5 \) in itself. Therefore, the Knaster-Tarski theorem states that the application \( F \)

has a least fixpoint \( [\text{Tarski}[1955]] \). Moreover, since \( F \) is a complete ordering-preserving morphism from the complete lattice \( L^5 \) to itself, this least fixpoint can be defined as the limit of Kleene’s sequence, Kleene [1952]:

\[
\lambda_2 = \langle 1, 1, 1, 1, 1 \rangle \\
\lambda_3 = F(\lambda_2) \\
= \langle T, (1 \lor 1) \land \text{non-nil}, 1, 1, (1 \lor T) \rangle \\
= \langle T, 1, 1, 1, 1, T \rangle
\]

\[
\lambda_4 = F(\lambda_3) \\
= \langle T, (1 \lor T) \land \text{non-nil}, 1, 1, (1 \lor T) \rangle \\
= \langle T, \text{non-nil}, 1, 1, 1, T \rangle
\]

\[
\lambda_5 = F(\lambda_4) \\
= \langle T, (1 \lor T) \land \text{non-nil}, 1, 1, (1 \lor T) \rangle \\
= \langle T, \text{non-nil}, 1, 1, 1, T \rangle
\]

Thus, Kleene’s sequence converges in a finite number of steps, which is obvious since \( L^5 \) is a finite lattice. The solution to our system of equations tells us that \( P_2 = P_3 = P_4 = \text{non-nil} \) which according to our interpretation means that \( pt \) is not \text{nil} at line 4. (7) and (9) of our program, which implies that the accesses of records through \( pt \) at line 5 and (10) are statically shown to be correct. With regard to the value of \( P_1 \) and \( P_5 \), its interpretation is that \( pt \) may be \text{nil} at program points (2) and (11). In particular, the test on \( pt \) at line (3) may not be identically true.

The simple programmer's idea of generalizing constant propagation may be derived from the above Kleene's sequence when eliminating useless computations. A symbolic execution of the program (where elementary actions are interpreted according to the simplified equations previously established) gives the following computation sequence:

\[
P_1 = \tau, (P_2 = 1, i \in [2, 5])
\]

\[
P_2 = (P_1 \lor P_5) \land \text{non-nil} \\
= (\tau \lor 1) \land \text{non-nil} \\
= \text{non-nil}
\]

\[
P_3 = P_2 \\
= \text{non-nil}
\]

\[
P_4 = P_2 \\
= \text{non-nil}
\]

\[
P_5 = P_3 \lor \tau \\
= \text{non-nil} \lor \tau \\
= \tau
\]

\[
P_2 = (P_1 \lor P_5) \land \text{non-nil} \\
= (\tau \lor \tau) \land \text{non-nil} \\
= \text{non-nil}, \text{same as above, stop.}
\]

Kildall[1973] and Wegbreit[1975] algorithms have been recognized, they are "efficient" versions of the Kleene’s sequence. Following Sintzoff[1972] we call this technique the abstract interpretation of programs. Abstract since some details about the data of the program are forgotten, and interpretation since both a new meaning is given to the program text and the information is gathered about the program by means of an interpreter which executes the program according to this new meaning. We then get a static summary of some facets of the possible executions of the program. A theoretic framework of abstract interpretation of programs together with various examples are given in Cousot[1976].

2.2 A Safe Linguistic Framework to Handle Nil Pointers

A complete and satisfactory solution of the problem of dereferencing or assigning to a nil name [as in \text{ref real} (\text{nil}) := 3.14] is proposed by Meertens[1976] within the framework of ALGOL 68. The pointer types are restricted to non-nil values by exclusion of \text{nil}-names (this is achieved by not providing a representation for the \text{nil} symbol), so that any name refers to a value. The type \text{void} is used to represent \text{nil}-names. Finally the type of \text{nil} and non-nil pointers is the union of the previous ones.
For example we can write a construction like

```pascal
mode list = union [ref cell, void]
mode cell : struct (integer value, list next)
```

to represent linked linear lists. An empty list is represented by the value empty, the only void value. Our routine would have to be rewritten:

```pascal
list pt := L;
while case pt in
    (ref cell pt') => if value of pt' = n then false
    else
        (pt := next of pt'; true)
    fi,
out
exec
do skip od;
```

adequate syntactic constructs) is correct. Since in this example the type system is finite, both approaches are equivalent as far as type verifications are concerned.

5. Variants of Record Structures

3.1 Unsafe Type Unions in PASCAL

In ALCOL 68[1975] a variable may assume values of different types. The type of this variable is then said to be the union of the types of these values. In PASCAL[1974] the concept of type unions is embodied in the form of variants of record structures: a record type may be specified as consis-

```pascal

```
representation:
    alphanum.tag := int;
    writeln(alphanum.i);

(Note that the tag is appropriately set, but without
care about its value one can write as well:
    alphanum.c := 'H';
    writeln(alphanum.i);)

3.3 Safe Type Unions in ALGOL 68/EUCLID

Suggestions have been made to provide syntactic
structures which ensure that type-unions are safe,
i.e. compile-time checkable. Such features forbid
assignments to the tag fields and let the compiler
determine the current tag value from context using
a statement similar to the "inspect when" of SIMULA
[1974].

In ALGOL 68 [1975] we would write:

    mode charint = union (integer, character);
    integer digit; character letter;
    charint alphanum;

The tag field is hidden from the programmer, and
may be checked using conformity clauses.

The antagonism with PASCAL is more obvious in
EUCLID [1975] which handles variant records in a
type-safe, ALGOL 68-like manner. Since EUCLID al-
 lows parameterized-types, the tag will usually be
a formal parameter of the type declaration:

    type mode = (int, char)
    type charint (tag : mode) =

    record
      case tag of
        int => var i : integer; end int
        char => var c : character; end char
      end case
    end charint

When a variable of the record type "charint" is
declared, the actual tag parameter may be a con-
stant:

    var digit : charint (int)
    var letter : charint (char)

or any, which allows type unions:

    var alphanum : charint (any)

ALGOL 68 or EUCLID are type-safe when dealing with
type unions since:

- No assignments to the tag fields are authorized
  once they have been initialized.

- Uniting is allowed and safe:
  alphanum := letter;
  is legal, because the type of the right hand side
  value charint (char) may be coerced to the type
  of the left hand side variable charint (any) (the
  type charint (any) permits alphanum to hold either a
  value of type charint (char) or a value of type charint
  (int)).

- There is no de-uniting coercion, since if
  letter := alphanum
  were allowed, the principle of type-checking
  would be violated. The only way to retrieve an
  object which has been united and to retrieve it
  in its original type is by a discriminating case
  statement. This ensures that the type transfer
  is safe since the tag is explicitly tested:

    case discriminating x = alphanum on tag of
        int =>' digit := x; end int
        char =>' letter := x; end char
    end case

This discriminating case statement ensures a com-
plete run-time check of which variant of a record
is in use, corresponding to the checks which can
be carried out by the compiler for all non-union

types.

3.3 Static Treatment of Type Unions

PASCAL has been deliberately designed to pro-
vide flexible type unions at the expense of secu-
rity (Wirth [1975]) ; however, a wise compiler should
be able to discern the secure programs by using the
following abstract interpretation of these programs:

Record values will be abstractly represented
by their tag fields. We will consider a program with
a single record type with variants identified by a
single tag, (the generalization to nested variants
and numerous record types is straightforward). The
tag is of enumerated type T which is a finite set
of discrete values. This set is augmented by a null
value which represents the non-initialized value.
Since at the same program point, but at two diffe-
rent moments of program execution, two different
values may be assumed by a tag field of a record
variable, a static summary of the potential program executions must consider a set of values for tag fields. (More generally, this is the case for variables of enumerated type.) Thus the abstract values of the tag will be chosen in $2^T$, the power-set of $T$, which is a finite complete lattice. Moreover, if the program contains simple variables of enumerated type $T$, it is convenient to take account of them in the program abstract interpretation process. Finally, if the program contains $m$ simple variables of type $T$ or record variables with tag of type $T$, our abstract data space is $(2^T \times \ldots \times 2^T)$ $m$ times. Since this space is a complete finite lattice, the abstract execution of programs can be performed at compile time.

<table>
<thead>
<tr>
<th>line</th>
<th>paul</th>
<th>mary</th>
<th>senior</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>(null)</td>
<td>(null)</td>
<td>(null)</td>
</tr>
<tr>
<td>(2)</td>
<td>(male)</td>
<td>(null)</td>
<td>(null)</td>
</tr>
<tr>
<td>(3)</td>
<td>the assignment to paul.age is ignored</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(4)</td>
<td>(male)</td>
<td>{female}</td>
<td>(null)</td>
</tr>
<tr>
<td>(5)</td>
<td>the assignment to mary.age is ignored.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Since the value of the test is statically unknown, this gives rise to two execution paths (6) and (8):</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(6)</td>
<td>(male)</td>
<td>{female}</td>
<td>(null)</td>
</tr>
<tr>
<td>(7)</td>
<td>(male)</td>
<td>{female}</td>
<td>(male)</td>
</tr>
<tr>
<td>(8)</td>
<td>(male)</td>
<td>{female}</td>
<td>(null)</td>
</tr>
<tr>
<td>(9)</td>
<td>(male)</td>
<td>{female}</td>
<td>{female}</td>
</tr>
<tr>
<td>(10)</td>
<td>The two execution paths are joined together.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The abstract interpretation of a test \((A = B)\) in a context where \(A\) and \(B\) are variables which may assume set of values \(S_A\) and \(S_B\) delivers a context where \(A\) and \(B\) may assume the set of values \(S_A \cap S_B\) on the true path. [Thus in (1) we get \(Paul = Senior\)]

\[
\begin{align*}
&\{} \text{Male} \in \{} \text{Male, Female} = \{\text{Male}\}. \\
\text{The context de-}
&\end{align*}
\]

1.1 Static variants to describe classes of data which are different but yet closely related. For example, Men and Women may be described as Persons depending on their sex, thus

\[
\text{EUCLID authorizes:} \quad \\
\text{type Person (Sex = (Male, Female)) = ...}
\]
3. Realization of implicit type transfer functions.  
EUCLID in recognition of the fact that controlled breaches of the type system are sometimes necessary, provides unchecked type conversions.

```pascal
i := unsigned-int << character('H')
```

assigns to `i` the internal code of the character 'H'. We have seen how a PASCAL compiler might report this fact to the user.

Finally, it is clear that PASCAL provides flexibility at the expense of security. We have shown that a compiler may report to the user which constructs have been used in either secure or insecure ways. The results of this static treatment of programs might also be useful in code generation. Thus we get a sophisticated compiler for a simple language. It is obvious then, that the programs will not be very readable, since the programmer has no preestablished constructs for expressing his intentions. However some simple intentions of the programmer which can be simply caught by compilers may necessitate rich and not necessarily easy to understand language constructs. This is the case in our next example concerning dynamic allocation of records.

4. Disjoint Collections of Linked Records

4.1 Collections in EUCLID

Suppose in PASCAL we have to represent two sets of records (of type `R`). We can use two arrays:

```pascal
var S1, S2 = array[1..n] of R;
```

With such a declaration, the PASCAL compiler knows that the sets `S1` and `S2` are disjoint, that is to say any modification of `S1` has no side effect on `S2` and vice-versa. Suppose that `n`, the maximal cardinality of the two sets is not known, we will use dynamically linked linear lists:

```pascal
type list = ^ elem;
        elem = record
            next : list;
            val : R;
            end;
```

```pascal
var S1, S2 : list;
```

This time, the readers of the program (e.g. PASCAL compilers) have to suppose that the sets `S1` and `S2` may share elements and it is now necessary to scan all the program to state the contrary.

In LIS[1974] one can specify that two pointers never refer to the same record:

```pascal
DS1 : domain of elem;
DS2 : domain of elem;
```

specify that `DS1` and `DS2` will be sets of disjoint dynamic variables. Now, if `S1` and `S2` are pointers into different domains:

```pascal
S1 : ^ DS1;
S2 : ^ DS2;
```

they point to different records of the same type. Unfortunately the confusion between a pointer to the first element of the linked structure, and the list is valid only in the programmer's intellect. `S1` and `S2` point to different records of type `elem`, which themselves may point to the same record. Thus the idea of domains has to be recursively applied in order to specify that elements of domain `DS1` point only to elements of `DS1`:

```pascal
DS1 : domain of elem1;
type elem1 = record
    next : ^ DS1;
    val : R;
    end;
```

and that elements of `DS2` can point only to elements of `DS2`:

```pascal
DS2 : domain of elem2;
type elem2 = record
    next : ^ DS2;
    val : R;
    end;
```

Since we want to guarantee that two pointers into different domains can never refer to the same variable we have to consider that `^DS1` and `^DS2` are different types of pointers. The trouble is now that `elem1` and `elem2` are different types, so that we have to write twice the algorithms (insertion, search, deletion ...) which handle the two similar lists `S1` and `S2`.

EUCLID[1976] is more flexible and authorizes types to be parameterized. Thus we will describe the types of lists `S1` and `S2` once, as depending on the domain (called collection in EUCLID) to which
they belong.

The type `elem` is parameterized by the name `C` of the collection to which elements of type `elem` point. This collection `C` is a collection of records (of type `elem`) pointing to `C`:

```pascal
type elem(C : collection of elem(C)) =
  record
    var next : *C
    var val : R
  end record
var DS1 : collection of elem(DS1)
var S1 : *DS1
var DS2 : collection of elem(DS2)
var S2 : *DS2
```

Now the operations on lists `S1` and `S2` can be described once, it just suffices to pass the name of the collection `DS1` or `DS2` to which they refer as a parameter:

```pascal
insert(DS1, S1, r)
```

will insert the record `r` in list `S1` which belongs to collection `DS1`. Now we have to declare the type of the formal parameter `DS` corresponding to the possible actual parameters `DS1` and `DS2`:

```pascal
procedure insert(DS : collection of elem(DS),
    var S : DS, val : R)
```

It is clear that `DS`, `DS1`, and `DS2` are just formal (or actual but different) collections of the same type. To make conspicuous that different collections will have the same type, we now want to give the name "listsupport" to the type of the collections supporting linked linear lists:

```pascal
type listsupport = collection of elem(?)
```

Since the type of a collection such as `DS1` depends on its name `DS1`, the type of the collection must be parameterized by that name:

```pascal
type listsupport(DS : ?) = collection of elem(DS)
```

A declaration such as:

```pascal
var DS1 = listsupport(DS1)
```

means that `DS1` is a collection of elements pointing to `DS1`. However, the above declaration of listsupport is incomplete since `DS` is a collection of type "listsupport":

```pascal
type listsupport(DS : listsupport(?)) =
```

Since we have entered a recursive question (each use of `listsupport` in the definition of `listsupport` must be provided by an actual parameter) we have to solve it by some language convention:

```pascal
type listsupport(DS : listsupport(parameter)) =
    collection of elem(DS)
```

The keyword `parameter` indicates that a shorthand has been used, the actual parameter will be provided later.

Since we succeeded in defining what is the type of collection supporting lists we now want to replace the definitions of this type by the name of that type, in particular in the definition of `elem`, to indicate that records of type `elem` point to collections of type `listsupport`. We get:

```pascal
type listsupport(DS : listsupport(parameter)) =
  forward

type elem(C : listsupport(parameter)) =
  record
    var next : *C
    var val : R
  end record

type listsupport = collection of elem(DS)
var DS1 : listsupport(DS1); S1 : *DS1
var DS2 : listsupport(DS2); S2 : *DS2
```

which is precise but somewhat overcomplicated when compared with the PASCAL declarations:

```pascal
type list = *elem;
elem = record
  next : list;
  val : R
end;
var S1, S2 : list;
{S1 and S2 are disjoint linked linear lists}.
```

Apart from the difficulty of coping with a new linguistic notion, the EUCLID approach has the advantage of the precision. Since the compiler knows that `S1` and `S2` are disjoint lists, it can produce better code especially for register allocation.

Moreover the combination of collections and restricted variants in records may yield efficient memory allocation strategies. Suppose we have a record type `R` with two variants `Ra`, `Rb` of different
memory sizes say 1 and 3 words:

```plaintext
Type Rtype = (Ra, Rb)
Type R (tag : Rtype) = record
  case tag in
    Ra = ... end Ra
    Rb = ... end Rb
  end case; end record
```

We have the following alternatives for memory allocation of collections of R:

- var C1 : collection of R(Ra)
- var C2 : collection of R(Rb)
- var C3 : collection of R(unknown)

(the type of records of collection C3 is unknown (it may be R(Ra) or R(Rb)). The type of a record will not change once allocated).

- var C4 : collection of R(any)

(The records of collection C4 can change from one variant to another during execution, by assigning values of different variants to the records).

The main defect of collections is that the number of collections is determined at compile time. Thus we cannot declare an array of disjoint linear lists:

Although of quite limited expressive power the notion of collection in EUCLID may appear somewhat difficult to understand. However its usefulness to compilers seems undeniable and we may in PASCAL let the compiler discover the collections.

### 4.2 Compiler Discovery of Disjoint Collections

We will represent a collection by the set of pointer variables which point within that collection.

**Example:**

![Diagram showing collections C1 and C2 with pointers V, W, X, Y, Z]

Collection C1 will be denoted (V, W), collection C2 will be denoted (X, Y, Z). We will try to partition the pointer variables of a program into disjoint collections. However in opposition to EUCLID, we will not try to find global collections but local ones. Thus the local invariants we will try to compute at each program point will be restricted to be of the form:

- \( (V, W \text{ are pointers to the same collection}) \)
- \( (X, Y, Z \text{ are pointers to the same collection}) \)

which we will denote:

- \( (V, W / X, Y, Z) \)

We now have to define the conjunction \( \overline{u} \) of such predicates (i.e. the union of sets of collections) for example:

\[
\{A,B,C / D,E\} \overline{\cup} \{F,A,G / H\} = \{A,B,C,F,G / D,E / H\}
\]

If on one hand A may point to a record referenced by B and C, or, on the other hand A may point to a record referenced by F and G, it is clear that A, B, C, F and G may point on the same record.

The instructions of the program provide useful information. After the instructions:
The following PASCAL procedure is supposed to do the job:

```
procedure copy (S1 : list; var S2 : list);
    var C1, C2, L : list;
begin
    {P0}
    C1 := S1; S2 := nil; L := nil;
    {P1}
    while C1 <> nil do
        begin
        {P2}
        new(C2); C2+.val := C1+.val; C2+.next := nil;
        {P3}
        if L = nil then
            {P4}
            S2 := C2
            {P5}
        else
            {P6}
            L+.next := C2 {P7};
            {P8}
            L := C2; C1 := C1+.next;
            {P9}
        end
    end
end

According to our abstract interpretation of the basic constructs of the language we can now establish the following system of equations:

(1) P1 = extract(L, extract(S2, extract(C1, P0) \ {c1, s1}))

(2) P2 = P1 \ P9
    (Since the test (C1 <> nil) gives us no information on collections when true)

(3) P3 = extract(C2, P2)
    (The assignment of non-pointer values and a deep modification in the structure pointed to by C2 are ignored)

(4) P4 = extract(L, P3)

(5) P5 = extract(S2, P4) \ {S2, C2}

(6) P6 = P3
    (since we ignore the fact that L <> nil)

(7) P7 = P5 \ {L, C2}

(8) P8 = P5 \ P7
(9) \( P_9 = \text{extract}(L,P_8) \cup \{L,C_2\} \)
(The statement \( C_1 := C_1 \text{next leaves} \ C_1 \) in the
same collection)

(10) \( P_{10} = \text{extract}(C_1,P_1 \cup P_9) \)

Since the theoretical conditions which ensure
that the above system of equations has a solution
are verified (Cousot [1978]) we can compute the least
fixpoint using a finite Kleene's sequence.
We start with the most disadvantageous initial pre-
dicate \( P_0 \), where on the one hand the parameters
\( \{S_1,S_2\} \) and on the other hand the local variables
\( \{C_1,C_2,L\} \) are supposed to be in the same collection:

\[
\begin{align*}
P_0 &= \{S_1,S_2 \cup C_1,C_2,L\} \quad P_1 = L, \forall i \in \{1,10\} \\
(1) &= P_1 = \text{extract}(L,\text{extract}(S_2,\text{extract}(C_1,P_0) \\
&\quad \cup \{C_1,S_1\})) \\
&= \text{extract}(L,\text{extract}(S_2,\{S_1,S_2/C_1,C_2,L\} \\
&\quad \cup \{C_1,S_2\})) \\
&= \text{extract}(L,\text{extract}(S_2,\{S_1,S_2/C_1,C_2,L\} \\
&\quad \cup \{C_1,S_2\})) \\
&= \text{extract}(L,\{S_1,S_2/C_1,C_2,L\}) \\
\end{align*}
\]

\[
\begin{align*}
P_1 &= \{S_1,C_1/S_2/C_2,L\} \\
(2) &= P_2 = P_1 \cup P_8 = P_1 \cup \bot = P_1 \\
(3) &= P_3 = \text{extract}(C_2,P_2) = \{S_1,C_1/S_2/C_2,L\} \\
(4) &= P_4 = \text{extract}(L,P_3) = \{S_1,C_1/S_2/C_2,L\} \\
(5) &= P_5 = \text{extract}(S_2,P_4) \cup \{S_2,C_2\} \\
&= \{S_1,C_1/S_2/C_2,L\} \cup \{S_2,C_2\} \\
\end{align*}
\]

\[
\begin{align*}
P_5 &= \{S_1,C_1/S_2/C_2,L\} \\
(6) &= P_6 = P_3 = \{S_1,C_1/S_2/C_2,L\} \\
(7) &= P_7 = P_6 \cup \{L,C_2\} \\
&= \{S_1,C_1/S_2/C_2,L\} \cup \{L,C_2\} \\
&= \{S_1,C_1/S_2/C_2,L\} \\
(8) &= P_8 = P_5 \cup P_7 \\
&= \{S_1,C_1/S_2/C_2,L\} \cup \{S_1,C_1/S_2/C_2,L\} \\
&= \{S_1,C_1/S_2/C_2,L\} \\
(9) &= P_9 = \text{extract}(L,P_8) \cup \{L,C_2\} \\
* \quad P_9 &= \{S_1,C_1/S_2/C_2,L\} \\
\end{align*}
\]

We go on cycling in the while-loop until the in-
viant \( P_0, \ldots, P_{10} \) have stabilized:

\[
\begin{align*}
(2) &= P_2 = P_1 \cup P_9 \\
&= \{S_1,C_1/S_2/C_2,L\} \cup \{S_1,C_1/S_2,C_2,L\} \\
* \quad P_2 &= \{S_1,C_1/S_2,C_2,L\} \\
(3) &= P_3 = \text{extract}(C_2,P_2) = \{S_1,C_1/S_2,L/C_2\} \\
* \quad P_4 &= \text{extract}(L,P_3) = \{S_1,L'/S_2/L/C_2\} \\
\end{align*}
\]

We come back for \( P_4 \) with the value of the previous
pass, so we stop on that path.

\[
\begin{align*}
(6) &= P_6 = P_3 = \{S_1,C_1/S_2,L/C_2\} \\
(7) &= P_7 = P_6 \cup \{L,C_2\} \\
* \quad P_7 &= \{S_1,C_1/S_2,L/C_2\} \\
(8) &= P_8 = P_5 \cup P_7 \\
&= \{S_1,C_1/S_2,C_2,L\} \cup \{S_1,C_1/S_2,C_2,L\} \\
* \quad P_8 &= \{S_1,C_1/S_2,C_2,L\} \\
\end{align*}
\]

Same value as above, stop on that path. It remains
only the path out of the loop :

\[
(10) \Rightarrow P_{10} = \text{extract}(C_1,P_1 \cup P_9) \\
* \quad \text{extract}(C_1, \{S_1,C_1/S_2,C_2/L\} \\
&\quad \cup \{S_1,C_1/S_2,C_2,L\}) \\
* \quad P_{10} = \{C_1,S_1/S_2,C_2,L\} \\
\]

The final results are marked by a star (*). The
main result is that although \( S_1 \) and \( S_2 \) may share
records on entry of the procedure "copy" :

\[
P_0 = \{S_1,S_2/C_1,C_2,L\} \\
\]

it is guaranteed that this is not the case on exit
of the procedure :

\[
P_{10} = \{C_1,S_1/S_2,C_2,L\} \\
\]

4.3 Remarks

a. This abstract interpretation of programs may be
refined as in EUCLID : when records have variants
one can associate with each collection the set
of tags of all records in the collection. This
in fact will be the main application of our de-
velopments of paragraph 3. We will be more fle-
 exible than the "one of" or "any" of EUCLID, and
authorize collections with say two variants
\( \{A,B\} \) among three possibilities \( \{A,B,C\} \). Other-
wise stated we reason on the following type hie-
rrarchy :

\[
\begin{align*}
\{A,B,C\} &= \tau \\
\{A,B\} &= \alpha \\
\{A,C\} &= \beta \\
\{B,C\} &= \gamma \\
\{A\} &= \psi \\
\{B\} &= \chi \\
\{C\} &= \omega \\
\{\}\ &= \bot
\end{align*}
\]
whereas EUCLID uses a simplified type inclusion scheme:

```
{A, B, C} = \tau
\downarrow
\{A\} \quad \{B\} \quad \{C\}
\downarrow
\{\} = 1
```

b. Besides and in opposition with EUCLID the collections are defined as local invariants. Very precise and detailed information can be gathered whereas the EUCLID programmer would have to globally specify the union of such information.

We now come to an example where a cooperation between the programmer and the compiler is absolutely necessary for secure and cheap use of type unions, that is to say a case when the compiler has definite disadvantages over the programmer.

5. Integer Subrange Type

- An optimizing compiler will be able to limit the number of objects which are supposed to have been modified by side-effects when assigning to objects designated by pointers, (useful in register allocation).
- Run-time tests may be inserted before a statement:
  ```
  dispose(X);
  ```
  to verify that no variable in the collection of `X` may access the record which `X` points to,
- The garbage collector may be called when all variables in a collection are "dead" (i.e. are not used before being assigned to),
- etc...

The simple abstract interpretation of programs illustrated here may be further investigated to recognize that data structures are used in stylized ways. (Boom[1974], Karr[1975]).

c. It is fairly however to say that EUCLID compilers may use the same techniques to locally refine the collections provided by the programmer. The advantage of EUCLID is then that when the programmer has declared his intentions (or better part of intentions since the expressive power of collections is limited), he is forced to conform to his declarations. For example he will not be able to use the same pointer variable to traverse two lists which are built in different types.

The automatic discovery of collections. The advantage however must be counterbalanced by the fact that parameterized collections (which are necessary with recursive data structures) may become inflexible and difficult to use.

5. Integer Subrange Type

- type index = 0..8

is used to specify that variables of type index will possess the properties of variables of the base integer type, under the restriction that its value remains within the specified range. (Wirth[1975]). In Cousot[1975], we developed a technique to have the compiler discover the subrange of integer variables. Let us take an obvious example:

```
1 := 1;
[p1]
while i <= 1000 do
  [p2]
i := i + 1 [p3];
[p4]
```

Let us denote by \([a, b]\) the predicate \(a \leq i \leq b\).

The system of equations corresponding to our example is:

1. \(P_1 = [1, 1]\)
2. \(P_2 = (P_1 \cup P_3) \cap [\inf, 1000]\)
3. \(P_3 = P_2 + [1, 1]\)
4. \(P_4 = (P_1 \cup P_3) \cap [1001, \sup]\)

where the is defined by \([a, b] + [c, d] = [a+c, b+d]\), and \(\cup\) and \(\cap\) are union and intersection of intervals. Suppose we know the solution to that system, i.e.

```
P_1 = [1, 1], P_2 = [1, 1000], P_3 = [2, 1001], P_4 = [1001, 1001]
```

It is obvious to let the compiler verify that
(1) \[ P_1 = [1, 1] \]

(2) \[ P_2 = (P_1 \cup P_3) \cap [\text{null}, 1000] \]
\[ = ([1, 1] \cup [2, 1001]) \cap [\text{null}, 1000] \]
\[ = ([1, 1001] \cap [\text{null}, 1000]) \]
\[ = [1, 1000] \]

(3) \[ P_3 = P_2 + [1, 1] \]
\[ = [1, 1000] + [1, 1] \]
\[ = [1+1, 1000+1] \]
\[ = [2, 1001] \]

(4) \[ P_4 = (P_1 \cup P_3) \cap [1000, \infty] \]
\[ = ([1, 1] \cup [2, 1001]) \cap [1000, \infty] \]
\[ = [1, 1001] \cap [1000, \infty] \]
\[ = [1000, 1001] \]

If we want the compiler to discover this fixpoint, we may try to solve the equations by algebraic manipulations (Cheatham and Townley[1976]) which may be quite inextricable. The other way is to use Kleene's sequence, but the trouble is that our abstract data space is an infinite lattice, and we may have infinite sequences. Since compilers must work even for programs which may turn out to loop, the only way to cope with the undecidable problem is to accept approximative answers. For example in the program:

```
for i := 1 to 100 do
begin
  n := 1;

  while n < 1 do
    if even(n) then n := n/2
    else n := 3 * n + 1;

  write (1)
end;
```

Cousot[1975] will discover an approximate range for \(n\) which will be \([1, \infty]\). However, if the actual range of \(n\) were known by the programmer and if the programmer could tell this to the compiler, then a verification would be simpler (in most cases but not on this difficult example).

We can now state our main objection against subrange types in PASCAL: the fact that range assertions must be given globally in the declaration prevents the programmer from giving the solution of the system of equations to the compiler. The programmer can only give an approximation of the solution, which is usually insufficient for the compiler to discover local subranges. To make it clear, instead of \(P_1, P_2, P_3, P_4\) the programmer is only able to declare \(\text{var i : 1..1001}\) that is to say that \(P_1 \cup P_2 \cup P_3 \cup P_4 \subseteq [1, 1001]\) which adds an inequation to the system of equations but does not provide its solution. We then consider integer subrange types as union types since the global declaration must be the union of all local subranges. Thus, if we declare:

```
\text{var i : 0..2;}
```

we really want to say that the type of \(i\) at each program point is one of the following alternatives:

```
0.0
0.1
1.1
1.2
2.2
```

We then understand a criticism by Habermann[1973] that subranges are not types, since a global subrange type definition does not determine

of operators that are applicable to variables of that type.

For example, let \(f\) be a function with one formal parameter of type \(2..10\) and \(i\) a variable globally declared of type \(0..5\). The variable \(i\) may be used at program point \(p\) in the expression \(f(1)\) provided that \(i\) may be united to the subrange \(2..10\). Dynamically the local type of \(i\) at program point \(p\) is \(\overline{3},\overline{1}\), which is simply derived from the value \(\overline{1}\) of the variable \(i\). In the expression \(f(i)\), \(i\) must be coerced from the type \(\overline{3},\overline{1}\) to the type \(2..10\). This is safe when \(2 \leq \overline{1}\) and \(\overline{1} \leq 10\). Staticly this signifies that the subrange of \(i\) at program point \(p\) must be a subrange of \(2..5\). This subrange of \(2..5\) cannot be locally specified in PASCAL.

This understanding of subranges leads us to the conclusion that integer subranges should be specified locally. Moreover, and in opposition with our previous examples we cannot expect the compiler to be
able to discover these local subrange properties. It is then essential that programmers provide them, by means of assertions or as previously by means of conformity clauses so that we would write in the spirit of ALGOL 68 (Maertens[1975]):

\[
\begin{align*}
    i &:= 1; \\
    \text{while case } i \text{ in } &:
\end{align*}
\]

However, when the type system is infinite (for example,

results will be obtained by type checking or type discovery as long as finite type systems are considered. The main difference between these approaches is the one between security (at the expense of flexibility) or simplicity (at the expense of precision, and of the possibility that compiler warnings be ignored).
8. Bibliography

ALGOL 60[1963]. Naur P. (Ed.)

ALGOL 68[1975]. Van Wijngaarden A. et al. (Eds.).

Boo\textsuperscript{m}[1974].
"Optimization Analysis of Programs in Languages with Pointer Variables".

Cheatham and Townley[1976].


Habermann[1973].

Hoare and Wirth[1973].

Kerr[1975].
Sintzoff[1975].
"Verifications d’assertions pour les fonctions utilisables comme valeurs et affectant des variables extérieures", in "Proving and Improving programs", ed. G. Huet and G. Kahn, IRIN.

Tarski[1955].

Wegbreit[1975].

Wirth[1975].