From ML to C via Modula-3

an approach to teaching programming

Peter Robinson
University of Cambridge

Revised December 1994

1 Abstract

The Computer Science course at the University of Cambridge teaches ML as an introductory language at the beginning of the freshman year, and then uses Modula-3 to introduce imperative programming at the end of that year. Further lectures on advanced features of Modula-3 are given early in the second year, together with separate lectures on C. Other, specialised languages are introduced subsequently as the course progresses.

This paper explains why this strategy for teaching was adopted and evaluates its operation in practice. The key features of ML and Modula-3 are presented and illustrated through a collection of example programs. Finally, a general assessment of the two languages is also presented.

2 Introduction

Choosing the right programming language for a commercial computing project involves balancing a number of conflicting requirements, but is usually resolved by commercial considerations rather than technical ones. Choosing the right language for introducing newcomers to Computer Science is free from such external constraints, and therefore much harder. Indeed, students in scientific disciplines are often taught to program when it is no longer clear that this is relevant as part of their professional development; teaching them to use standard software packages may be more appropriate.

The main Computer Science course at the University of Cambridge is the three-year Computer Science Tripos. Half of the first year is devoted to Computer Science topics (including discrete mathematics), and the other half is drawn from lectures given to Natural Scientists, including continuous mathematics and a particular science subject.

Until recently, all Natural Scientists were taught elementary programming in FORTRAN as part of the mathematics component of their first year course. This has now been changed so that they receive instead lectures on utility computing, using a word processor (Microsoft Word), a spreadsheet (Microsoft Excel) and a symbolic mathematics system (MathCad). These are used to convey the principles of data handling and algorithm design (even going so far as to ilustrate the numerical solution of differential equations in a spreadsheet) which seems sufficient for students who will be computer users rather than developers of new computer systems.

2.1 The first language

Computer Scientists are, of course, rather different. They need to start with a sound foundation for programming that can establish the principles which will subsequently be applied in many different languages. Three main objectives can be established:

Mathematical basis: Formal manipulation of computer programs and proof of their correctness is becoming increasingly important. Students need to see programs as formal descriptions of abstract algorithms. A mathematical language also relates directly to parallel first-year courses in digital logic and discrete mathematics.

Strong typing: The value of strong typing in writing correct and maintainable programs is now well established. This is particularly important in evolution of large systems where a team of programmers may have to work together over a number of years. A rich type system also allows data structures to be introduced clearly.

Functional emphasis: A functional style of programming is conducive to correct programming, and also lends itself to mathematical analysis of algorithms.

It should also be said that a friendly environment for experimenting is a great virtue; this probably implies the use of an interpreted language.

However, it is important to emphasise that commercial relevance is not in this list. A university Computer Science course is not an industrial training course. The graduates' value comes not from their skill with a particular language that happens to be popular at the moment, but from understanding the principles of programming languages in such a way that they can learn and evaluate new languages as they encounter them in their professional careers.

These objectives led us to the choice of ML as the initial teaching language for Computer Scientists. This choice also has an interesting side effect. Students entering the course vary widely in their previous experience of computers and programming, from those who have hardly touched a keyboard to those who may have spent a year programming in industry before coming to university. It is important not to make the beginners feel themseleves to be at a disadvantage, and also not to bore the experts. ML meets these requirements nicely – the experts tend to have used imperative languages such as Basic, C or Pascal and find themselves with no great advantage over the beginners. Indeed, their preconceptions and self-taught programming habits often put them at a disadvantage.

2.2 The second language

After starting with ML, it is useful to move on to a more conventional imperative programming language with a new set of objectives:

Completeness: The language should exhibit all the facilities of a modern language – objects and inheritance, exception handling, garbage collection and concurrency.

Large scale programming: Strong typing should extend across separately compiled modules, but there should be a controlled way of circumventing its protection for low level code. This suggests the use of a separate interface description language.

Libraries and environment: Extensive libraries serve two purposes. First they serve as illustrations of programming style and the construction of re-usuable

code. Secondly, they provide a rich environment of facilties which students can draw on when they undertake substantial projects of their own later in the course.

At the same time, the language should not lose sight of the objectives for an initial programming language listed above.

Again, there is no requirement for the language to be popular with industry; it is the principles that matter.

Modula-3 meets these requirements and is introduced towards the end of the first year of the Computer Science course. The languages thread of the course continues with lectures on C/C++ and Prolog, together with brief historical excursions into LISP and COBOL. C is included as a concession to its widespread use as a sort of machine-independent low-level language, and Prolog introduces a rather different style of programming. This exposure to a variety of programming idioms equips the students to understand, assess and use a very wide variety of languages in practice. For example, many use embedded scripting languages such as Tcl, Python or Obliq in their final year projects with no further formal training.

ML and Modula-3 will now be discussed in more detail, and the paper concludes with an evaluation of their strengths and weaknesses. This is not intended as a complete description of either language, but rather as a summary of their distinguishing characteristics, illustrated by examples.

3 ML

Standard ML is descended from the meta-language (hence ML) for the LCF proof system developed at Edinburgh in the 1970s [4]. It is defined in a report from the University of Edinburgh [7] which is accompanied by a gentler introduction [6]. The language has become popular for teaching programming and a number of introductory texts have now been published [13, 19, 20] together with a more advanced book ilustrating the language's use for a wide variety of problems [15].

The key features of ML are [6]:

Functional: Functions are first class data objects which may be passed as arguments, returned as results and stored in data structures. Assignment and other side-effects are discouraged.

Interactive: Phrases are typed in, analysed, compiled and executed interactively with any results being printed out directly. There are also more conventinal compilers.

Strong typing: Every legal expression has a type which constrains its use. However, most types are inferred by the compiler rather than having to be specified by the programmer.

Polymorphism: The compiler infers the most general type of an expression, and this is specialised in actual use. This supports generic programming with no additional effort by the programmer.

Abstract types: Types can be defined in terms of permitted operations while keeping implementation details private.

Static scoping: All identifiers are resolved at compile time. However, procedure execution can be controlled by pattern matching of arguments at run time.

Type-safe exceptions: Exceptions allow procedures to return out-of-band results (often arising from abnormal conditions) to be communicated in a type-safe way.

Modules: Type safety is maintained when large programs can be constructed out of separately compiled components.

The language will now be illustrated by a number of examples and then evaluated.

3.1 Big numbers

Consider the manipulation of arbitrarily large natural numbers, stored as a list of digits to the base 10. It is convenient to use a little-endian convention, storing the least significant digit at the head of the list. We can start with a couple of utility routines to convert between big numbers and ordinary integers. These can be typed directly into the ML interpreter: the `-` is its normal prompt and `=` its prompt for a continuation line of a phrase. Each complete unit of input concludes with a semicolon. The interpreter responds by printing out the value and type of the input which is also stored as the current value, `it`, initially empty.

```
Standard ML of New Jersey, Version 0.93, February 15, 1993
val it = () : unit
- fun big2int nil = 0
=   | big2int (b :: bb) = b + 10 * big2int bb;
val big2int = fn : int list -> int
- big2int [1, 2, 3];
val it = 321 : int
-  big2int [1, 2, 3, 4, 5, 6, 7, 1, 2, 3, 4, 5, 6, 7];
uncaught exception Overflow
-
```

In this we see the function `big2int` defined and then tested on a couple of input values. The function is defined using *pattern matching*; if the input value is an empty list, `nil`, the literal value 0 is returned, otherwise the input is a list with first element `b` and tail `bb` which represents the integer value of `b` plus 10 times the value of the tail. These patterns appear after the `fun` keyword as repeated definitions of the function separated by vertical bars, `|`. The ML type system infers that the input must be a list of integers and that the result is a single integer, and the interpreter prints the signature of `big2int` as `fn : int list -> int` correspondingly.

This is then tested by trying it out on a couple of lists of integers. The first duly prints out the value 321 as the value of the expression stored in `it` but the second raises a run-time exception when there is arithmetic overflow because the input list represents a number that is too big to fit into an integer. (More recent ML system use arbitrary length integers by default, so this would still work. Of course, it would also render this example somewhat pointless.)

The converse function `int2big` is defined and tested similarly:

```
- fun int2big 0 = nil
=   | int2big i = (i mod 10) :: int2big (i div 10);
val int2big = fn : int -> int list
- int2big 123;
val it = [3,2,1] : int list
-
```

In this case the `::` operator is used to construct a list. Notice how recusion is used to manipulate recursive data structures.

Finally, a procedure to add big numbers and so to compute powers of two can be written:

```
- fun add aa bb =
=   let fun doadd (nil, nil, c) = if c = 0 then nil else [c]
=         | doadd ((a::aa), nil, c) =
=             ((a+c) mod 10) :: doadd (aa, nil, (a+c) div 10)
=         | doadd (nil, (b::bb), c) = doadd ((b::bb), nil, c)
=         | doadd (a::aa, b::bb, c) =
=             ((a+b+c) mod 10) :: doadd (aa, bb, (a+b+c) div 10)
=   in
=     doadd (aa, bb, 0)
=   end;
val add = fn : int list -> int list -> int list
- fun twoto n = if n < 1 then [1] else
=   let val h = twoto (n-1)
=   in add h h
=   end;
val twoto = fn : int -> int list
- twoto 7;
val it = [8,2,1] : int list
- big2int it;
val it = 128 : int
-
```

An auxiliary function `doadd` is defined locally with an extra argument (the carry between the addition of successive digits). In fact there is a further subtle difference between the signatures of `add` and `doadd` relating to the fact that all ML functions take a single argument. `add` is *curried*, so this argument is just the first big number, `aa`, and it returns an anonymous function that takes the argument `bb`, returning the sum of the lists. This is reflected in the signature printed by the interpreter. `doadd` on the other hand takes a triple consisitng of two lists and an integer as its single argument. Its signature is not printed, but would be `fn : int list * int list * int -> int list`.

Finally, this is tested by working out 2^7 and converting the result back to an integer.

3.2 A stack of records

The triple in `doadd` is an example of a record, but its fields are identified by their order. It is also possible to name fields, for example:

```
- val pr = {name = "Peter Robinson", address = "Cambridge"};
val pr = {address="Cambridge",name="Peter Robinson"}
  : {address:string, name:string}
- val mw = {address = "Milton Keynes", name = "Mark Woodman"};
val mw = {address="Milton Keynes",name="Mark Woodman"}
  : {address:string, name:string}
-
```

defines values `pr` and `mw` both of whose types are records with two named fields, both of type string. The order of the fields in the definition is irrelevant; they are automatically arranged in a canonical order. Individual fields can be extracted with a selection operator:

```
- #name pr;
val it = "Peter Robinson" : string
-
```

The simplest way to make a stack of such records would be to write a couple of functions manipulating lists:

```
- fun push (s, r) = r :: s;
val push = fn : 'a list * 'a -> 'a list
- fun pop (r :: s) = (s, r);
std_in:0.0-0.0 Warning: match nonexhaustive
          r :: s => ...
val pop = fn : 'a list -> 'a list * 'a
-
```

The type inference system works out that the `push` function is generic, that is it can operate on stacks of any base type. This is represented by the use of `'a` (read as α) for a type variable. `pop` takes a stack and returns a pair consisting of the popped stack and its former first item. However, there are a couple of deficiencies in this approach: the empty stack is represented by the empty list, `nil`, which is untidy and the action of `pop` on an empty stack is undefined. This is identified by the ML interpreter as an incomplete set of patterns for the arguments to `pop`; an empty list does not match the single pattern and so would give rise to a run-time exception.

A better approach would be to define a stack by the operations permitted on it, more in the object-oriented style. We can define a stack to be either empty or constructed by pushing an item of an arbitrary type α onto an existing stack of αs:

```
- datatype 'a stack = empty | push of ('a stack) * 'a ;
datatype  'a stack
  con empty : 'a stack
  con push : 'a stack * 'a -> 'a stack
- empty;
val it = empty : 'a stack
- push (it, pr);
val it = push (empty,{address="Cambridge",name="Peter Robinson"})
  : {address:string, name:string} stack
- push (it, 42);
std_in:7.1-7.13 Error: operator and operand don't agree (tycon mismatch)
  operator domain: {address:string, name:string} stack
                   * {address:string, name:string}
  operand:         {address:string, name:string} stack * int
  in expression:
    push (it,42)
-
```

Here an abstract, generic type, an α `stack` is defined by its two possible constructors: `empty` which gives an empty stack and `push` which puts an extra item onto the stack. `empty` is then used to produce an empty stack which appears as the current expression, `it`. Note that `it` has a generic type at this point. A name and address record is then pushed onto the stack, whose type now becomes specifically that of a stack of name and address records. It would now be possible to push other such records (such as `mw` defined above) onto the stack, but instead an attempt is made to push an integer onto the stack; this is an incompatible type and a diagnostic message is printed.

A funtion to pop items off the stack matches against the two possible patterns, but first an exception is defined to deal with the special case of an empty stack:

```
- exception nocando;
exception nocando
```

```
- fun pop empty = raise nocando
=   | pop (push (s, r)) = (s, r);
val pop = fn : 'a stack -> 'a stack * 'a
- pop it;
val it = (empty,{address="Cambridge",name="Peter Robinson"})
  : {address:string, name:string} stack * {address:string, name:string}
- #1 it;
val it = empty : {address:string, name:string} stack
- pop it;
uncaught exception nocando
-
```

The `#1` operator picks the first element of a tuple rather like selecting a field from a record.

A client using these routines could catch the exception simply by following the invocation of `pop` by a clause `handle nocando => ...` where the ellipses denote an appropriate expression to be returned in this case, which would have to have the same type as the normal return from `pop`. An example of this will be given later.

3.3 A workshop database

Finally, consider a type structure for a database to process people attending a workshop. The main record structure could be roughly as for the names and addresses above, but additional information is needed for particular classes of person. This is most easily handled by defining a new property type together with a function to yield a text string explaining the property:

```
- datatype property = presenter of (string * string) | chair of string | ou;
datatype  property
  con chair : string -> property
  con ou : property
  con presenter : string * string -> property
- fun text (presenter (s, t)) = "Presenting " ^ s ^ " at " ^ t
=   | text (chair (s)) = "Chairing " ^ s
=   | text ou = "from Open University";
val text = fn : property -> string
-
```

Pattern matching is used in the `text` function to distinguish the different variants of the `property` type, and `^` is the string concatenation operator.

A stack of properties could then be attached to each person attending the workshop:

```
- empty;
val it = empty : 'a stack
- push (it, presenter ("From ML to M3", "15:50"));
val it = push (empty,presenter ("From ML to M3","15:50")) : property stack
- push (it, ou);
val it = push (push (empty,presenter #),ou) : property stack
- fun props ps =
=   let val (s, p) = pop ps
=   in (text p) ^ "\n" ^ props s end
=   handle nocando => "";
val props = fn : property stack -> string
- props it;
```

```
val it = "from Open University\nPresenting From ML to M3 at 15:50\n" : string
- print it;
from Open University
Presenting From ML to M3 at 15:50
val it = () : unit
-
```

Here an empty stack is created and then two properties (the `presenter` property with two field values and the unparameterised `ou` property) pushed onto it. The `props` function concatenates all the proeprties on the stack into a single string. Note how the exception for an empty stack is caught and used to return an empty string and how the result from `pop` is split into its components with a nested `let` clause. `"\n"` indicates the new line character, as is shown when the final string is printed.

3.4 Evaluation

These examples should have given a the general flavour of ML, but what is its real rôle? It certainly meets all the criteria for an initial teaching language, but it has much broader uses than that. ML has been widely used for research work on theorem proving and, in particular, for work on formal verification of hardware and software. This is now moving into industrial projects and full commercially supported implementations are available on small PCs as well as professional workstations, although development environments are still rather limited.

The core of the language is very simple and can be defined in just a few pages; however, it is also quite appropriate to use it for large projects. Its mathematical basis gives it great coherence and uniformity. The strong type checking and abstract data types shown in the examples are obviously of value in large systems and ML also supports a mechanism for hiding internal information in packages, revealing only a chosen set of definitions through a signature. This facilitates the orderly construction and maintenance of large systems consisting of many, separately compiled files of code.

The language lends itself to formal specification and analysis, but is not so well suited for some applications. For example, commercial data processing does not fit well with a functional style of programming (although spreadsheets have been written in ML).

Although this was deliberately omitted from the examples above, it is possible to define mutable variables and for functions to have side effects, but these are not comfortable within the language. Garbage collection and an interpreted implementation are not obviously suitable for real-time systems either, although there are now compilers generating efficient code and modern garabage collectors are not as disruptive as their predecessors.

4 Modula-3

Modula-3 is descended from Pascal [21, 8] via Mesa [12], Cedar [9], Modula-2 [22, 23] and Modula-2+ [17]. It is defined in Greg Nelson's book [14], which also gives a rationale for the language design and gives examples of its novel features in use. There is an introductory textbook [5] and a version of Robert Sedgewick's book on algorithms using Modula-3 [18].

The langauge's design goals are encapsulated in the preface to the *Modula-3 report* [3]:

The goal of Modula-3 is to be as simple and safe as it can be while meeting the needs of modern systems programmers. Instead of exploring new features, we studied the features of the Modula family of languages that have proven themselves in practice and tried to simplify them into a harmonious language. We found that most of the successful features were aimed at one of two main goals: greater robustness, and a simpler, more systematic type system.

Modula-3 descends from Mesa, Modula-2, Cedar, and Modula-2+. It also resembles its cousins Object Pascal, Oberon, and Euclid.

Modula-3 retains one of Modula-2's most successful features, the provision for explicit interfaces between modules. It adds objects and classes, exception handling, garbage collection, lightweight processes (or threads), and the isolation of unsafe features.

The key features of Modula-3 are [14]:

Interfaces: An explicit interface reveals only the public declarations in a module, while allowing other parts of it to be kept private. Each module *imports* the interfaces which it requries and *exports* the interfaces that it implements. The interface can be thought of as a contract between the supplier and client of a library module, specifying (amongst other things) the signatures of its procedures, while deferring until later the exact nature of their implementation. They form a natural part of the design documentation of a large program.

Objects: An object is an abstract data type defined in terms of the operations or *methods* permitted on it. A new object type can be defined as a subtype of an existing type, in which case it *inherits* all the methods of the parent type while possibly adding new methods. It can also *override* the existing methods with alternative implementations having the same signature. (It is also possible to mask an inherited method by a new method with the same name but a different signature, but the obscured method can still be invoked.)

Objects and interfaces are combined in Modula-3 to provide *partially opaque types*, where some of an object's fields are visible in a scope while others are hidden.

Generics: Modula-3 does not provide the full polymorphism of ML, but does allow a module (both interface and implementation) to be parameterised by another module. The generic module acts as a template in which some of the imported interfaces are regarded as formal parameters, bound to actual interfaces when the generic is instantiated. This is effectively a textual operation, undertaken at compilation time.

Threads: Dividing a computation into concurrent processes (or threads of control) is a fundamental method of separating concerns. In particular, any program dealing with external activities – filing system, communications network, human users and so on – should not suspend its dealing with all of them while waiting for just one to respond. Separating the program's activities into separate threads which can block individually without affecting the others' execution makes this simpler to deal with.

Safety: Many of the problems with low-level langauges such as C arise through accidental corruption of a program's code or data after using an invalid array index or performing incorrect address arithmetic. Such programs can be protected (at least from eachother) by placing them in separate address spaces, but this may limit performance. It is better to check for incorrect

behaviour and handle it gracefully, rather than allow the program to continue unpredictably.

Garbage collection: A particular unsafe run-time error is to free a data structure still referred to by dangling pointers. Alternatively, storage leaks caused by failure to free unreachable structures cause an executing program's data to grow without bound. Both of these problems can be resolved by tracing references and recovering redundant space by automatic garbage collection.

Exceptions: Another class of unsafe error arises when procedures report errors by returning special values, which are too easily left unchecked by the programmer. Exceptions allow such out-of-band results to be returned and checked with very low overhead in the normal, error-free case, while making the behaviour clear in the abnormal case.

Type system: Modula-3 is strongly typed and particular attention has been paid to making the type system uniform. In particular, a sub-type relation is defined and used to specify assignment compatibility and inheritance rules; the type of every expression can be determined from its constituents independently of its use and there are no automatic type conversions.

Simplicity: C.A.R. Hoare has suggested that as a rule of thumb a language is too complicated if it can't be described precisely and readably in 50 pages. The designers of Modula-3 elevated this to a design principle, which they only narrowly failed to achieve.

As with ML, the language will be illustrated by a number of examples and then discussed.

4.1 Big numbers

Again, arbitrarily large natural numbers will be stored as a list of digits to the base 10. However, in Modula-3 the list is constructed explicitly with pointers rather than being manipulated directly within the language as in ML.

The Modula-3 compiler operates on complete modules, so it useful to see the entire program at once:

```
MODULE Main;

IMPORT Char, Stdio, Text, Wr;

CONST Base = 10;

TYPE
  BigNum = REF RECORD
                 digit: INTEGER;
                 rest : BigNum    := NIL;
               END;

PROCEDURE Create (i: INTEGER): BigNum =
  BEGIN
    IF i = 0 THEN
      RETURN NIL
    ELSE
      RETURN
        NEW (BigNum, digit := i MOD Base, rest := Create (i DIV Base));
```

```
    END;
  END Create;

PROCEDURE Add (a, b: BigNum; carryIn := 0): BigNum =
  BEGIN
    IF a = NIL THEN
      IF carryIn > 0 THEN
        RETURN Add (b, Create (carryIn))
      ELSE
        RETURN b
      END;
    ELSIF b = NIL THEN
      IF carryIn > 0 THEN
        RETURN Add (a, Create (carryIn))
      ELSE
        RETURN a
      END;
    ELSE
      WITH d = a.digit + b.digit + carryIn DO
        RETURN NEW (BigNum, digit := d MOD Base,
                    rest := Add (a.rest, b.rest, d DIV Base));
      END;
    END;
  END Add;

PROCEDURE ToText (b: BigNum; first := TRUE): TEXT =
  BEGIN
    IF b = NIL THEN
      IF first THEN RETURN "0" ELSE RETURN "" END
    ELSE
      RETURN ToText (b.rest, FALSE)
               & Text.FromChar (VAL (ORD ('0') + b.digit, CHAR))
    END;
  END ToText;

PROCEDURE FromText (t: TEXT): BigNum =
  VAR b: BigNum := NIL;
  BEGIN
    FOR i := 0 TO Text.Length (t) - 1 DO
      WITH ch = Text.GetChar (t, i) DO
        IF ch IN Char.Digits THEN
          b := NEW (BigNum, digit := ORD (ch) - ORD ('0'), rest := b);
        END;
      END;
    END;
    RETURN b;
  END FromText;

VAR bi := Create (1);

BEGIN
  FOR i := 1 TO 100 DO
    bi := Add (bi, bi);
    Wr.PutText (Stdio.stdout, ToText (bi) & "\n");
```

```
  END;
  Wr.Close (Stdio.stdout);

END Main.
```

The first observation is that there is a lot more boiler-plate text wrapped round this program than is need for its equivalent in ML. First, there is a heading identifying the module; in fact this has the special name `Main` identifying it as the main program. Secondly, there are explicit `IMPORT` requests for separately compiled modules used by this program. The definitions in the corresponding interfaces are made available in the ensuing scope. Thirdly, all the type information is explicit; every detail of the `BigNum` is specified as a pointer to a record with two fields. A small detail worth noting is that the fields can have default values, initialised whenever a record is allocated. In this case the tail pointer is set to default to `NIL`.

As noted above, the type of an expression is computed from its constituents. In this example, the type of the constant `Base` will be inferred to be an integer, and this tested for compatibility wherever it is used. This can often be done by the compiler, but may require a run-time check.

The bulk of the program is taken up with the definition of four procedures which should be fairly self-explanatory. The `NEW` procedure allocates space on the heap, returning a pointer. Its first argument is a reference type and any further arguments specify initial values for fields in the record referred to. The third argument to the `Add` procedure has a default value of 0; this avoids the need for the auxiliary `doadd` function used in the ML equivalent above. Otherwise, the procedure is roughly equivalent to the ML, eccept that the pattern matching of arguments is replaced by explicit testing. Conversions between big numbers and Modula-3's built-in `TEXT` type make use of utility routines in the standard `Char` and `Text` interfaces.

Finally, the body of the module creates a big number with value 1 and adds it to itself repeatedly, printing out values from 2^1 to 2^{100}. Incidentally, the repeated assignment to the variable `bi` and the various manipulations of the `TEXT` type will result in the generation of large amounts of unreachable heap storage which will be recovered by the garbage collector.

4.2 A stack of records

A name and address record could be defined as a type in Modula-3 as follows:

```
TYPE Record = RECORD
                name,
                address: TEXT := "";
              END;
```

Both fields are of the built-in `TEXT` type and have default values of empty strings. The langauge is case sensitive, so there is no confusion between the user-defined `Record` type and the built-in type constructor `RECORD`.

It is common practice to have a separate interface for each abstract data type, so a better definition might be to create a separate file:

```
INTERFACE Record;

TYPE T = RECORD
           name,
           address: TEXT := "";
         END;

END Record.
```

It is conventional to call the main type in an interface `T` and it is then referred to in other modules in qualified form as `Record.T`.

A stack of these records could then be constructed using a linked list as follows:

```
MODULE Main;

IMPORT Record, Stdio, Wr;

TYPE Stack = REF RECORD
                  item: Record.T;
                  next: Stack := NIL;
                END;

EXCEPTION NoCanDo;

PROCEDURE Push (VAR s: Stack; r: Record.T) =
  BEGIN
    s := NEW (Stack, item := r, next := s);
  END Push;

PROCEDURE Pop (VAR s: Stack): Record.T RAISES {NoCanDo} =
  BEGIN
    IF s = NIL THEN RAISE NoCanDo
    ELSE
      VAR r := s.item;
      BEGIN
        s := s.next;
        RETURN r;
      END;
    END;
  END Pop;

VAR stack: Stack := NIL;

BEGIN
  Push (stack, Record.T {name := "Peter Robinson", address := "Cambridge"});
  Push (stack, Record.T {name := "Mark Woodman", address := "Milton Keynes"});
  TRY
    LOOP Wr.PutText (Stdio.stdout, Pop (stack) .name & "\n") END;
  EXCEPT
    NoCanDo => Wr.PutText (Stdio.stdout, "That's all folks!\n");
  END;
  Wr.Close (Stdio.stdout);

END Main.
```

This is fairly standard imperative programming. Both the `Push` and `Pop` procedures take the stack as a variable argument, implying call-by-reference, and allowing them to have side effects, modifying the contents of the stack. A more functional approach would bee to return the modified stack, as in the ML example above. The initial stack is empty, indicated by a `NIL` value. `Push` is then invoked on it with a couple of literal record constructors to insert some data. The `Pop` procedure raises an exception when passed an empty stack as its argument and this is used in the main body of the program to terminate an otherwise infinite loop. When run, this

program would print out the two names on the stack, together with the message *That's all folks!*

This program can be refined in three ways: the stack can be made generic, capable of stacking elements of any type, it can be made into an abstract data type as an object with *push* and *pop* methods, and the implementations of these methods can be hidden. The finished result would consist of several further files. In addition to the `Record` interface above, there is a generic stack interface:

```
GENERIC INTERFACE Stack (Value);

EXCEPTION Empty;

TYPE
  Public = OBJECT METHODS
             push (v: Value.T);
             pop (): Value.T RAISES {Empty};
           END;

  T <: Public;

END Stack.
```

This has the formal parameter `Value` which will be instantiated to an actual interface name later; it can be thought of as a further imported interface. The actual interface will have to provide a definition for a type `T` since this interface refers to `Value.T`. For this application the `Record` interface defined above will be suitable, but so would many others including, for example, the standard `Text` interface.

The type declarations achieve the second and third refinements. `Public` is an object type with two methods whose signatures are given. The actual procedures supplied later to implement these methods all have an additional first argument which identifies the instance on which they are being invoked.

`T` is simply defined to be a specialisation of `Public`; it will provide the same methods but their implementations are hidden and extra data fields may be added to the object. This is an example of an *opaque type* in Modula-3 – there is only a *partial revelation* of `T`.

This would be accompanied by a generic implementation:

```
GENERIC MODULE Stack (Value);

TYPE
  List = REF RECORD
               item: Value.T;
               next: List := NIL;
             END;

REVEAL
  T = Public BRANDED OBJECT
                       list: List := NIL;
                     OVERRIDES
                       push := Push;
                       pop := Pop;
                     END;

PROCEDURE Push (self: T; value: Value.T) =
  BEGIN
```

```
    self.list := NEW (List, item := value, next := self.list);
  END Push;

PROCEDURE Pop (self: T): Value.T RAISES {Empty} =
  BEGIN
    IF self.list = NIL THEN RAISE Empty
    ELSE
      VAR v := self.list.item;
      BEGIN
        self.list := self.list.next;
        RETURN v;
      END;
    END;
  END Pop;

BEGIN
END Stack.
```

This more-or-less follows the earlier, direct program with the important addition of a revelation of the implementation of `T` within the scope of this module. In fact, `T` is declared to be a specialisation of `Public`, it is `BRANDED` to make its type unique, it has a private data field to hold the linked list storing the contents of the stack, and it overrides the (null) methods of the `Public` type with new implementations.

`Push` and `Pop` have signatures that match the method declarations in `Public`, with the addition of an additional first argument identifying the particular instance of the `Stack.T` object for which they are being invoked. The code is much as before.

The generic interface and implementation are instantiated for the `Record` interface to give specific `RecordStacks`:

```
INTERFACE RecordStack = Stack (Record) END RecordStack.
MODULE RecordStack = Stack (Record) END RecordStack.
```

Finally, these can be tested by a new main program:

```
MODULE Main;

IMPORT Record, RecordStack, Stdio, Wr;

VAR stack := NEW (RecordStack.T);

BEGIN
  stack.push (Record.T {name := "Peter Robinson", address := "Cambridge"});
  stack.push (Record.T {name := "Mark Woodman", address := "Milton Keynes"});
  TRY
    LOOP Wr.PutText (Stdio.stdout, stack.pop () .name & "\n") END;
  EXCEPT
    RecordStack.Empty => Wr.PutText (Stdio.stdout, "That's all folks!\n");
  END;
  Wr.Close (Stdio.stdout);

END Main.
```

which works in exactly the same way as its predecessor.

4.3 A workshop database

As with the ML example, the main record structure is fairly straightforward but a new type is needed to handle the properties of delegates. A stack of these properties could then be included in the main record.

The interface looks like this:

```
INTERFACE Properties;

TYPE
  T = OBJECT METHODS
        text (): TEXT;
      END;

  PresenterP = T OBJECT METHODS
                   init (subject, slot: TEXT): Presenter;
                 END;
  Presenter <: PresenterP;

  ChairP = T OBJECT METHODS
               init (session: TEXT): Chair;
             END;
  Chair <: ChairP;

  OU <: T;

END Properties.
```

A base type, `Properties.T` is defined which has a single method that yields a text string describing the property. Separate public sub-types `PresenterP` and `ChairP` are derived from this for each property to be stored. These have distinct `init` methods that allow their private data fields to be given appropriate initial values. Finally, `Presenter` and `Chair` are partially revealed to be sub-types of these.

The implementation supplies revelations for the opaque types together with implementations of all their methods:

```
MODULE Properties;

REVEAL
  Presenter = PresenterP BRANDED OBJECT
                                   subject,
                                   slot: TEXT;
                                 OVERRIDES
                                   init := PresenterInit;
                                   text := PresenterText;
                                 END;

  Chair = ChairP BRANDED OBJECT
                           session: TEXT;
                         OVERRIDES
                           init := ChairInit;
                           text := ChairText;
                         END;

  OU = T BRANDED OBJECT OVERRIDES
```

```
                  text := OUText;
                END;

PROCEDURE PresenterInit (self: Presenter; subject, slot: TEXT): Presenter =
  BEGIN
    self.subject := subject;
    self.slot := slot;
    RETURN self;
  END PresenterInit;

PROCEDURE PresenterText (self: Presenter): TEXT =
  BEGIN
    RETURN "Presenting " & self.subject & " at " & self.slot;
  END PresenterText;

PROCEDURE ChairInit (self: Chair; session: TEXT): Chair =
  BEGIN
    self.session := session;
    RETURN self;
  END ChairInit;

PROCEDURE ChairText (self: Chair): TEXT =
  BEGIN
    RETURN "Chairing " & self.session;
  END ChairText;

PROCEDURE OUText (self: OU): TEXT =
  BEGIN
    RETURN "from Open University";
  END OUText;

BEGIN
END Properties.
```

This is somewhat verbose, but most of the code is mechanical in nature. Note how the different overriding implementations of the `text` method for the specialisations of `Properties.T` compose the appropriate text strings for the respective properties.

The interface and implementation for a property stack can now be instantiated from the generic stack:

```
INTERFACE PropStack = Stack (Properties) END PropStack.
MODULE PropStack = Stack (Properties) END PropStack.
```

Finally, a test program can use all this:

```
MODULE Workshop EXPORTS Main;

IMPORT Properties, PropStack, Stdio, Wr;

TYPE
  Person = OBJECT
             name: TEXT := "";
             props: PropStack.T;
           METHODS
             init (name: TEXT): Person := Init;
```

```
            text (): TEXT := Text;
          END;

PROCEDURE Init (self: Person; name: TEXT): Person =
  BEGIN
    self.name := name;
    self.props := NEW (PropStack.T);
    RETURN self;
  END Init;

PROCEDURE Text (self: Person): TEXT =
  VAR t := self.name & ":\n";
  BEGIN
    TRY
      LOOP t := t & self.props.pop () .text () & "\n" END;
    EXCEPT
      PropStack.Empty => RETURN t;
    END;
  END Text;

VAR p := NEW (Person) .init ("Robinson");

BEGIN
  p.props.push (NEW (Properties.Presenter) .init ("From ML to M3", "15:50"));
  p.props.push (NEW (Properties.OU));
  Wr.PutText (Stdio.stdout, p.text ());
  Wr.Close (Stdio.stdout);

END Workshop.
```

The same general approach is taken here. A `Person` type is defined whose initialisation method stores a name in the record and sets up an empty stack of properties. Various properties are then pushed onto the stack. These have different types, but they are all sub-types of `properties.T` and so are compatible for pushing onto a `PropStack.T`, and this can be checked by the compiler with no run-time overhead. A text method pops all the properties off the stack and concatenates them into a text string. (A purist might argue that the definition of the `Person` type and its methods should be removed to a separate module, and would probably be right.)

When run, the program would print out:

```
Robinson:
from Open University
Presenting From ML to M3 at 15:50
```

The observant reader will have noticed that calling the `text` method on a `person` removes all their properties; such are the perils of programming with side effects. In practice, a new method would be added to the generic stack to allow iteration over its elements without actually popping them.

4.4 Evaluation

Modula-3 is still a relatively young language and its use is mainly concentrated in universities and commercial research laboratories. The most widely used implementation comes from DEC's Systems Research Center in Palo Alto and runs on

many Unix platforms. An experimental PC implementation is available and further developments, including a GNU implementation, should be available presently. Several hundred library modules are freely distributed. These include a general toolkit for the X window system [11, 10], a more specialised interface toolkit [2] and a system for building graphical user interfaces [1]. There are also three different implementations of *network object* systems for writing distributed programs.

The latest DEC SRC implementation uses the GNU back-ends which makes it reasonably efficient and highly portable. However, programs tend to be rather large, not least because of the run-time library needed to manage threads, garbage collection and so on, although shared libraries alleviate this where the operating system permits it.

The langauge is not suitable for teaching as a first language: it is too big, too much knowledge about the environment is needed, the turnround for compiling and linking is too slow, debugging facilities are primitive and so on. However, it serves very well as a vehicle for exploring programming techniques in a conventional data structures and algorithms course, for teaching concurrency and programming language design, for looking at compilation issues and for practical software engineering. It is excellent for students' final year projects where the modules facilitate the management of a large system and the extensive libraries can be exploited. The easy construction of graphical user interfaces to Modula-3 programs is particularly attractive.

The same considerations mean that Modula-3 is an excellent vehicle for large commercial projects, and it has been used for a number of substantial systems running to hundreds of thousands of lines of code at DEC and Xerox. These include operating system work, real-time communications and straightforward applications. At the moment, the programming environment and project management tools tend to be constructed from standard Unix tools, which is a disadvantage, although work on more conventional development systems is being undertaken.

The goal of Modula-3 was to be as simple and safe as it could be while meeting the needs of modern systems programmers. This goal has substantially been met. Exceptions, threads and garbage collection all help to avoid common errors in programs, particularly those that will have to run continuously for long periods. Strong typing extended through interfaces across separately compiled modules is a proven technique for building large systems and assists the reuse of existing code, especially in conjunction with generics. The use of objects and partial revelations gives control over levels of abstraction in libraries, allowing compromises to be made between modularity and efficiency as the need dictates.

Some might question Modula-3's restriction to single inheritance where each object type is derived from a single parent type. This restriction gives much cleaner semantics to the language, which is important for formal verification, and also admits more efficient implementations. Moreover, there seem to be very few practical cases where multiple inheritance is actually of significant value.

The type system for Modula-3 was to some extent dictated by considerations of type safety in distributed systems. In particular, it was designed to allow structured values to be passed between programs safely. As a consequence it is also possible to save data structures in a filing system and recover them in a type-safe way through a system of *pickles* which provides the basis for programming persistent systems.

5 Conclusion

This paper has presented a set of guidelines for selecting initial languages for teaching programming and has suggested that two languages — ML and Modula-3 — are particularly well suited to the task. The main features of these two languages

were then presented and illustrated through extended examples, and the languages evaluated.

ML has been used as the initial teaching language for the Computer Science Tripos at Cambridge since 1987 and this has been followed by Modula-3 since 1990. 16 hours of lectures are budgeted for ML, but some of the more advanced features of the language (exceptions and modules) are omitted. These are accompanied by a series of graduated practical classes and exercises. The whole of Modula-3 is taught in two series of 12 lectures each, again accompanied by practical work [16].

The use of practical classes is important. Programming is a practical skill and can not be taught purely through lectures; carefully supervised classes are vital if the students are to develop reasonable programming style as well as technical familiarity with the language. Each language has its own idiomatic usage which can be illustrated through examples in lectures (and, indeed, in this paper), but this can only be fully appreciated when it is being applied to a new problem. Lectures on dynamics can not teach you how to ride a bicycle!

Moreover the approach appears to be popular with the students. ML and, particularly, Modula-3 are the langauges of choice for final-year projects. There also appears to be a correlation between the use of these languages and higher marks for the project. It would appear that all desiderata for systems programming are genuinely valuable and a language like Modula-3 is conducive to higher productivity.

The same considerations suggest that ML and Modula-3 would be of value in commercial projects. ML has already found favour where mathematical analysis is deemed to be important and its interpreted implementation lends itself to rapid prototyping of systems. However, Modula-3 is still mainly used only in academic and research environments. This seems odd: Modula-3 meets the requirements of a langauge like Ada but avoids unnecessary complexity and achieves greater coherence. Further competition comes from C++, which is widely used but shares the worst aspects of C's syntax and unsafe features. Indeed, a common complaint amongst systems programmers is that company policy dictates the use of C++ when their professional judgement would be to use a safer language such as Modula-3. Of course, it is possible to write correct and maintainable programs in any language, but a language like Modula-3 makes it more natural. Fortunately, students who have learned the idiom of Modula-3 are then able to approach other languages, learn them and use them in a disciplined way. Perhaps the availability of better supported implementations will allow more rational policies to be adopted and such contortions avoided.

References

[1] Marc Brown and James Meehan. The FormsVBT reference manual. Technical report, DEC Systems Research Center, March 1993.

[2] Marc Brown and James Meehan. VBTkit reference manual. Technical report, DEC Systems Research Center, March 1993.

[3] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. The Modula-3 report (revised). Technical report, DEC Systems Research Center and Olivetti Research Center, November 1989.

[4] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF.* Springer-Verlag, 1978.

[5] Sam Harbison. *Modula-3.* Prentice-Hall, Englewood Cliffs, NJ 07632, 1992.

[6] Robert Harper. Introduction to Standard ML. Technical Report ECS-LFCS-86-14, University of Edinburgh, Laboratory for Foundations of Computer Science, November 1986.

[7] Robert Harper, David MacQueen, and Robin Milner. Standard ML. Technical Report ECS-LFCS-86-2, University of Edinburgh, Laboratory for Foundations of Computer Science, March 1986.

[8] Kathleen Jensen and Niklaus Wirth. *Pascal user manual and report.* Springer-Verlag, 1975.

[9] Butler Lampson. A description of the Cedar language. Technical Report CSL-83-15, Xerox PARC, 1983.

[10] Mark Manasse and Greg Nelson. Trestle tutorial. Technical Report 69, DEC Systems Research Center, May 1921.

[11] Mark Manasse and Greg Nelson. Trestle reference manual. Technical Report 68, DEC Systems Research Center, December 1991.

[12] James Mitchell, William Maybury, and Richard Sweet. Mesa language manual. Technical Report CSL-79-3, Xerox PARC, April 1979.

[13] Colin Myers, Chris Clack, and Ellen Poon. *Programming with Standard ML.* Prentice-Hall, 1993.

[14] Greg Nelson, editor. *Systems Programming With Modula-3.* Prentice-Hall, Englewood Cliffs, NJ 07632, 1991.

[15] Laurence Paulson. *ML for the working programmer.* Cambridge University Press, 1991.

[16] Peter Robinson. Modula-3 in an undergraduate Computer Science course. *Proc. 2nd International Modula-2 Conference*, September 1991.

[17] Paul Rovner, Roy Levin, and John Wick. On extending Modula-2 for building large integrated systems. Technical Report 3, DEC Systems Research Center, January 1985.

[18] Robert Sedgwick. *Algorithms in Modula-3.* Addison-Wesley, 1993.

[19] Jeffrey D Ullman. *Elements of ML programming.* Prentice-Hall, 1993.

[20] Åke Wikström. *Functional programming using Standard ML.* Prentice-Hall, 1987.

[21] Niklaus Wirth. The programming language Pascal. *Acta Informatica*, 1(1):35–63, 1971.

[22] Niklaus Wirth. Modula-2. Technical Report 36, ETH Zürich, March 1980.

[23] Niklaus Wirth. *Programming in Modula-2.* Springer-Verlag, 4 edition, 1988.