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 liberal notion of
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 futuristic 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). Using a simple propagation algorithm from the text 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:

\[
\begin{align*}
(1) & \quad P1 = (pt = L) \land (b = true) \\
(2) & \quad P2 = (P1 \lor P5) \land ((pt \leftarrow nil) \land b) \\
(3) & \quad P3 = (P2 \land (pt!.value = n)) \land (b = false) \\
(4) & \quad P4 = P2 \land (pt!.value \leftarrow n) \\
(5) & \quad P5 = P3 \lor (3 \cdot pt\!.value \leftarrow P4) \land pt = pt!+.next)
\end{align*}
\]
be an empty or non-empty linear list, we get (pt = nil) or (pt <> 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:

\[ \lambda_1 = \langle 1, 1, 1, 1, 1, 1 \rangle \]
\[ \lambda_2 = F(\lambda_1) = \langle \top, (\top \or 1) \text{ and non-nil, } 1, 1, 1, (\top \or 1) \rangle \]
\[ \lambda_3 = F(\lambda_2) = \langle \top, (\top \or 1) \text{ and non-nil, } 1, 1, 1, (\top \or 1) \rangle \]

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:

\[ P_1 = \top, (P_1 = 1, i \in [2, 3]) \]
\[ P_2 = (P_1 \or P_5) \text{ and non-nil} \]
\[ P_3 = P_2 \]
\[ P_4 = P_2 \]
\[ P_5 = (P_3 \or \top) \]
\[ P_2 = (P_1 \or P_5) \text{ and non-nil} \]

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 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 statically 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

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 Meer tens[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

\[
\text{mode list} = \text{union} (\text{ref cell, void})
\]

\[
\text{mode cell : struct (integer value, list next)}
\]

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:

\[
\text{list pt} := \text{L;}
\]

\[
\text{while case pt in}
\]

\[
(\text{ref cell pt'}) \Rightarrow \text{if value of pt'=}n \text{ then false else}
\]

\[
(\text{pt:=next of pt'; true})
\]

\[
\text{fl,}
\]

\[
\text{out} \Rightarrow \text{false}
\]

\[
\text{esac}
\]

\[
\text{do skip od;}
\]

This program is safe, since in ALGOL 68 the non-safe coercion of pt from mode \text{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])).

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:

\[
\text{type mode} = (\text{int, char});
\]

\[
\text{type charint} =
\]

\[
\text{record}
\]

\[
\text{case tag : mode of}
\]

\[
\text{int : (i : integer);} \]

\[
\text{char : (c : character)}
\]

\[
\text{end;}
\]

\[
\text{var digit, letter, alphanum : charint;}
\]

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 \text{alphanum.tag = char}. However this is not statically 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.
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 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 discriminating 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 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 these facts.

<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. Since the value of the test is statically un-</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 = \{Male\} \cap \{Male, Female\} = \{Male\}]. The context delivered for the false path is:

\[
A = \begin{cases} \text{true} & \text{if (} |S_A \cap S_B| - 1 \text{) and not (} S_A S_B \text{)} \text{ then } S_A - S_B \\ \text{false} \end{cases}
\]

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

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:

\[
\begin{align*}
\text{type Person \{Sex = (Male, Female)\} = ...} \\
\text{type Man = Person(Male)} \\
\text{type Woman = Person(Female)}
\end{align*}
\]

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

1.2 Dynamic variants, to describe objects whose
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:

```c
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 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:

```c
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:

```c
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:

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

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

```c
S1 : * DS1;
S2 : * DS2;
```

then 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:

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

var S1, S2 : list;
```

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

```c
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
```

```pascal
    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, 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 sup-

```pascal
    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:

```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 type 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;
```
memory sizes say 1 and 3 words:

```assembly
Type Rtype = (R_a, R_b)
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:

```
  C1
  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 are pointers to the same collection)
- (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 \( \bar{u} \) of such predicates (i.e. the union of sets of collections) for example:

- \( \{A,B,C / D,E\} \bar{u} \{F,A,G / H\} = \{A,B,C,F,G / D,E / H\} \)

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:
\[ X := \mathit{nil}; \]
\[ X := Y; \{ \text{where } Y \text{ is known to be } \mathit{nil} \}\]
\[ \text{if } X = \mathit{nil} \text{ then } \ldots \]
\[ \text{new } (X); \]

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

\[ P_1 = \{ X_1, X_2, \ldots, X, X_n / Y_1, \ldots, Y_n \} \]

the above instructions lead to an output predicate:

\[ P_2 = \text{\textsc{extract}(}X, P_1) \]
\[ = \{ X / X_1, \ldots, X_n / Y_1, \ldots, Y_n \} \]

More generally, with an input predicate \( P_1 \), a pointer assignment such as:

\[ X_.\text{next} \ldots, t_.\text{next} := Y_.next \ldots, t_.next \]

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 \( P_2 = P_1 \cup \{ X, Y \} \). The following PASCAL procedure is supposed to do the job:

\[ \text{procedure copy (}S_1 : \text{list}; \text{var } S_2 : \text{list}); \]
\[ \text{var } C_1, C_2, L : \text{list}; \]
\[ \begin{align*}
\text{begin} \\
\{ P_0 \}
\end{align*} \]
\[ C_1 := S_1; S_2 := \mathit{nil}; L := \mathit{nil}; \]
\[ \{ P_1 \}
\]
\[ \text{while } C_1 \not= \mathit{nil} \text{ do} \]
\[ \begin{align*}
\text{begin} \\
\{ P_2 \}
\end{align*} \]
\[ \text{new}(C_2); C_2_.\text{val} := C_1_.\text{val}; C_2_.\text{next} := \mathit{nil}; \]
\[ \{ P_3 \}
\]
\[ \text{if } L = \mathit{nil} \cdot \text{then} \]
\[ \{ P_4 \}
\]
\[ S_2 := C_2 \]
\[ \{ P_5 \}
\]
\[ \text{else} \]
\[ \{ P_6 \}
\]
\[ L_.\text{next} := C_2 \{ P_7 \}; \]
\[ \{ P_8 \}
\]
\[ L := C_2; C_1 := C_1_.\text{next}; \]
\[ \{ P_9 \}
\]
\[ \text{end} \]
\[ \{ P_{10} \}
\]

end;

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

1. \[ P_1 = \text{\textsc{extract}}(L, \text{\textsc{extract}}(S_2, \text{\textsc{extract}}(C_1, P_0) \cup \{ C_1, S_1 \}) \]) \]
2. \[ P_2 = P_1 \cup P_9 \]
   (Since the test \( C_1 \not= \mathit{nil} \) gives us no information on collections when true)
3. \[ P_3 = \text{\textsc{extract}}(C_2, P_2) \]
   (The assignment of non-pointer values and a deep modification in the structure pointed to by \( C_2 \) are ignored)
4. \[ P_4 = \text{\textsc{extract}}(L, P_3) \]
5. \[ P_5 = \text{\textsc{extract}}(S_2, P_4) \cup \{ S_2, C_2 \} \]
6. \[ P_6 = P_3 \]
   (Since we ignore the fact that \( L \not= \mathit{nil} \))
7. \[ P_7 = P_6 \cup \{ L, C_2 \} \]
8. \[ P_8 = P_5 \cup P_7 \]
(9) \( P9 = \text{extract}(L,P8) \quad \uparrow \quad \{L,C2\} \)

(The statement \( C1 := C1 \text{. next leaves } C1 \text{ in the same collection} \))

(10) \( P10 = \text{extract}(C1,P1 \uparrow P9) \)

Since the theoretical conditions which ensure that the above system of equations has a solution are verified (Coquelin[1978]) 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:

\* \( P0 = \{S1,S2 / C1,C2,L\} \quad P1 = L, \forall i \in [1,10] \)

(1) \( \Rightarrow P1 = \text{extract}(L,\text{extract}(S2,\text{extract}(C1,P0) \quad \uparrow \quad \{C1,S1\})) \)

\* \( = \text{extract}(L,\text{extract}(S2,\{S1,S2/C1,C2,L\} \quad \uparrow \quad \{C1,S1\})) \)

\* \( = \text{extract}(L,\text{extract}(S2,\{S1,S2,C2/L\})) \quad \uparrow \quad \{S1,C1/S2,C2,L\} \)

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

(2) \( \Rightarrow P2 = P1 \quad \uparrow \quad P8 = P1 \quad \uparrow \quad L = P1 \)

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

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

(5) \( \Rightarrow P5 = \text{extract}(S2,P4) \quad \uparrow \quad \{S2,C2\} \)

* \( = \{S1,C1/S2,C2,L\} \quad \uparrow \quad \{S2,C2\} \)

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

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

(7) \( \Rightarrow P7 = P6 \quad \uparrow \quad L,C2 \)

\* \( = \{S1,C1/S2,C2,L\} \quad \uparrow \quad \{L,C2\} \quad \uparrow \quad \{S1,C1/S2,C2,L\} \)

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

(8) \( \Rightarrow P8 = P5 \quad \uparrow \quad P7 \)

\* \( = \{S1,C1/S2,C2,L\} \quad \uparrow \quad \{S1,C1/S2,C2,L\} \)

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

(9) \( \Rightarrow P9 = \text{extract}(L,P8) \quad \uparrow \quad L,C2 \)

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

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

(2) \( \Rightarrow P2 = P1 \quad \uparrow \quad P9 \)

\* \( = \{S1,C1/S2,C2,L\} \quad \uparrow \quad \{S1,C1/S2,C2,L\} \)

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

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

* (4) \( \Rightarrow P4 = \text{extract}(L,P3) = \{S1,L'/S2/L/C2\} \)

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

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

(7) \( \Rightarrow P7 = P6 \quad \uparrow \quad L,C2 \)

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

(8) \( \Rightarrow P8 = P5 \quad \uparrow \quad P7 \)

* \( = \{S1,C1/S2,C2,L\} \quad \uparrow \quad \{S1,C1/S2,C2,L\} \)

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

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

(10) \( \Rightarrow P10 = \text{extract}(C1,P1 \quad \uparrow \quad P9) \)

* \( = \text{extract}(C1, \{S1,C1/S2,C2,L\} \quad \uparrow \quad \{S1,C1/S2,C2,L\}) \)

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

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

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" 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:

\[
\begin{align*}
\{A,B,C\} & = \tau \\
\{A,B\} & \quad \{A,C\} \quad \{B,C\} \\
\{A\} & \qquad \{B\} \qquad \{C\} \\
\{\} & \qquad \{\} = \bot
\end{align*}
\]
whereas EUCLID uses a simplified type inclusion scheme:

\[
\begin{align*}
\{A,B,C\} &= \top \\
\{A\} &= \{B\} = \{C\} \\
\{\} &= \bot
\end{align*}
\]

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:

\[
i := 1; \hspace{1cm} (P1)
\]

```
while i <= 1000 do
   i := i + 1 \hspace{1cm} (P2)
   i := i + 1 \hspace{1cm} (P3);
```

- 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:

```
type 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. (Wirch [1975]). In Cousot[1975], we developed a technique to have the compiler discover the subrange of

```
integer variable, let in the type of the variable.
```

```
i := 1; \hspace{1cm} (P1)
while i <= 1000 do
   i := i + 1 \hspace{1cm} (P2)
   i := i + 1 \hspace{1cm} (P3);
```

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. \(P3 = P2 \cap [1, 1]\)
4. \(P4 = (P1 \cup P3) \cap [1001, +\infty]\)

where \(\cup\) is defined by \([a,b] \cup [c,d] = [a+c,b+d]\), and \(\cap\) 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 verify that this solution is a fixpoint of the system:
(1) \[ P_1 = [1, 1] \]

(2) \[ P_2 = (P_1 \cup P_3) \cap [0, 1000] \]
\[ = ([1, 1] \cup [2, 1001]) \cap [0, 1000] \]
\[ = ([1, 1001] \cap [0, 1000]) \]
\[ = [1, 1000] \]

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

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

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 problem is to give up.
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
results will be obtained by type checking or type
discovery as long as finite type systems are consi-
dered. The main difference between these approaches
8. Bibliography

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

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

Bohm[1974].
"Optimization Analysis of Programs in Languages with Pointer Variables".

Cheatham and Townley[1976].

Cousot[1975].

Cousot[1976].

Dembinski and Schwartz[1976].

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


Habermann[1973].

Hoare and Wirth[1973].

Kerr[1975].

Kildall[1973].

Kleene[1952].


Meertens[1975].

Pascal[1974]. Jensen K., Wirth N.


Sintzoff[1972].
Sintzoff[1975].

"Verifications d'assertions pour les fonctions