AN IMPLEMENTATION OF ECL DATA TYPES

Benjamin M. Brosgol

Center for Research in Computing Technology
Harvard University
Cambridge, Mass. 02138

I. Introduction:

The data-type definition scheme in ECL is designed to furnish its users with a natural notation, in which the composition and behavior of complex objects can be readily described, and which simultaneously produces efficient underlying representations. The purpose of this paper is to discuss how these objectives are met in the implementation of ECL on Harvard's PDP-10.

Section II outlines the fundamentals of data definition in ECL. For a full discussion of the ECL programming system and its language component, the reader is referred to [1], [2], or [4]; an overview of the system is presented in a companion paper by Wegbreit [5]. A complete description of ECL's data extension mechanism is given in [1] and [3].

Section III presents the main features of the implementation of the ECL data definition scheme, discusses the "mode compiler", analyzes the internal representation chosen for data objects, and describes the efficiencies resulting from this representation.

Finally, section IV deals with some of the interfaces between the data extension mechanism and the rest of the ECL system, and section V proposes some areas where further research would be useful.

II. Main Features:

The fundamental feature of the ECL data definition facility is that data-types (hereafter called modes) are values in the language — there are mode-valued constants, mode-valued variables, and executable routines which can take modes as arguments or deliver them as results.

For example, the built-in data-types provided by ECL are mode-valued constants: these include BOOL, INT, REAL, CHAR, REF, MODE, and SYMBOL. The first four are the data-types for boolean (logical), integer, real (floating-point), and character values. A REF is an object which can point to any value allocated in the heap. A data-type, whether built-in or user-defined, has mode MODE.

And SYMBOLs include, among other things, any variable names appearing in the course of an ECL program.

The ECL programmer can define new data-types falling into five classes, corresponding to the mode construction routines ROW, STRUCT, PTR, PROC, and ONEOF. We illustrate these routines in turn:

A. ROW

The routine ROW is used to create a new mode M, each of whose instances consists of some number of components, each component having the same mode, say M'. For example,

```
triple←ROW(3,INT);
m1←ROW(begin k>0 => 2*k; 4 end, triple);
m2←ROW(BOOL);
```

Any object of mode triple will consist of three components, each an INT. If k>0, each object of mode m1 will comprise 2*k sub-objects, where each sub-object is a triple; if k is less than or equal to 0, each m1 will consist of 4 triples. And an object of mode m2 will comprise some number of BOOLs, the number (the size of the object) specified when the object is created; different objects of mode m2 can be of different sizes. The mode m2 is said to be "length unresolved" in this instance. The built-in mode STRING is also length unresolved, being equivalent to ROW(CHAR).

B. STRUCT

An object whose data type is of class ROW is subject to the restriction that all its components have the same mode and size. A second class of modes, STRUCT, allows composite objects whose components do not necessarily have the same mode. For example,

```
record←STRUCT(id:STRING, history:triple, flag:BOOL);
```

This assignment establishes record as a mode-valued variable whose value is a STRUCTure of three components. The first component is a STRING, the second is a triple, and the third is a BOOL; the names "id", "history", and "flag", respectively, may be used as selectors for these components. Since STRING is a length unresolved mode, so is record.

C. PTR

ECL provides both a stack discipline, as in ALGOL 60, which expands and contracts on block entry and exit, and a free storage region, or heap, as in LISP and ALGOL 68, which is periodically garbage collected. Objects existing in the heap can be referenced only through pointers; e.g., a datum of mode REF is a pointer which can point to (i.e., contain the address of) an arbitrary object allocated in the heap, thus permitting the sharing of values.

The mode construction routine PTR is used to define a mode which we may term "restricted pointer". For example,

```
printname←PTR(string);
```

produces a mode whose objects are restricted to point only to data of mode string. Similarly, objects of mode arith\ptr, where the latter is defined by:

```
arith\ptr←PTR(INT, REAL);
```

can point only to objects of mode INT or objects of mode REAL. While anything that can be done with a restricted pointer can also be accomplished by a REF, the former is useful from efficiency considerations: (1) in many cases storage can be saved by not carrying type information explicitly with the pointer, as must be done with REFs, and (2) more efficient compilation can result when the mode of the object pointed to is known.

D. PROC

An object whose mode is PROC(M1,...,Mn; MO) will be a routine whose arguments have data types M1,...,Mn respectively, and whose result has mode MO. For example,

```
hash\fns←PROC(string; INT);
```

defines a mode whose objects are routines which, given a string as argument, produce an INT.

E. ONEOF

It is frequently useful to have a routine which has an argument, result, or local variable whose mode can vary from one call to another; e.g., a function which takes an INT or a REAL and produces as result an INT or a REAL. This can be accomplished through the mode construction routine ONEOF. To illustrate,

```
arith←ONEOF(INT, REAL);
arith\fn←PROC(arith, arith; arith);
```

An object whose mode is arith\fn is said to be a <u>generic</u> routine.

ECL provides as a built-in data type the mode ANY, which denotes the "union" of all modes; i.e., ANY bears the same relation to ONEOF as REF does to PTR.

III. Implementation:

A. The Allocation and Completion of DDBs

All of the information which the ECL system needs concerning a data type is contained in a body of storage called a "DDB", or "data definition block". A DDB is a structure which includes several fields: e.g., pointers to selection, generation, assignment, and garbage collection functions; a variety of BOOLs indicating whether the mode is length resolved, whether objects of the mode occupy less than one word and whether they include any embedded pointers; etc. Each primitive data type has a DDB which is assembled into the system, while a user-defined type may cause the creation of a new DDB.

To illustrate the basic mechanics of the implementation, we assume that the user is defining:

```
tuple←ROW(n, INT);
```

An "external denotation" (or "canonical name") is constructed; if n has the value 5, then this denotation is the symbol "ROW(5,INT)". A check is made to see if this is also the denotation of any other data type — if so, the value of the mode tuple is a pointer to the DDB for this data type; if not, a new DDB is constructed, and the value of the mode tuple is a pointer to this DDB. Thus a mode value is a pointer to a DDB — the built-in definition for MODE is, in fact, PTR(DDB).

Continuing with the example, let us suppose that no other data type has the canonical name "ROW(5,INT)". Then a DDB gets allocated (by calling the generation function for the mode DDB), and several fields of the DDB are filled in: e.g., the canonical name, the class (here ROW), and the descriptor. The latter is a structure comprising an INT (here 5) and a MODE (here INT).

At this point the "mode compiler" is called. Given the descriptor of the new data type, this system routine deduces the storage layout for objects of the mode and, based on this internal representation, compiles three pieces of machine code: a generation function, which will be called whenever an object of mode tuple is to be created; a selection function, called when a selection is performed on a tuple; and an assignment function, invoked when an assignment of one tuple to another is performed.

After the compilation of these functions, the remaining fields in the DDB are filled in, and a pointer to the DDB is created; this pointer is the value of the new mode.

It should be mentioned that the sequence of events will be somewhat different in some cases. For example, when the mode to be constructed is a ONEOF, the mode compiler is not called, since there will never be an object whose mode is a ONEOF.

B. The Mode Compiler

The decision to implement a mode compiler which produces generation, assignment, and selection functions tailored to the individual mode was based primarily on two considerations:

1) The alternative (say a single system function SELECT which takes, as arguments, an object X, its mode M, and the index I) would be unattractively slow. Savings of up to 50% of execution time can be realized by having mode specific functions.

2) Having separate assignment and generation functions for each mode facilitates the efficient treatment of "sensitive" modes and data monitoring. E.g, attempted generation of an object of mode M can be trapped by the generation function for M, thus avoiding payment of any overhead when generating objects of other modes.

Rather than compile large bodies of code in line for each function it compiles, the mode compiler instead produces several words summarizing the essential features of the mode, followed by a call on a block of machine instructions. The result is a considerable saving of storage, at the expense of three or four extra machine instructions during function execution.

C. Internal Representation

As mentioned earlier, when the mode compiler is given the description of a new data type, it deduces a storage layout for objects of the new mode. The fundamental notion which shapes this representation is the "storage axiom" [1, page 328], which asserts that for any mode M, any object of mode M will occupy a contiguous block of storage, whose size is fixed for the lifetime of the object. This axiom essentially rules out the use of hidden pointers in the implementation and simplifies the management of stack and heap storage. While ruling out such features as arrays with flexible bounds, these features can be obtained as extensions, with the user exercising precise control over how they are handled.

Another significant factor influencing the choice of storage layouts is the desire to minimize the time required to perform such operations on objects as selection, assignment, generation, size calculation, and tracing (during garbage collection). The attempt was made to choose

a representation to minimize the amount of storage taken by objects while allowing these operations to be performed efficiently.

The actual storage layout chosen for an object depends on the class of the object's mode:

a) Primitives

A BOOL will take 1 bit, a CHAR 7 bits (its ASCII code), and an INT and a REAL one word each. Since addresses on the PDP-10 take 18 bits, and since a mode, being a pointer to a DDB, is an address, a REF consists of a full word: half of it is the mode, and the other half the address, of the object referenced.

b) PTRs

An object whose mode is a simple PTR will occupy 18 bits. That is, if the mode of X is PTR(M) for some mode M, then X takes 18 bits, independent of M.

If the mode of the object Y is a united PTR (i.e., PTR(M1,...,Mn)) then Y will look like a REF unless all the Mi are spaced, in which case Y will only occupy 18 bits. (A mode M is spaced if there is a special segment of heap which is used solely for objects of mode M, so that the address is sufficient to determine the data type.) Among the spaced modes are the primitives INT and REAL and the built-in data types DDB and SYMBOL.

For "composite" modes (i.e., ROWs and STRUCTs) the internal representation will depend critically on the storage layout of the sub-objects, since we are abiding by the "axiom of contiguous storage". The general scheme is to pack byte sub-objects as tightly as possible, except that a byte cannot overlap between one word and another. Similarly, each object requiring a full word or more will begin on a full-word boundary. These assumptions were adopted to match the byte and full word addressing conventions on the host machine, the PDP-10.

To illustrate the internal representation for some of the various ROWs:

ROW(4,CHAR); Each object takes 28 bits.

ROW(37,BOOL); Each object takes 2 words (the word size on the PDP-10 is 36 bits)

ROW(CHAR); For each object X there is a size specification K; X will take w words, where w=1+(K+4)/5. The first word (the "header") consists of w in one half, K in the other; the remaining w-1 words contain the K CHARs.

ROW(ROW(BOOL)); For each object Y there is a size specification K1, K2; Y will take w words, where w=1+K1*w', with w'=1+(K2+35)/36. The first word of Y consists of w and K1; the remainder of Y comprises the K1 sub-objects. Each sub-object has a layout analogous to the one described in the previous case, ROW(CHAR).

Choosing an internal representation for objects whose mode is of class STRUCT is complicated by the fact that the components can be of different modes. In this case the storage will be laid out so that the byte components are first, packed according to a scheme which scans the byte sub-objects in decreasing order of size and arranges the placement of the currently scanned byte in the first word that has enough bits remaining. Following the byte components are the sub-objects requiring at least a full word; and finally appear the components (if any) whose modes are length unresolved. Though not logically necessary, a set of "internal pointers" (relative addresses) is used to facilitate selection on these latter sub-objects (see [1, page 323]). For each mode of class STRUCT, a table is compiled (as part of the selection function) which encodes this representation.

As far as the other mode classes are concerned, PROCs are treated as a special type of PTR, and there is never an object whose mode is of class ONEOF, so internal representation is not an issue in this case.

D. Effects of Internal Representation

The storage layout scheme described in the preceding subsection has proved to be an efficient host for ECL's data extension facility. For example: explicit size information is never stored with an object of length resolved mode; selection on an object whose mode is a length unresolved matrix is fast because of the easy retrieval of the index bound and component size from the object.

As mentioned earlier, a function produced by the mode compiler will typically consist of several words encoding the features of the specific mode, followed by a call on one of several blocks of machine language. For example, there are 13 distinct routines for row selection functions, with the categories corresponding both to natural differences in representation and special efficiencies desired because of their expected frequency of use. In the case of assignment functions there are 9 separate routines, while for generation functions there are 21. (In the latter case the mode compiler goes to rather great lengths to compile an efficient function: e.g., in some cases a "template" is produced which only needs to be copied when an object is generated.)

IV. System Interfaces:

The direct application of the mode compiler is to produce functions which will be called from the interpreter

to perform selection, assignment, and generation for objects of user-defined modes (thus these operations are partly compiled, even from interpreted programs). In addition to this use, however, there are other times when the mode compiler, or the functions it produces, are invoked.

One such occasion is during system initialization. ECL was carefully structured, being formally defined through its own data extension facility, so that any object required during the execution of a program has an ECL-definable data type. The initialization (or "bootstrapping") phase then includes a sequence of calls on the mode compiler, which constructs functions for each non-primitive built-in mode. (Thus when any system component needs to create an object whose mode is built-in, it will call the generation function compiled for that mode.) Of course, some of these functions have to be assembled in; for example, the functions produced by the mode compiler are objects whose mode is the built-in data type CEXPR (compiled explicit routine), so to get the bootstrapping off the ground (the mode compiler constructs CEXPRs by calling the generation function of the mode CEXPR) a hand coded generation function is assembld into the DDB for CEXPR.

In addition to being a vital part of bootstrapping, the mode compiler is also valuable to the regular ECL compiler. Each function produced by the mode compiler has two entry points: a "B" entry, to which the interpreter branches and which assumes arguments on the name stack; and an "R" entry, which assumes arguments in a predetermined set of machine registers. The ECL compiler, in producing code for an assignment, selection, or generation, can thus compile a call on the "R" entry of the function produced by the mode compiler (if it is trying to optimize on space), or it may compile code in-line (if it is optimizing time).

V. Areas of Further Study:

In the preceding sections we have discussed various aspects of the implementation of the data extension mechanism in ECL; in particular, we concentrated on the issues of choosing an internal representation and compiling efficient functions which employ that representation. Let us conclude by suggesting some places where further research might result in fruitful generalizations.

One such area would involve work in formalizing the illusive concept of "representation". In the implementation of the ECL data definition scheme, the fact that bytes are packed in a ROW or a STRUCT, that a header word for a length unresolved row contains the total length and the number of components, etc., are all implicit in the operation of the

mode compiler and the functions it produces. It would be useful if the representation information could be "factored out"; perhaps the mode compiler could accept a representation as an argument and then compile functions corresponding to this representation.

A related problem would be to allow (but not to compel) the user to influence the way in which his objects are laid out. For example, if he wants to define the mode

```
m1←ROW(2,BOOL);
```

and is more concerned with saving time during selection than with saving space, he might prefer the sub-objects not to be packed into 2 bits, but stored one per word instead.

VI. Summary:

The treatment of data types in ECL is based on a set of fairly straightforward notions, most of which are readily implemented. Through the choice of a suitable internal representation, and a mode compiler which produces generation, selection, and assignment functions tailored to each mode, the ECL programming system provides its users with a highly efficient mechanism for dealing with complex, structured data.

Bibliography

[1] Wegbreit, B., Studies in extensible programming languages, ESD-TR-70-297, Harvard University, Cambridge, Mass., May 1970

[2] Wegbreit, B., The ECL Programming System. Technical report, Division of Engineering and Applied Physics, Harvard University, Cambridge, Mass., April 1971. (To appear in Proc. FJCC 1971).

[3] Wegbreit, B., The Treatment of Data Types in EL1. Technical report, Division of Engineering and Applied Physics, Harvard University, Cambridge, Mass., May 1971 (Submitted for publication).

[4] ECL User's Manual; (In preparation)

[5] Wegbreit, B., An Overview of the ECL Programming System. Proc. of the International Symposium on Extensible Languages, SIGPLAN Notices, Vol. 6, Number 12 (December, 1971).