ACM
Association for Computing Machinery

SIGPLAN Notices
Volume 12, Number 3 March 1977

OPERATING SYSTEMS Review
Volume 11, Number 2 April 1977

SOFTWARE ENGINEERING Notes
Volume 2, Number 2 March 1977

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 68 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 staticly 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 staticly enfor-

* Attaché de Recherche au C.N.R.S., Laboratoire Associé N° 7.

** This work was supported by IRIA-SESORI under grants 75-035 and 76-160.

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 software products, the language designer may reply upon :

- The design of a refined and safe type system, which necessitates linguistic constructs which propagate strong type properties. The rules of the language 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 improved 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 futurustic and mysterious automatic program verifiers.

We illustrate the two approaches by means of examples; The compiler techniques we propose for the static analysis of programs have a degree of sophistication comparable to program optimization techniques rather than program verification techniques, Cousot [1976]. It is shown that the language design approach and the compiler design approach are strongly related since both need a refinement of the type notion. They differ by the fact that one needs a type checker whereas the other uses a type discoverer, but we show the close connexion between type checking and discovery.

We show that strong type enforcement or discovery may be equivalent (e.g. nil references, type unions, collections of non intermixing pointers). This is not the case for infinite type systems (e.g. integer ranges), which are not compile time checkable. In such a case type discovery is really needed and can be facilitated by appropriate syntactic constructs. Finally we propose a means by which language designers can establish a balance between the 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 pointers are the facts that they can lead to serious type violations (PL/1) and that they may be left dangling. One can take care of these objections, by guaranteeing 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 remain 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 meaningful. 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 :

- the type of pointers to a record type
- the subtype of non-nil pointers to that record type
- the subtype of nil pointers to that record type (which happens to have only one value)

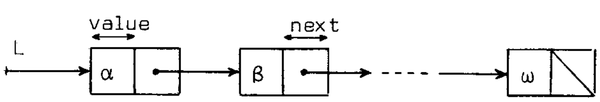
The rule is that dereferencing can be applied only to pointers of non-nil subtype. Since this rule must be enforceable by the programming system the language designer has three solutions :

- Run-time checks (these checks are usually very cheap for pointers when using the hardware memory protection facilities. However for system implementation languages generating code in master-mode this hardware detection is not always utilizable. Moreover, for more complicated examples such as array subscripting these run-time checks are very 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 :



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

```
(1)    pt := L; b := true;
(2)    {P1}
(3)    while (pt <> nil) and b do
(4)      ({P2}
(5)       if pt↑.value = n then
(6)          b := false
(7)          {P3}
(8)       else
(9)          {P4}
(10)         pt := pt↑.next;
(11)      {P5});
```

The above piece of program is correct with regard to accesses to records via pointers, since pt is not nil when dereferenced at lines (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 :

(1) P1 = (pt = L) and (b = true)

(2) P2 = (P1 or P5) and ((pt <> nil) and b)

(3) P3 = (P2 and (pt↑.value = n)) and (b = false)

(4) P4 = P2 and (pt↑.value <> n)

(5) P5 = P3 or (∃ pt' | $S^{pt'}_{pt}$(P4) and pt = pt'↑.next)

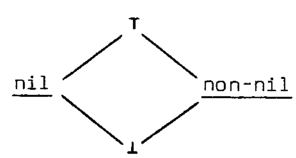
(Equation (5) has been deliberately oversimplified, see Dembinski and Schwartz[1976]).

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 :

pt = nil, pt <> nil, (pt = nil) or (pt <> nil)

respectively denoted by nil, non-nil, ⊤ . These predicates form a complete lattice whose HASSE's diagram is :

L =



Where ⊥ 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 :

(1') P1 = ⊤

(2') P2 = (P1 or P5) and non-nil

(3') P3 = P2

(4') P4 = P2

(5') P5 = P4 or ⊤

(In equation (1) we consider (pt = L) since L may be an empty or non-empty linear list, we get (pt = nil) or (pt <> nil) denoted $\top$, 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 :

$$<P1, P2, P3, P4, P5> = F(<P1, P2, P3, P4, P5>)$$

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 (Tarski[1955]). Moreover, since F is a complete order-preserving morphism from the complete lattice L^5 in itself, this least fixpoint can be defined as the limit of Kleene's sequence, Kleene [1952] :

$$\begin{aligned}
\lambda_0 &= <\bot, \bot, \bot, \bot, \bot> \\
\lambda_1 &= F(\lambda_0) \\
&= <\top, (\bot \text{ or } \bot) \text{ and non-nil}, \bot, \bot, (\bot \text{ or } \top)> \\
&= <\top, \bot, \bot, \bot, \top> \\
\lambda_2 &= F(\lambda_1) \\
&= <\top, (\top \text{ or } \top) \text{ and non-nil}, \bot, \bot, (\bot \text{ or } \top)> \\
&= <\top, \text{non-nil}, \bot, \bot, \top> \\
\lambda_3 &= F(\lambda_2) \\
&= <\top, (\top \text{ or } \top) \text{ and non-nil}, \text{non-nil}, \text{non-nil}, (\bot \text{ or } \top)> \\
&= <\top, \text{non-nil}, \text{non-nil}, \text{non-nil}, \top> \\
\lambda_4 &= F(\lambda_3) \\
&= <\top, (\top \text{ or } \top) \text{ and non-nil}, \text{non-nil}, \text{non-nil}, (\text{non-nil or } \top)> \\
&= <\top, \text{non-nil}, \text{non-nil}, \text{non-nil}, \top> \\
&= \lambda_3
\end{aligned}$$

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 P2 = P3 = P4 = non-nil, which according to our interpretation means that pt is not nil at lines (4), (7) and (9) of our program, which implies that the accesses of records through pt at lines (5) and (10) are staticly shown to be correct. With regard to the value of P1 and P5, its interpretation is that pt may be 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 :

$$\begin{aligned}
P1 &= \top, \ (P_i = \bot, \ i \in [2, 5]) \\
P2 &= (P1 \text{ or } P5) \text{ and non-nil} \\
&= (\top \text{ or } \bot) \text{ and non-nil} \\
&= \text{non-nil} \\
P3 &= P2 \\
&= \text{non-nil} \\
P4 &= P2 \\
&= \text{non-nil} \\
P5 &= P3 \text{ or } \top \\
&= \text{non-nil or } \top \\
&= \top \\
P2 &= (P1 \text{ or } P5) \text{ and non-nil} \\
&= (\top \text{ or } \top) \text{ and non-nil} \\
&= \text{non-nil, same as above, stop.}
\end{aligned}$$

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 interpretor 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 ref real (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 nil-names (this is achieved by not providing a representation for the nil symbol), so that any name refers to a value. The type void is used to represent nil-names. Finally the type of nil and non-nil pointers is the union of the previous ones.

For example we can write a construction like

```
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 :

```
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            ==> false
     esac
do skip od;
```

This program is safe, since in ALGOL 68 the non-safe coercion of pt from mode union (ref cell, void) to mode ref cell has to be made explicit by a conformity case construct. The idea is therefore to force the programmer to explicitly perform the run-time tests, which in this example is dictated anyway by the logic of the problem (the rewritten version admittedly looks a bit cumbersome, but more convenient ways of expressing such a flow of control may be exhibited (Dijkstra[1975])).

2.3 *Remarks*

It is remarkable that both approaches necessitate the same secure type system, yet they differ in the choices of making it available or not to the programmer.

The refined type system considers the pointer type as the union of two sybtypes : pure (non-nil) pointers and dummy (nil) pointers. Type safety is guaranteed by requiring strong typing : the type of a value determines which operations may be meaningfully applied to it.

In both cases the type correctness has to be verified or established by the compiler. For that purpose an (often implicit) system of equations is used. In one case the solution to that system of equations has to be found by the compiler, in the other case the compiler simply verifies that the solution supplied by the programmer (by means of 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.

3. *Variants of Record Structures*

3.1 *Unsafe Type Unions in PASCAL*

In ALGOL 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 consisting of several variants, optionally discriminated by a tag field.

Example :

```
type mode = (int, char);
type charint =
  record
       case tag : mode of
           int : (i : integer);
           char : (c : character)
  end;
var digit, letter, alphanum : charint;
```

In a program containing these declarations, the occurrence of a variable designator alphanum.c is only valid, if at this point that variable is of type character. It is so, (if and) only if alphanum.tag = char. However this is not staticly verified by the PASCAL compilers for the following reasons :

- The tag field of a variant record definition is optional, and may exist only in the programmer's mind.
- When present, the tag field may be assigned, thus allowing to realize implicit type transfer functions. For instance, a variable of type character :

  ```
  alphanum.tag := char;
  alphanum.c := 'H';
  ```

  may be interpreted as being of type integer for the purpose of printing the internal 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);)
```

3.2 *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 [1976] which handles variant records in a type-safe, ALGOL 68-like manner. Since EUCLID allows 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 constant :

```
      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 discrimating case statement ensures a complete 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 provide flexible type unions at the expense of security (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 different 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 powerset 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.

Example :

```
  type person =
     record
        ...
        case sex : (male, female) of
        ...
     end;
     var paul, mary, senior : person;
(1)
     paul.sex := male;
(2)
     paul.age := ...;
(3)
     mary.sex := female;
(4)
     mary.age := ...;
(5)
     if paul.age ≥ mary.age then
(6)
        senior := paul;
(7)
     else
(8)
        senior := mary;
(9)
     end;
(10)
```

The symbolic execution of that piece of program would be :

line	paul	mary	senior
(1)	{null}	{null}	{null}
(2)	{male}	{null}	{null}
(3)	the assignment to paul.age is ignored		
(4)	{male}	{female}	{null}
(5)	the assignment to mary.age is ignored. Since the value of the test is staticly unknown, this gives rise to two execution paths (6) and (8) :		
(6)	{male}	{female}	{null}
(7)	{male}	{female}	{male}
(8)	{male}	{female}	{null}
(9)	{male}	{female}	{female}
(10)	The two execution paths are joined together :		
	{male}u{male} = {male}	{female}u{female} = {female}	{male} u {female} = {male, female}

Note that at line (10) it is clear that "senior" may have tag values "male" or "female". However, we don't appreciate the fact that :

senior.sex = if paul.age ≥ mary.age then male else female fi

but neither do ALGOL 68 nor EUCLID. With these languages it is evident that in some cases the programmer knows perfectly well which alternative of a union type is used, but is unable to exploit this knowledge, since he must use a discriminating case statement. This same limitation arises with our static treatment of programs, more powerful schemes exist (Sintzoff[1975]).

Finally, in the static treatment of programs useful information will be gathered from case statements, and if statements, used as ALGOL 68 conformity tests.

Example :

{Paul = {male} ; Mary = {Female} ; Senior = {Male, Female}}

```
if Senior.Sex = Paul.sex then
   ... (1) ...
else
   ... (2) ...
fi
```

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 = {Male} ∩ {Male, Female} = {Male}).The context delivered for the false path is :

$$A = \underline{\text{if}}\ (|S_A \cap S_B| = 1)\ \underline{\text{and}}\ \underline{\text{not}}\ (S_A \subset S_B)\ \underline{\text{then}}\ S_A - S_B\ \underline{\text{else}}\ S_A\ \underline{\text{fi}}$$

(Thus in (2) we get Paul : {male} and Senior = {Female}).

When this abstract interpretation of programs is terminated we can recognize secure programs by the following facts :

- There are no assignments to tag fields, other than for initialization(which is recognized by the fact that the tag value is changed from <u>null</u> to some value). We can also authorize useless tag assignments, i.e. those which assign to a tag without changing its value.
- The unsafe de-uniting coercions must be checked. This cannot occur when a record variable is assigned to another, since all record variables are considered to be of union types. (Note that such an assignment may indirectly modify a tag value, but this is safe). De-uniting coercions only occur when accessing a field in a record. This is safe only if the tag has been staticly established to be of correct value. Otherwise, a warning is reported to the user, and a run-time check inserted in the program.

3.4 *Flexibility Versus Security*

This compiler approach has the advantage of flexibility over the secure language approach. It is clear that all EUCLID programs translated into PASCAL will be recognized to be safe by this technique.

Following Wirth[1975] there appear to be three different motivations behind the desire for variants of record structures :

1. The need for heterogeneous structures, in two main cases :

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 EUCLID authorizes :

```
type Person (Sex = (Male, Female)) = ...
type Man = Person(Male)
type Woman = Person(Female)
```

In PASCAL however, variables of abstract type Man and Woman may be staticly recognized when their tag values never change.

1.2 Dynamic variants, to describe objects whose components depend on a possibly changing state. For example a car may be moving or stopped, thus EUCLID authorizes :

```
type Car (State : (moving, stopped, des-
          troyed)) = ...
var mycar : Car(any)
```

Since the actual parameter supplied for the tag is <u>any</u>, the variable can be changed from one variant to another during execution, by assigning values of different variants to the variable. However, no refinement is allowed, and no proper subset of the possible tag values can be used :

```
var mycar : Car({moving, stopped})
```

This fact may be discovered by a static analysis of the program, and might be useful in memory allocation.

2. Storage Sharing (Overlays). This implies the use of the same storage area (expressed in the language as "the same actual variable") for different purposes i.e. for representing different abstract variables whose lifetimes are disjoint (block structure is not incorporated in PASCAL). This is a typical case of free union, where no tag will be carried along to indicate the currently valid variant. This tag may be staticly simulated, provided that one ensures an appropriate setting of the tag upon assignment to fields of the variant. Unsafe assignments will be identified and therefore the mutation from one abstract variable to another may be reported to the user.

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, by means of type converters :

```
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 :

```
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 :

```
type list = ↑ elem;
     elem = record
                next : list;
                val : R;
            end;
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 ; the declarations

```
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 :

```
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 :

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

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

```
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 :

```
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 :

```
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 :

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

It is clear that DS, DS1, 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 :

```
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 :

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

A declaration such as :

```
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" :

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

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 :

```
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 type elem, to indicate that records of type elem point to collections of type listsupport. We get :

```
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 :

```
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 copeing 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 :

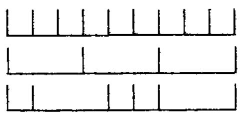
```
Type Rtype = (Ra, Rb)
Type R (tag : Rtype) = record
      case tag in
         Ra == ... end Ra
         Rb == ... end Rb
      end case; end record
```

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)

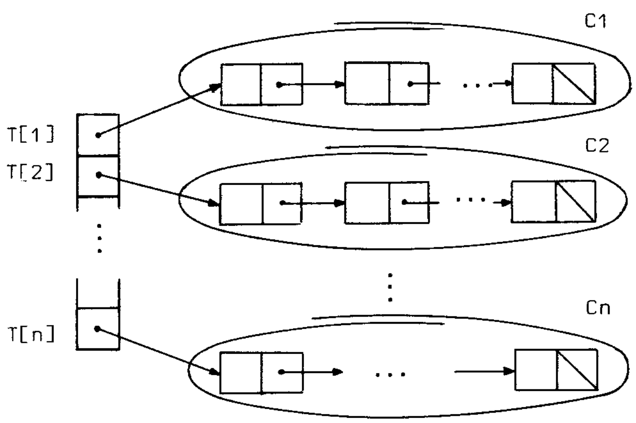


(the type of records of collection C3 is unknown (it may be R(Ra) or R(Rb)). The type of a record will not change once allocated).

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 :



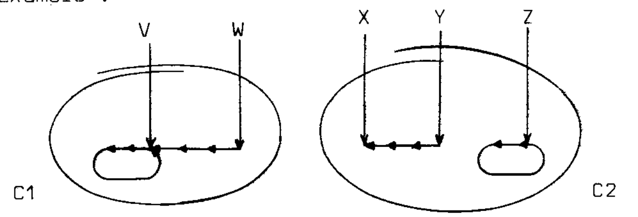
It has not been recognized that a globally declared collection is in fact the union of smaller collections which are valid at various program points (which would be useful to the compiler). A similar criticism is that the concept of collection cannot be used recursively, that is to say one cannot partition a collection into disjoint sub-collections.

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 :



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 are pointers to the same collection)

and

(X, Y, Z are pointers to the same collection)

which we will denote :

{V, W / X,Y,Z}

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

{A,B,C / D,E} $\bar{\cup}$ {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 :

```
X := nil;
X := Y; {where Y is known to be nil}
if X = nil then ...
new (X);
```

it is known that X will point to no record at all, or will be the only pointer to the newly allocated record. Thus we have isolated a collection (empty or consisting of a single record). With an input predicate

$$P1 = \{X_1, X_2, \ldots, X, \ldots, X_n \;/\; Y_1, \ldots, Y_n\}$$

the above instructions lead to an output predicate :

$$P2 = \underline{\text{extract}}(X, P1) = \{X \;/\; X_1, \ldots, X_n \;/\; Y_1, \ldots, Y_n\}$$

More generally, with an input predicate P1, a pointer assignement such as :

$$\underbrace{X\uparrow.\text{next} \ \ldots\ \uparrow.\text{next}}_{\text{optional}} := \underbrace{Y\uparrow.\text{next} \ \ldots\ \uparrow.\text{next}}_{\text{optional}}$$

may cause X and Y to indirectly point to a common record. Hence they are put in the same collection. The output predicate will be $P2 = P1 \mathbin{\bar{\cup}} \{X,Y\}$.

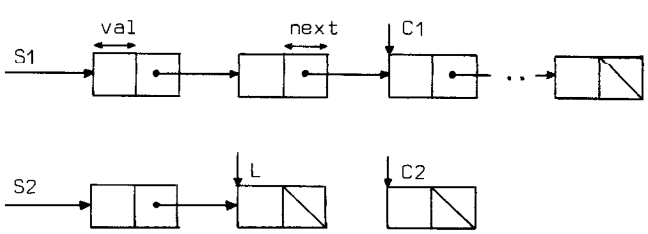
A sensible remark is that the value delivered by the right-hand side of the assignment may be nil, in which case this may cause a collection to be broken into two disjoint sub-collections. For simplicity, we ignore this fact, other than in the obvious case :

$$\{P1\} \quad X := \underbrace{Y\uparrow.\text{next} \ \ldots\ \uparrow.\text{next}}_{\text{optional}} \quad \{P2\} \quad ;$$

which will cause X to be disconnected from its collection and be connected to a record of the collection of Y. When X and Y are not the same variable, the output assertion P2 will be related to the input assertion P1 by :

$$P2 = \underline{\text{extract}}(X, P1) \mathbin{\bar{\cup}} \{X, Y\}$$

Now, we will give an example. We have chosen the copying of a linked linear list :



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
   {P10}
end;
```

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

(1) $P1 = \underline{\text{extract}}(L,\underline{\text{extract}}(S2,\underline{\text{extract}}(C1,P0) \mathbin{\bar{\cup}} \{C1,S1\}))$

(2) $P2 = P1 \mathbin{\bar{\cup}} P9$
   (Since the test (C1 <> nil) gives us no information on collections when true)

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

(4) $P4 = \underline{\text{extract}}(L,P3)$

(5) $P5 = \underline{\text{extract}}(S2, P4) \mathbin{\bar{\cup}} \{S2, C2\}$

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

(7) $P7 = P6 \mathbin{\bar{\cup}} \{L,C2\}$

(8) $P8 = P5 \mathbin{\bar{\cup}} P7$

(9) P9 = extract(L,P8) $\bar{\cup}$ {L,C2}

(The statement C1 := C1.next leaves C1 in the same collection)

(10) P10 = extract(C1,P1 $\bar{\cup}$ P9)

Since the theoretical conditions which ensure that the above system of equations has a solution are verified (Cousot[1976]) we can compute the least fixpoint using a finite Kleene's sequence.

We start with the most disadvantageous initial predicate P0, where on the one hand the parameters (S1,S2) and on the other hand the local variables (C1,C2,L) are supposed to be in the same collection :

* P0 = {S1,S2 / C1,C2,L} Pi = $\perp$, $\forall$i $\in$ [1,10]

(1) => P1 = extract(L,extract(S2,extract(C1,P0) $\bar{\cup}$ {C1,S1}))

= extract(L,extract(S2,{S1,S2/C1/C2,L} $\bar{\cup}$ {C1,S1}))

= extract(L,extract(S2,{S1,S2,C1/C2,L}))

= extract(L,{S1,C1/S2/C2,L})

* P1 = {S1,C1/S2/C2/L}

(2) => P2 = P1 $\bar{\cup}$ P9 = P1 $\bar{\cup}$ $\perp$ = P1

(3) => P3 = extract(C2,P2) = {S1,C1/S2/C2/L}

(4) => P4 = extract(L,P3) = {S1,C1/S2/C2/L}

(5) => P5 = extract(S2,P4) $\bar{\cup}$ {S2,C2}

= {S1,C1/S2/C2/L} $\bar{\cup}$ {S2,C2}

* P5 = {S1,C1/S2,C2/L}

(6) => P6 = P3 = {S1,C1/S2/C2/L}

(7) => P7 = P6 $\bar{\cup}$ {L,C2}

= {S1,C1/S2/C2/L} $\bar{\cup}$ {L,C2}

= {S1,C1/S2/C2,L}

(8) => P8 = P5 $\bar{\cup}$ P7

= {S1,C1/S2,C2/L} $\bar{\cup}$ {S1,C1/S2/C2,L}

= {S1,C1/S2,C2,L}

(9) => P9 = extract(L,P8) $\bar{\cup}$ {L,C2}

* P9 = {S1,C1/S2,C2,L}

We go on cycling in the while-loop until the invariant P0, ..., P10 have stabilized :

(2) => P2 = P1 $\bar{\cup}$ P9

= {S1,C1/S2/C2/L} $\bar{\cup}$ {S1,C1/S2,C2,L}

* P2 = {S1,C1/S2,C2,L}

*(3) => P3 = extract(C2,P2) = {S1,C1/S2,L/C2}

*(4) => P4 = extract(L,P3) = {S1,C1/S2/L/C2}

We come back for P4 with the value of the previous pass, so we stop on that path.

*(6) => P6 = P3 = {S1,C1/S2,L/C2}

(7) => P7 = P6 $\bar{\cup}$ {L,C2}

* P7 = {S1,C1/S2,L,C2}

(8) => P8 = P5 $\bar{\cup}$ P7

= {S1,C1/S2,C2/L} $\bar{\cup}$ {S1,C1/S2,L,C2}

* P8 = {S1,C1/S2,L,C2}

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

(10)=> P10 = extract(C1, P1 $\bar{\cup}$ P9)

= extract(C1, {S1,C1/S2/C2/L} $\bar{\cup}$ {S1,C1/S2,C2,L})

= extract(C1, {S1,C1/S2,C2,L})

* P10 = {C1/S1/S2,C2,L}

The final results are marked by a star (*). The main result is that although S1 and S2 may share records on entry of the procedure "copy" :

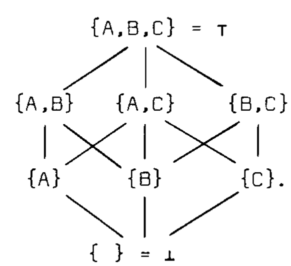
P0 = {S1,S2/C1,C2,L}

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

P10 = {C1/S1/S2,C2,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 developments of paragraph 3. We will be more flexible than the "one of" or "any" of EUCLID, and authorize collections with say two variants {A,B} among three possibilities {A,B,C}. Otherwise stated we reason on the following type hierarchy :



whereas EUCLID uses a simplified type inclusion scheme :

$$
\begin{tikzcd}[column sep=small]
 & \{A,B,C\} = \top \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\{A\} \arrow[dr, dash] & \{B\} \arrow[d, dash] & \{C\} \arrow[dl, dash] \\
 & \{\ \} = \bot &
\end{tikzcd}
$$

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. This localization of collections may have important consequences :

- 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 :

  <u>dispose</u>(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 we illustrated here may be further investigated to recognize that data structures are used in stylized ways. Boom[1974], Karr[1975].

c. It is fair 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 collections. On the contrary this may confuse 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.

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*

A subrange type such as :

<u>type</u> index = 0..9

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 :

```
i := 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) $P1 = [1,1]$

(2) $P2 = (P1 \cup P3) \cap [-\infty, 1000]$

(3) $P3 = P2 + [1, 1]$

(4) $P4 = (P1 \cup P3) \cap [1001, +\infty]$

where $+$ 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.

$P1 = [1,1]$, $P2 = [1, 1000]$, $P3 = [2, 1001]$, $P4 = [1001, 1001]$

It is obvious to let the compiler <u>verify</u> that this solution is a fixpoint of the system :

$$
\begin{aligned}
(1) \Longrightarrow P1 &= [1, 1] \\
(2) \Longrightarrow P2 &= (P1 \cup P3) \cap [-\infty, 1000] \\
&= ([1, 1] \cup [2, 1001]) \cap [-\infty, 1000] \\
&= ([1, 1001] \cap [-\infty, 1000]) \\
&= [1, 1000] \\
(3) \Longrightarrow P3 &= P2 + [1, 1] \\
&= [1, 1000] + [1, 1] \\
&= [1 + 1, 1000 + 1] \\
&= [2, 1001] \\
(4) \Longrightarrow P4 &= (P1 \cup P3) \cap [1001, +\infty] \\
&= ([1, 1] \cup [2, 1001]) \cap [1001, +\infty] \\
&= [1, 1001] \cap [1001, +\infty] \\
&= [1001, 1001]
\end{aligned}
$$

If on the contrary 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 :

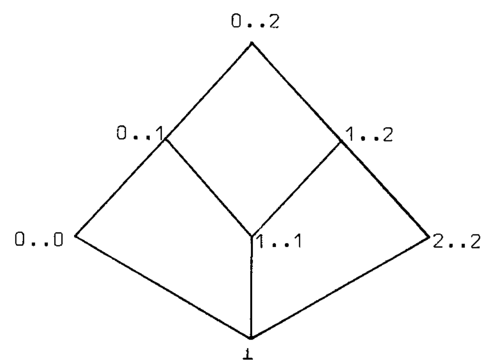
```
for i := 1 to 100 do
   begin
      n := i;
      while n <> 1 do
         if even(n) then n := n/2
                    else n := 3 * n + 1;
      write (i)
   end;
```

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 prevent 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 P1, P2, P3, P4 the programmer is only able to declare *var* i : 1..1001 that is to say that $P1 \cup P2 \cup P3 \cup P4 \subseteq [1, 1001]$ 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 :

var i : 0..2;

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



We then understand a criticism by Habermann[1973] that subranges are not types, since a global subrange type definition does not determine the set 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(i) provided that i may be united to the subrange 2..10. Dynamically the local type of i at program point p is $\bar{i}..\bar{i}$, which is simply derived from the value $\bar{i}$ of the variable i. In the expression f(i), i must be coerced from the type $\bar{i}..\bar{i}$ to the type 2..10. This is safe when $2 \le \bar{i}$ and $\bar{i} \le 10$. 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 (Meertens[1975]) :

```
i := 1;
while case i in
         (1..1000)p:(i := p + 1; true),
         out       :false
       esac
do skip od;
```

These constructs give the solution of the system of equations which the compiler has to solve for strong type checking. The redundancies (equations identically verified) can be eliminated. Moreover the PASCAL restriction that the bounds of ranges must be manifest constants is a definite advantage since this verification will involve no symbolic formula manipulations. Run-time tests will remain necessary in difficult cases, but their number will be decreased.

6. *Conclusion*

We illustrated the fact that unsecure data types (which do not guarantee all operations on values of that type to be meaningful) can be considered as the union of secure (sub) types. Examples of these were pointers, variants in records, records in collections, integer subranges.

A type-safe programming system must staticly determine which safe subtype of the union is used when checking correct use of operations on union typed objects. The language designer may achieve this goal by one of the following alternatives:

- Incorporate rules and constructs in the language so that any operation of the language can be staticly shown to be operating on correctly typed arguments.
- Design a compiler in order to verify that the security rules have not been transgressed, although not enforced by the language.

It was argued that in both cases, the same compiling techniques must be used, and comparable 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).
However when the type union system is infinite (integer subrange type), it has been shown that static type checking necessitates language constructs which allow subtypes to be locally derived.

The argument was based upon the observation that type verification consists in establishing a solution to a system of type equations. Global type declarations give an approximation of the solutions to that system. The discovery of a particular solution from that approximation may involve infinite computations. On the contrary, if the language is designed to directly provide a solution to the compiler, type checking consists in a straightforward verification.

This reasoning might turn out to be useful to language designers who until now could not logically prove the validity of their design of language constructs. Moreover this reasoning may serve as a basis to define type safety in languages and prove particular languages to be type reliable.

7. *Acknowledgements*

P. Cousot benefitted from discussions on this topic with colleagues in IFIP Working Group 2.4 (Machine-oriented higher-level languages). The organization with J. Horning and K. Correll of a debate on EUCLID was especially helpful.

We were very lucky to have F. Blanc do the typing for us.

8. *Bibliography*

ALGOL 60[1963]. Naur P. (Ed.)
"Revised Report on the Algorithmic Language ALGOL 60", CACM, 6, (Jan. 1963), pp. 1-17.

ALGOL 68[1975]. Van Wijngaarden A. et al. (Eds.).
"Revised Report on the Algorithmic Language ALGOL 68", Acta Informatica 5 (1975), pp. 1-236.

Boom[1974].
"Optimization Analysis of Programs in Languages with Pointer Variables".
Ph.D. Thesis, Dept. Appl. Anal. and Comp. Science. University of Waterloo, Ontario (June).

Cheatham and Townley[1976].
"Symbolic Evaluation of Programs : a look at loop analysis". Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation. August 1976.

Cousot[1975].
"Static Determination of Dynamic Properties of Programs". U.S.M.G., RR 25, Nov. 1975, to appear in the proceedings of "Colloque International sur la programmation", Paris 13-15 April 1976, Dunod.

Cousot[1976]
"Abstract Interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints".
Fourth ACM Symposium on Principles of Programming Languages. Los Angeles, January 1977.

Dembinski and Schwartz[1976].
"The pointer type in programming languages : a new approach", to appear in the proceedings of "Colloque International sur la Programmation", Paris, 13-15 April 1976, Dunod.

Dijkstra[1975].
"Guarded commands, non determinacy and formal derivation of programs", CACM, 18, (1975), pp. 453-457.

EUCLID[1976]. Lampson B.W., Horning J.J., London R.L., Mitchell J.G., Popek G.J.
"Report on the Programming Language EUCLID", MOL-Bulletin (P. Cousot ed.), Nb 5, Sept. 1976, pp. 91-172.

Habermann[1973].
"Critical Comments on the Programming Language Pascal", Acta Informatica 3, 47-57 (1973), Springer-Verlag.

Hoare and Wirth[1973].
"An Axiomatic Definition of the Programming Language PASCAL", Acta Informatica, 2, 1973, pp. 335-355.

Karr[1975].
"Gathering Information About Programs", Mass. Comp. Associates Inc. CA-7507-1411, July 1974.

Kildall[1973].
"A Unified Approach to Global Program Optimization", Conf. Record of ACM Symposium on Principles of Programming Languages, Boston, Mass., pp. 194-206, October 1973.

Kleene[1952].
"Introduction to Metamathematics", North-Holland Publishing Co, Amsterdam.

LIS[1974]. Ichbiah J.D., Rissen J.P., Heliard J.C., Cousot P.
"The System Implementation Language LIS" Reference Manual , CII Technical Report 4549 E1/EN, December 1974, Revised January 1976.

Meertens[1975].
"Mode and Meaning", in "New Directions in Algorithmic Languages 1975" (S. Schuman, ed.) pp. 125-138, IRIA, Paris.

Pascal[1974]. Jensen K., Wirth N.
"PASCAL, User Manual and Report", Lecture Notes in Computer Science, Nb.18, Springer-Verlag, 1974.

SIMULA[1974]. Birtwistle, Dahl, Myhrhaug, Nygaard.
"SIMULA Begin", Studentlitteratur, Lund, 1974.

Sintzoff[1972]
"Calculating Properties of Programs by Valuations on Specific Models", SIGPLAN Notices, Vol. 7, Nb.1, Jan. 1972, pp. 203-207.

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, IRIA.

Tarski[1955].

"A lattice-theoretical Fixpoint Theorem and its Applications", Pacific Journal of Mathematics, 5 (1955), pp. 235-309.

Wegbreit[1975].

"Property Extraction in Well-Founded Property Sets", I.E.E.E. Trans. on Software Engineering, Vol. SE-1, nb. 3, pp. 270-285.

Wirth[1975].

"An Assessment of the Programming Language PASCAL", SIGPLAN Notices, Vol. 10, nb. 6, June 1975, pp. 23-30.