Compile-time type-matching

By J. B. Hext*

Run-time type-matching is an inefficient process which can be avoided by a suitably structured language. A procedure is outlined for analyzing a program at compile-time to determine the types of all its expressions and to insert transfer functions on them where necessary. This is given a systematic basis by treating the available types as a partially ordered set. Applications are illustrated from CPL. The treatment is extended to cover structures, arrays and procedures.

If x is an ALGOL variable of type **real**, the statement

$$x := 1$$

is usually interpreted as the assignment of an **integer** constant to a **real** variable, which therefore requires the insertion of an **integer-to-real** transfer function. Many implementations urge the programmer to write

$$x := 1 \cdot 0$$

instead, because otherwise the transfer will be made every time the assignment is executed. More sophisticated versions avoid this inefficiency by making the transfer once and for all at compile-time. The programmer is thereby relieved of one rather tiresome optimizing detail.

If i is an **integer** variable, then

$$x := i$$

also requires an **integer-to-real** transfer. But in this case the transfer cannot be carried out at compile-time because the value of i is not known. However, a good compiler will insert the transfer function in the program, so that once again there is no need to check anything at run-time.

Unfortunately this is not always possible in ALGOL (see Naur, *et al.*, 1963), since the type of an expression may be unpredictable. In particular, if i is a formal parameter called by name its type need not be specified, and so a run-time check is required. Some implementations avoid this difficulty by requiring that the types of all formal parameters be specified. But there still remains the awkward case of

$$x := 2 \uparrow i$$

in which the type of the right-hand side is **integer** if i is positive but **real** if i is negative. So once again the need for a transfer function must be checked at run-time.

It would clearly be possible to remove these features of ALGOL so that run-time type-checking is no longer required. The compiler could then carry out the type-checking by some fairly simple, *ad hoc* process, since the range of types is so small. However, in languages with a greater range of data structures, such as CPL and PL/1, the problems of compiling become increasingly complex and a more systematic treatment is required.

The technique described below was originally developed for CPL. Illustrations will be taken from this language, since it raises several interesting problems. But the approach chosen can be used as a basis for handling any number of types and type-matching problems.

Features of CPL

The following features of CPL (Cambridge, 1965) will be referred to.

1. The conditional expression:

$$B \rightarrow E1, E2$$

This is equivalent to the ALGOL form

$$\textbf{if } B \textbf{ then } E1 \textbf{ else } E2$$

2. The **value of** expression:

$$\textbf{value of } \langle \text{block} \rangle$$

The block must contain commands of the form

$$\textbf{result is } E$$

and the first such command encountered during execution returns the value of E as the value of the expression.

3. Initialized declarations:

$$\textbf{let } i = 1;$$

$$\textbf{rec } f\,[\textbf{index } n] = n = 0 \rightarrow 1, n \times f\,[n-1];$$

The types of i and f are deduced from the right-hand sides. Functions enclose their arguments in square brackets; they are not self-referencing unless preceded by **rec**.

4. Structured variables and multiple assignments:

```
let x, y, z = 0·5, −0·5, (1·5, −1·5);
    x, y := y, x;
       z := x + 1, y + 1;
```

x and y become **real**, z becomes (**real, real**). The multiple assignments have an obvious and very useful meaning.

A type-matching procedure

The action of a type-matching routine, *TYM*, may be described in terms of three subsidiary procedures: *Type*, *Settype* and *Uptype*. *TYM* analyzes each section of a

* *Basser Computing Department, University of Sydney, Sydney, N.S.W., Australia. Formerly at the University Mathematical Laboratory, Cambridge, England. This work was carried out at Cambridge under a grant from the Science Research Council.*

program in turn and applies these procedures as required.

The central operation is to determine the type of an expression, E, by means of the function *Type*. This function not only evaluates the type of E, but also carries out, as a side-effect, a complete analysis of the structure of E, inserting transfer functions where necessary and distinguishing the particular forms of any polymorphic operators. For example,

$$Type\ [`x + 1']$$

will give the result **real**. It will also apply an **integer-to-real** transfer function to '1' and replace '+' by '**real** +'.

If E involves an expression of the form

value of ⟨block⟩

then *Type* will call in *TYM* to analyze the block. *TYM*, *Type*, *Settype* and *Uptype* are all recursive procedures defined in parallel.

The operation *Settype* records the type of each variable as deduced from its declaration. For example, the action of *TYM* on encountering the declaration

let $x = E$

is

$$Settype\ [x, Type\ [E]]$$

The complications caused by recursive declarations are discussed below.

The routine

$$Uptype\ [\langle\text{expression}\rangle\ E, \langle type\rangle\ T]$$

analyzes E in much the same way as *Type*, but finishes by inserting transfer functions, if necessary, to give E the type T. For example, *TYM* operating on the command

if B **then do** $x := E$

would carry out

Uptype [B, **Boolean**]
Uptype [E, *Type* [x]]

Basic types

A *basic type* describes the nature of a single item of data, such as a number, truth value or list element. The basic types in ALGOL are **real**, **integer**, **Boolean** and **string**. CPL includes several others such as **index**, **double**, **complex** and **logical**. In each case there is a finite collection of basic types and any operations involving them can be listed explicitly.

In some other languages the situation is more complicated. For example, in PL/1 (IBM, 1965) a distinction is made between the *type* of a quantity and its *representation*. There are only four types: arithmetic, character string, bit string and label. But within one type there may be many representations. For instance, the representation of an arithmetic quantity comprises its base (binary/decimal), scale (fixed/float), mode (real/complex) and precision. If a programmer does not specify the representation of a variable, then a set of default values is assumed; the precision default is defined separately for each implementation.

It will be seen that the PL/1 notion of representation covers the CPL distinctions between the types **index**, **integer**, **real**, **double**, **complex**, etc. In both cases, the type and representation of a variable remain constant throughout a program (precision in PL/1 can be defined only by decimal integer constants). For the purposes of this paper, therefore, no distinction will be made between type and representation: both concepts will be included in the term "type".

However, one important feature emerges from PL/1, namely that the number of basic types can be effectively infinite. The transfer functions between them therefore cannot be listed explicitly. Instead, we must assume the existence of some other technique for defining all the possible transfer functions. A similar situation arises in the consideration of type structures, which will be discussed below.

For the purposes of subsequent development, it is useful to introduce two special basic types, **unknown** and **general**. **unknown** may be thought of as the type which is assumed for every variable before the program is analyzed. If, when the analysis is finished, any variable still retains the type **unknown**, then an error is assumed. Such would be the case for the declaration

rec $x = x$;

The type **general** applies to a variable whose type varies at run-time. For example, in the declaration

let $x = Cond \rightarrow$ **true**, $2\cdot3$;

the type of x can only be determined dynamically; but it is essential to give it some classification at compile-time, and so it is given the type **general**. If such dynamic types are not implemented, then this also causes a compile-time report.

Partial ordering

It is essential for the operation of *TYM* that every expression has a type which can be determined at compile-time. A case of particular interest is that of the conditional expression. For example, if i is **integer** and x is **real**, what is the type of

$$B \rightarrow i, x$$

The answer may be given, as suggested above, that this must be determined dynamically and that the type is therefore **general**. However, it is reasonable, and in many ways preferable, to say that its type is **real**. We choose **real** rather than **integer** because it preserves a higher precision: we do not wish to round off x to an integer unless we have to.

In general, we may ask, if x, x' have types t, t' what is the type of

$$B \rightarrow x, x'$$

The answer is best given by defining a *partial ordering* on the set of all basic types. This is a relation $\geqslant$ such that

(i) for all t, $t \geqslant t$
(ii) if $t \geqslant t'$ and $t' \geqslant t$, then $t = t'$
(iii) if $t \geqslant t'$ and $t' \geqslant t''$, then $t \geqslant t''$

In this case, the relation $t \geqslant t'$ implies that any quantity of type t' may also be represented as one of type t without loss of accuracy.

A partially ordered set may be depicted by a *Hasse diagram.* **Fig. 1** is a diagram for 8 basic types in CPL, which illustrates the ideas involved. A path from t' to t indicates $t \geqslant t'$. We say that t is "greater than" t', or that it is a "higher type than" t'. The path normally implies the existence of a transfer function between t and t'.

It is often the case that $t \ngeqslant t'$ and $t' \ngeqslant t$; for example, $t =$ **Boolean**, $t' =$ **real**. The point of interest in the diagram is then their *least upper bound*, L say. This, if it exists, is the unique lowest element L such that $L \geqslant t$ and $L \geqslant t'$, in this case **general**. We write

$$L = t \vee t'$$

If $t \geqslant t'$, then $L = t$. A *greatest lower bound* may be defined similarly.

A *lattice* is a partially ordered set in which every pair of elements possesses a least upper bound and a greatest lower bound. By introducing the types **unknown** and **general** in the above way, we have ensured that the set of basic types constitutes a lattice.

We may now return to the question—what is the type of

$$B \rightarrow x, x'$$

and answer $t \vee t'$. Provided that the types are ordered to form a lattice, this is always well defined. It also satisfies any intuitive notions in the simple cases.

This concept has other useful applications. For example, the type of the expression

```
value of §  .......
            result is E1;
            .......
            result is E2;
               .
               .
               .
            result is En  §
```

may be defined as

$$Type\,[E1] \vee Type\,[E2] \vee \ldots \vee Type\,[En]$$

A third application arises in the definition of recursive functions, as described below.

When the number of basic types is infinite, it is not possible to display their ordering in a Hasse diagram. But it will normally be possible to define it by some other means. The next section illustrates a case in point.

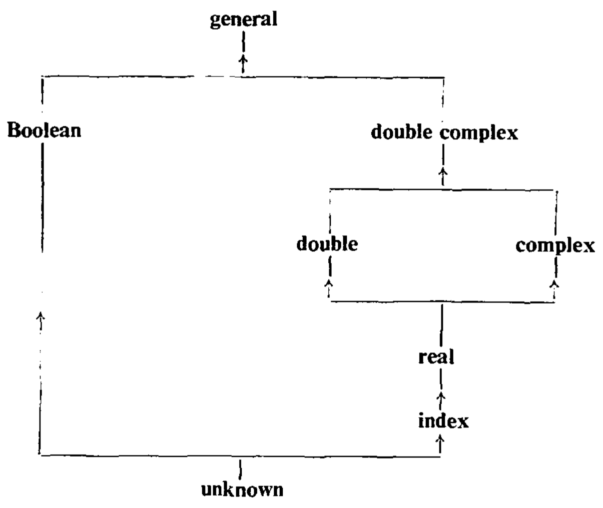

Fig.1. – Hasse diagram

Type structure

In CPL, as also in PL/1, a variable may represent more than one basic quantity. It may be (**real**, **real**) or (**Boolean**, **index**) or even (**Boolean**, (**real**, **real**)). This is especially useful for functions which return more than one result. For example, integer division normally yields a quotient and a remainder: the type of its result is therefore (**integer**, **integer**).

If we think of a type t as representing the space of all quantities of that type, and t' similarly, then the type (t, t') may be thought of as the Cartesian product of these two spaces. If z has this type, then its components may be accessed by a multiple assignment of the form

$$x, x' := z$$

Alternatively, explicit selector functions may be provided; PL/1's qualified names serve a similar purpose.

The notation used above represents type structures by means of the LISP notation for lists (see McCarthy, *et al.*, 1963). The basic types are the atoms; a type structure is a list of such atoms, running to arbitrary depth. We may therefore employ the concepts of list-processing in order to define various operations on type structures. In the CPL notation, we have two functions, *Hd* and *Tl* (LISP's 'car' and 'cdr'), for accessing the front member and the remainder of a list. An empty list is denoted by **nil**, which is also an atom.

Following Iverson (1962), we say that two types are *compatible* if their structures are the same:

$$\begin{aligned} Compat\,[t, t'] = {} & Atom\,[t] \wedge Atom\,[t'] \rightarrow \textbf{true}, \\ & Atom\,[t] \vee Atom\,[t'] \rightarrow \textbf{false}, \\ & Compat\,[Hd[t], Hd[t']] \wedge Compat\,[Tl[t], Tl[t']] \end{aligned}$$

We can extend binary operations to compatible lists by applying them to pairs of corresponding atoms, t_i and t'_i. Thus we define

(i) $t \geqslant t'$ if and only if $t_i \geqslant t'_i$ for each i.

(ii) $T = t \vee t'$ if and only if T is also compatible and for each i

$$T_i = t_i \vee t'_i$$

(iii) There exists a transfer function between t and t' if and only if there exists one between each t_i and t'_i.

None of these definitions apply if t and t' are not compatible. The best we can say in such cases is that

$$t \vee t' = \mathbf{general}$$

There is an alternative approach which classifies structures according to the numbers of their atoms. Thus t and t' are *similar* if they have the same number of atoms. Clearly compatibility implies similarity. For the class of structures containing n atoms there exists an obvious canonical form, namely the one-level list of n atoms. A partial ordering of similar types can then be given in terms of their canonical forms. For example, the least upper bound of

((integer, real), real)

and

(real, (real, integer))

would be

(real, real, real)

The conversion of a type structure t to canonical form is given by

```
Flatten [t]   = Null [t] → nil,
                Atom [Hd[t]] → Cons [Hd[t],
                  Flatten [Tl[t]]],
                Concat [Flatten [Hd[t]],
                  Flatten [Tl[t]]]
where Concat [t, t'] = Null [t] → t',
                Cons [Hd[t], Concat [Tl[t], t']]
```

The inverse operation is similar but rather more complicated.

This approach allows greater flexibility in the use of structured variables, though its advantages in practice are probably not very great. However, despite its additional complications, it is comparatively easy to implement, since a program operating with a conventional linear stack mechanism would be using the canonical forms anyhow.

Functions

The type of a function is a notion which is already familiar in mathematical logic. If f is a function with domain D and range R, its type may be denoted by

$$[D \rightarrow R]$$

On encountering an application of f to some argument, x say, *TYM* will call in *Type* to determine the type of the expression $f[x]$. *Type* will first carry out

$$\mathit{Uptype}\ [x, D] \qquad (1)$$

and then return the result R.

If f is a parameterless function, its type may be denoted by

$$[\mathbf{nil} \rightarrow R]$$

Its application must then be written (in CPL) as

$$f[\ ]$$

On encountering this expression, *Type* will not carry out (1) but will simply check that the domain of f is **nil**. An attempt to carry out

$$\mathit{Uptype}\ [x, \mathbf{nil}]$$

will always cause a report, and in this sense **nil** may be thought of as representing the null space.

If f takes another function as its argument, its type is

$$[[D \rightarrow R_1] \rightarrow R_2]$$

If D is a basic type, we say that f is a function of class 2. Similarly, if f has the type

$$[[\ldots [D \rightarrow R_1] \rightarrow R_2] \ldots \rightarrow R_n]$$

we say that it belongs to class n. In this hierarchy the basic types belong to class $\emptyset$; the compound type

$$(t_1, t_2, \ldots, t_n)$$

belongs to the class

$$\mathit{max}\ [\mathit{class}\ [t_1], \mathit{class}\ [t_2], \ldots, \mathit{class}\ [t_n]]$$

and the function type

$$[D \rightarrow R]$$

belongs to the class

$$\mathit{class}\ [D] + 1$$

This classification of function types excludes the possibility that a function can take itself as an argument. There is a close parallel in mathematical logic, in which a similar theory of types was introduced to avoid certain antinomies, such as:

this sentence is not a true sentence

McCarthy (1963) has shown that, by dropping this classification and allowing functions to take themselves as arguments, it is possible to introduce the power of recursion without a recursion operator. But the complications do not justify the step in practice.

A function in CPL is defined by a declaration such as

$$\mathbf{let}\ f\ [\mathbf{real}\ x] = E$$

where E is some expression involving x. The domain of f is thus specified explicitly, namely **real**; its range is *Type* $[E]$.

A problem arises when f is recursive: if E involves f, then *Type* $[E]$ must be evaluated before the types of all its components are known. The difficulty may be overcome by making successive approximations to R. The first approximation, t_0, is **unknown**. We then define

$$t_i = t_{i-1} \vee Type\,[E] \ldots i \geqslant 1$$

re-assigning t_i to R after each evaluation. The approximations end when

$$t_i = t_{i-1}$$

The approximations will always terminate provided there is an upper bound to the length of paths in the types lattice. Only the most extreme pathological cases could cause an infinite evaluation. Normally t_1 will give the required result, as in

$$\textbf{rec}\, f\, [\textbf{index}\ n] = n{=}0 \rightarrow 1,\ n \times f\,[n-1]$$

The method may be extended in an obvious way to cover any number of functions defined in parallel.

Routines and arrays

A routine is similar to a function, except that it does not produce any result. Its type is therefore represented simply as

$$[D]$$

where D is its domain.

An array is subject to various operations and restrictions which distinguish it from a function. However, for the purposes of type-matching it can be handled in much the same way as a function. We therefore give its type a similar form, namely

$$[D : R]$$

where D is the type of its subscript list and R the type of its elements. For example, the CPL type **matrix** is equivalent to

$$[\textbf{index}, \textbf{index} : \textbf{real}]$$

This allows for the possibility that arrays should be subscripted by some type other than **index**. For example, a truth table may be regarded as an array with **Boolean** subscripts.

Further transfer functions

If a formal parameter is a function, the programmer is normally required to ensure that its corresponding actual parameter has the necessary type. However, it is theoretically possible to allow its type to differ and to insert a transfer function accordingly. Such a transfer function would have the form

$$[t_1 \rightarrow t_2]\text{-to-}[t_3 \rightarrow t_4]$$

Its result, when applied to f (of type $[t_1 \rightarrow t_2]$) would be a new function, g say, defined by

$$g[x] = t_2\text{-to-}t_4\ [f[t_3\text{-to-}t_1[x]]]$$

The right-hand side, of course, represents the action taken when f is applied to an argument of type t_3 in a context which requires a result of type t_4: t_3-to-t_1 is applied to its argument and t_2-to-t_4 to its result, but f itself is not affected. The need for such a transfer arises only in an actual-formal correspondence, as above, or when assignments are allowed between function variables.

A similar treatment may be given to routines, but not to arrays. The type of an array's subscript list cannot be altered without turning the array into a function. However, the type of its elements can be changed by applying the appropriate transfer function to each element in turn.

Transfer functions such as these, though clearly definable in theory, are not so easy to implement; nor would they often be of any essential practical value. We shall not therefore pursue their basic structure any further.

Implementation

The treatment of types as outlined above (except for the previous section) has been used in the CPL compiler at Cambridge. Transfer functions are automatically inserted at compile-time wherever necessary, and executed at compile-time wherever possible.

The type-matching procedures are implemented by means of list-processing techniques. A syntax analysis reduces the program to a list structure, which is then subjected to various processes. One of these is to link every occurrence of an identifier with its corresponding declaration. A property list is formed for each variable, to which *TYM* adds its type. The type of any identifier is then immediately accessible in subsequent analysis.

The time and space required for type-matching are by no means negligible. But the process offers three major advantages: it relieves the programmer of tiresome details, it provides a check on various aspects of his program, and it saves time in execution. By placing the process on a systematic basis, the way is left open for the introduction of additional types at a later stage.

References

Cambridge (1965). *CPL Elementary Programming Manual*, Edition II. University Mathematical Laboratory, Cambridge: internal report.

International Business Machines Corporation (1965). PL/1: Language Specifications. IBM Systems Reference Library, File No. S360-29.

Iverson, K. E. (1962). *A Programming Language*, New York: John Wiley and Sons Inc.

McCarthy, J. (1963). "A basis for a mathematical theory of computation", *Computer Programming and Formal Systems*, Amsterdam: North Holland Publishing Co.

McCarthy, J. *et al.* (1963). *LISP* 1.5 *Programmer's Manual*, Massachusetts Institute of Technology, Cambridge, Mass.

Naur, P. *et al.* (1963). "Revised report on the algorithmic language ALGOL 60", *The Computer Journal*, Vol. 5, p. 349.