Formal program development in Extended ML for the working programmer[*]

Donald Sannella[†]

Abstract

Extended ML is a framework for the formal development of programs in the Standard ML programming language from high-level specifications of their required input/output behaviour. It strongly supports the development of modular programs consisting of an interconnected collection of generic and reusable units. The Extended ML framework includes a methodology for formal program development which establishes a number of ways of proceeding from a given specification of a programming task towards a program. Each such step gives rise to one or more proof obligations which must be proved in order to establish the correctness of that step. This paper is intended as a user-oriented summary of the Extended ML language and methodology. Theoretical technicalities are avoided whenever possible, with emphasis placed on the practical aspects of formal program development. An extended example of a complete program development in Extended ML is included.

1 Introduction

The ultimate goal of work on program specification is to establish a practical framework for the systematic production of correct programs from requirements specifications via a sequence of verified-correct development steps. Such a framework should ideally have a number of desirable characteristics. Among other important issues are the following:

Formality: The outcome of the program development process is guaranteed to be correct with respect to the original requirements specification if development steps are proved correct in some formal logical calculus which is sound with respect to a complete mathematical semantics of the specification and programming languages involved. Any hedge on this strictly formal point of view invalidates all guarantees.

Methodology: Given a specification of a programming task, it is helpful if the framework provides some form of direction in working towards a solution. One possibility is if the framework sets forth a certain number of kinds of development steps which apply to specifications of a given form, together with the conditions which must be established in order to guarantee correctness. Coming up with development steps is a difficult creative task and standardized methods of making progress are very important in reducing this difficulty to a manageable level.

Modularity: Large programs should be built in a modular fashion from small and relatively independent program units, and the framework should support such an approach. Apart from the advantages which this gives in allowing large programming tasks to be broken into a number of smaller and more manageable separate tasks, this allows previously-developed programs and components of such programs to be reused in other programs. One may imagine formal program development becoming almost practically feasible in spite of its great unitary cost because of the potential of spreading this cost over many different projects.

[*]Appeared in *Proc. 3rd BCS/FACS Workshop on Refinement*, Hursley Park, 1990. Springer Workshops in Computing, 99–130 (1991).

[†]Laboratory for Foundations of Computer Science, Department of Computer Science, Edinburgh University.

Machine support: The eventual practical feasibility of systematic program development by stepwise refinement hinges on the availability of computer-aided tools to support various development activities. This is necessary both because of the sheer amount of (mostly clerical) work involved and because of the need to avoid the possibility of human error.

This leaves aside many questions, including: How may the original requirements specification be formulated so as to ensure that it accurately expresses the needs of the customer? How should choices between some number of possible development paths be made? Such issues are no less important than those mentioned above, but they will not be addressed here.

Extended ML is a framework for the formal development of programs in the Standard ML programming language from high-level specifications of their required input/output behaviour. Extended ML is a completely formal framework with a very extensively-developed mathematical basis in the theory of algebraic specifications. It strongly supports the development of modular programs consisting of an interconnected collection of generic and reusable units. The Extended ML framework includes a methodology for formal program development which establishes a number of ways of proceeding from a given specification of a programming task towards a program. Each such step (modular decomposition, etc.) gives rise to one or more proof obligations which must be proved in order to establish the correctness of that step. On the minus side, at present Extended ML can only be used to develop programs written in a small purely functional subset of Standard ML, and a computer-aided system to support program development is still in the design stage.

The Extended ML language is a wide-spectrum language which encompasses both specifications and executable programs in a single unified framework. It is a simple extension of the Standard ML programming language in which axioms are permitted in module interfaces and in place of code in module bodies. This allows all stages in the development of a program to be expressed in the Extended ML language, from the initial high-level specification to the final program itself and including intermediate stages in which specification and program are intermingled.

Formally developing a program in Extended ML means writing a high-level specification of a generic Standard ML module and then refining this specification top-down by means of a sequence (actually, a tree) of development steps until an executable Standard ML program is obtained. The development has a tree-like structure since one of the ways to proceed from a specification is to decompose it into a number of smaller specifications which can then be independently refined further. In programming terms, this corresponds to implementing a program module by decomposition into a number of independent sub-modules.

This paper is intended as a user-oriented summary of the Extended ML language and methodology. Theoretical technicalities are avoided whenever possible, with emphasis placed on the practical aspects of formal program development. Some of the details (mainly those concerned with *behavioural equivalence*) are glossed over in the interests of simplicity. Section 3 provides an overview of the Extended ML language, preceded by a brief review of the Standard ML programming language on which it is based in Section 2. Section 4 presents a methodology for developing Standard ML programs from Extended ML specifications by stepwise refinement. An extended example of a complete program development in Extended ML is included in Section 5; this is the most important section of the paper for the reader who merely wishes to get a taste of what formal program development is like. Finally, Section 6 concludes with some remarks about some potential areas of future progress. Readers who are interested in the theory which underlies Extended ML should consult [ST 89]; among other things, this explains in detail why the formal program development process outlined here is guaranteed to yield a program which is correct with respect to the original specification of requirements.

2 An overview of Standard ML

The aim of this section is to briefly review the main features of the Standard ML programming language which are relevant to Extended ML, in an attempt to make this paper self-contained. A complete description of the language appears in [HMM 86], and a complete formal semantics is in

[HMT 89] which also includes historical comments on the development of the language. The features of Standard ML are introduced at a more tutorial level in [Wik 87] (core language only), [Tofte 89] (mainly module language), [Har 89], and [Reade 89].

Standard ML consists of two sub-languages: the Standard ML "core language" and the Standard ML "module language". The core language provides constructs for programming "in the small" by defining a collection of types and values of those types. Programs written in the core language look very similar to programs in Hope [BMS 80], Miranda [BW 88] or Haskell [HW 89]. The module language provides constructs for programming "in the large" by defining and combining a number of self-contained program units. These sub-languages can be viewed as more or less independent since there are relatively few points of contact between the sub-languages. A similar modularization mechanism could be added to other programming languages; see [SW 87] for the design of an ML-style module system for Prolog.

2.1 The Standard ML core language

Standard ML is a strongly typed language. Every expression has a type which is inferred automatically by the Standard ML compiler. Expressions are required to obey the typing rules before being evaluated, and a well-typed expression is guaranteed to produce no run-time type errors. A new type is defined by giving its name and listing the ways in which values of that type may be constructed from values of other types. For example, the type of integer sequences may be defined as follows:

```
datatype sequence =
    nil
  | cons of int * sequence
```

(`int` is the type of integers, which is built-in). A typical value of type `sequence` is then:

```
cons(3,cons(4,cons(0,cons(3,nil))))
```

Conceptually, every value in Standard ML is represented as a term consisting of a *constructor* applied to a number of sub-terms, each of which in turn represents another value. In the above definition, `nil` is a nullary constructor and `cons` is a binary constructor (of type `int * sequence -> sequence`). Constructor functions are uninterpreted; they just construct. There is no need to define a lower-level representation of sequences in terms of arrays or pointers. Note that type definitions may be recursive, as in the above example. The type of integers may be viewed as if it were defined as follows:

```
datatype int = ... | ~3 | ~2 | ~1 | 0 | 1 | 2 | 3 | ...
```

Other built-in types include booleans (`bool`) and character strings (`string`), and there are a few built-in functions such as `<= : int * int -> bool` (less than or equal), `size : string -> int`, and `not : bool -> bool`. The function `=` : t `*` t `-> bool` (and its negation, `<>`) is automatically provided for any user-defined type t.[1] It is possible to give new names to existing types and type expressions:

```
type age = int
```

This creates an additional name `age` for the existing type `int`.

Functions are defined by a sequence of one of more equations, each of which specifies the value of the function over some subset of the set of possible argument values. This subset is described by a *pattern* (a term containing constructors and variables only, without repeated variables) on the left-hand side of the equation. The pattern is thereby used for case selection and variable binding. For example:

```
fun length(nil) = 0
  | length(cons(a,s)) = 1 + length(s)
```

[1]This is the case for the subset of the Standard ML core language used here, but not in general.

This defines a function `length : sequence -> int` (this type is inferred automatically by the Standard ML compiler). One way of viewing such a definition is as a set of rewrite rules:

```
length(cons(3,cons(4,cons(0,cons(3,nil)))))
                  ⇒  1+length(cons(4,cons(0,cons(3,nil))))
                  ⇒  1+(1+length(cons(0,cons(3,nil))))
                  ⇒  1+(1+(1+length(cons(3,nil))))
                  ⇒  1+(1+(1+(1+length(nil))))
                  ⇒  1+(1+(1+(1+0)))
                  ⇒  4
```

Function definitions are often recursive, as in this example; of course, this defines a terminating function only if the recursion is well-founded in the usual sense. The patterns on the left-hand side of equations should normally be disjoint and should exhaust the possibilities given in the definition of the argument type(s). Constants may also be defined:

```
val three_ones = cons(1,cons(1,cons(1,nil)))
```

Such definitions may not be recursive. Constants and functions are referred to collectively as *values*.

Values may be defined in terms of other values, of course. In the following example, a function to sort a sequence into ascending order is defined using an auxiliary function which inserts an integer into a ordered list:

```
fun insert(a,nil) = cons(a,nil)
  | insert(a,cons(b,s)) = if a<=b then cons(a,cons(b,s))
                          else cons(b,insert(a,s))

fun sort nil = nil
  | sort(cons(a,s)) = insert(a,sort s)
```

This defines `insert : int * sequence -> sequence` and `sort : sequence -> sequence`. Evaluating the expression:

```
sort(cons(11,cons(5,cons(8,nil))))
```

will yield the result:

```
cons(5,cons(8,cons(11,nil))) : sequence
```

The Standard ML core language includes an assortment of other features, but we will only be concerned with simple type definitions and value definitions such as those in the examples above. Features which we will not use include: polymorphic types (it is possible to define sequences of values of type α, for arbitrary α, and define functions over such types which work for any α); higher-order functions and functions as first-class citizens (which means that values can have types like `(int -> int) -> (sequence -> sequence)` and functions can be embedded in data structures); and imperative features (references and exceptions). We restrict ourselves to this simple "pure" subset of the Standard ML core language because adequate (algebraic-style) formal foundations for the additional features are not yet available. To keep things simple we will make the additional assumption throughout this paper that all functions we deal with are total.

2.2 The Standard ML module language

The Standard ML module language provides mechanisms which allow large Standard ML programs to be structured into self-contained program units with explicitly-specified interfaces. Under this scheme, interfaces (called *signatures*) and their implementations (called *structures*) are defined separately. Every structure has a signature which gives the names of the types and values defined in the structure. Structures may be built on top of existing structures, so each one is actually a *hierarchy* of structures, and this is reflected in its signature. Components of structures are accessed using qualified names such as `A.B.n` (referring to the component `n` of the structure component `B` of the structure `A`). *Functors* are "parameterized" structures; the application of a functor to a structure yields a structure. A functor has an input signature describing structures to which it may be applied, and an output signature describing the structure which results from such an application. It is possible, and sometimes necessary to allow interaction between different parts of a program, to declare that certain substructures (or just certain types) in the hierarchy are identical or *shared*. This issue will be discussed later in this section.

An example of a simple modular program in Standard ML is given below. This generalizes the program above for sorting a sequence of integers, by allowing a sequence of values of arbitrary type to be sorted provided an order relation is supplied.

```
signature PO =
   sig
       type elem
       val le : elem * elem -> bool
   end

signature SORT =
   sig
       structure Elements : PO
       datatype sequence =
           nil
         | cons of Elements.elem * sequence
       val sort : sequence -> sequence
   end

functor Sort(X : PO) : SORT =
   struct
       structure Elements = X
       datatype sequence =
           nil
         | cons of Elements.elem * sequence
       fun insert(a,nil) = cons(a,nil)
         | insert(a,cons(b,s)) = if Elements.le(a,b) then cons(a,cons(b,s))
                                 else cons(b,insert(a,s))
       fun sort nil = nil
         | sort(cons(a,s)) = insert(a,sort s)
   end
```

This defines a functor called `Sort` which may be applied to any structure matching the signature `PO`, whereupon it will yield a structure matching the signature `SORT`. In order for the definition of `Sort` to be correctly typed, the body of `Sort` must define a structure which contains: a substructure called `Elements` which matches `PO`; a type called `sequence` having constructors called `nil` and `cons` with the types given, and no other constructors; and a function called `sort` with the type given. The definition of `Sort` is indeed correctly typed, and this is determined automatically at compile time.

We can define a structure of signature `PO` and apply `Sort` to this structure as follows:

```
structure IntPO : PO =
   struct
       type elem = int
       val le = op <=
   end

structure SortInt = Sort(IntPO)
```

Now, `SortInt.sort` may be applied to the sequence

```
SortInt.cons(11,SortInt.cons(5,SortInt.cons(8,SortInt.nil)))
```

to yield

```
SortInt.cons(5,SortInt.cons(8,SortInt.cons(11,SortInt.nil))) : SortInt.sequence
```

Since the function `insert` is not mentioned in the output signature `SORT`, it is considered local to the body of `Sort` and does not appear in the structure `SortInt`. The body of `Sort` makes no reference to other functors but of course it is possible to define new functors by building on top of existing functors. For example, it would be possible to isolate the definition of sequences and functions on sequences in a functor, and then refer to this functor in the body of `Sort`.

The `datatype` declaration in `SORT` constrains the type `sequence` defined in the body of `Sort` to have constructors called `nil` and `cons`, and no other constructors. We can use this information outside the body of the functor to define functions over this type by case analysis, and to test values of this type for equality. The following defines a function `sum` of type `SortInt.sequence -> int`:

```
fun sum(SortInt.nil) = 0
  | sum(SortInt.cons(a,s)) = a + sum(s)
```

Although this is a very convenient notation, it relies on the fact that `SortInt.sequence` is defined as a `datatype` with a known set of constructors. It is sometimes desirable to hide such information about the representation of a type, keeping it local to the body of the functor which defines the type; this permits the representation to be changed for reasons such as time or space efficiency without changing other code which makes use of the type. The following version of `SORT` mentions the values `cons` and `nil`, but does not require them to be constructors:

```
signature SORT' =
   sig
       structure Elements : PO
       type sequence
       val nil : sequence
       val cons : Elements.elem * sequence -> sequence
       val sort : sequence -> sequence
   end
```

If the definition of `Sort` were changed to use this as output signature, then the above definition of `sum` would not be well-formed, even if the type `sequence` were defined as a `datatype` as above. In fact, without some additional discriminator and destructor functions (such as `null`, `hd` and `tl`) it would be impossible to define `sum` outside the body of `Sort`. A variation on the above would be to replace the declaration of `sequence` in `SORT'` with the line:

```
       eqtype sequence
```

This exports the equality function `= : sequence * sequence -> bool` (which would be hidden in the case of `SORT'`) but not the constructors.

Multi-argument functors are treated as single-argument functors in which the input signature requires a structure with multiple substructures. For example, here is a functor which takes two structures matching `PO` and produces another structure matching `PO` (the lexicographic ordering on pairs):

```
functor Lexicographic(structure X : PO
                      structure Y : PO) : PO =
   struct
       type elem = X.elem * Y.elem
       fun le((x,y),(x',y')) = if X.le(x,x')
                               then if X.le(x',x) then Y.le(y,y') else true
                               else false
   end
```

If `IntPO` is defined as above and `BoolPO` is defined as follows:

```
structure BoolPO : PO =
   struct
       type elem = bool
       fun le(true,true) = true
         | le(true,false) = false
         | le(false,b) = true
   end
```

then the functor `Lexicographic` may be applied to these two structures to define an order relation on ⟨`int` $\times$ `bool`⟩-pairs as follows:

```
structure Lex = Lexicographic(structure X = IntPO
                              structure Y = BoolPO)
```

Then `Lex.le((2,true),(2,false))` gives the value `false`.

When multi-argument functors are defined, it is sometimes necessary to declare that certain components of the argument structures are common to both structures. A contrived example is the following:

```
functor Wrong(structure X : PO
              structure Y : PO) : PO =
   struct
       type elem = X.elem
       fun le(a,b) = X.le(a,b) andalso Y.le(a,b)
   end
```

(`andalso` is logical conjunction). The definition of `Wrong` is ill-typed: in the definition of the function `le`, the variables `a` and `b` are required to be of type `X.elem` (because of the first conjunct) and of type `Y.elem` (because of the second conjunct). Some applications of `Wrong` (for example, to a structure in which `X` is `IntPO` and `Y` is like `IntPO` but with the opposite ordering) will be well-typed since `X.elem` and `Y.elem` are the same type, but other applications (for example, to a structure in which `X` is `IntPO` and `Y` is `BoolPO`) will be ill-typed. The input signature of the following functor includes a *sharing constraint* which restricts application to appropriate structures:

```
functor Right(structure X : PO
              structure Y : PO
              sharing type X.elem = Y.elem) : PO =
   struct
       type elem = X.elem
       fun le(a,b) = X.le(a,b) andalso Y.le(a,b)
   end
```

In this example, it was only necessary to require that `X.elem` and `Y.elem` are the same types. It is sometimes necessary to require that whole (sub)structures are the same. For example:

```
functor Strange(structure X : PO
                structure Y : PO
                sharing X = Y) : PO =
   struct
       type elem = X.elem
       fun le(a,b) = X.le(a,b) andalso Y.le(a,b)
   end
```

This functor can only be applied to structures having two identical substructures `X` and `Y`.

It is possible to use sharing constraints to make explicit the fact that parts of the argument structure of a functor are inherited by the result structure. This information can be added to the output signature of the `Sort` functor above as follows:

```
functor Sort'(X : PO) : sig include SORT
                            sharing Elements = X
                        end =
   struct
       structure Elements = X
       datatype sequence =
           nil
         | cons of Elements.elem * sequence
       fun insert(a,nil) = cons(a,nil)
         | insert(a,cons(b,s)) = if Elements.le(a,b) then cons(a,cons(b,s))
                                 else cons(b,insert(a,s))
       fun sort nil = nil
         | sort(cons(a,s)) = insert(a,sort s)
   end
```

The declaration `include SORT` has the same effect as repeating the declarations in the signature `SORT` above. The sharing constraint `sharing Elements = X` asserts that the substructure `Elements` of the result structure is identical to the argument structure.

This example exposes a subtle but important difference between the Standard ML module language and modules as used in Extended ML. In Standard ML and Extended ML, signatures serve both to impose constraints on the bodies of structures/functors and to restrict the information which is made available externally about the types and functions which are defined in structure/functor bodies. In the examples above this was used to hide local functions (such as `insert` in `Sort`) and to hide the fact that certain values are constructors (such as `nil` and `cons` in `SORT'`). In Standard ML, the information passed to the outside world about a structure/functor is taken to be that in its signature(s) augmented by any information about type and structure sharing which can be inferred from the body (sharing *by construction* in [MacQ 86]). Extended ML is more strict: only the information which is explicitly recorded in the signature(s) of a structure/functor is available externally. Thus, any program which is well-typed in Extended ML will be well-typed in Standard ML but not vice versa. This additional strictness is vital to allow parts of a large software system to be developed and maintained independently. The main effect of this is that it is often necessary to include explicit inheritance constraints like the one in `Sort'` above. Without this constraint, the information that the type `Elements.elem` in the structure `Sort'(IntPO)` is the type `int` would be unavailable. (This means that structures in Extended ML are actually *abstractions* in the sense of [MacQ 86], and functors are parameterized abstractions.)

3 The Extended ML wide-spectrum language

This section reviews the main features of the Extended ML specification/programming language. A more complete introduction to the Extended ML language appears in [ST 85]. The version of Extended ML used in this paper is different in certain details from the one presented in [ST 85] but the general motivation and ideas and the overall appearance of specifications remains the same. [SS 89] defines the syntax and some aspects of the semantics of Extended ML, and a complete formal semantics will be forthcoming.

Extended ML is intended as a vehicle for the systematic formal development of programs from specifications by means of individually-verified steps. Extended ML is called a "wide-spectrum" language since it allows all stages in the formal development process to be expressed in a single unified framework, from the initial high-level specification to the final program itself and including intermediate stages in which specification and program are intermingled. The eventual product of the formal development process is a modular program in Standard ML, and thus Standard ML (that is, the "pure" subset of Standard ML described in Section 2) is the executable sub-language of Extended ML. Earlier stages in the development of such a program are incomplete modular programs in which some parts are only specified by means of axioms rather than defined in an executable fashion by means of ML code. This allows more information to be provided in signatures (in the form of axioms specifying properties which are required to hold of any structure matching that signature), and less information to be provided in structure and functor bodies (since axioms are permitted in place of ML code).

In Section 4, a methodology is described for gradually refining such specifications to obtain programs. During the development process it is possible (and indeed normal) to use ML's module facilities to decompose a given programming task into a number of independent subtasks. This is perhaps the most novel aspect of the Extended ML methodology — its main strength lies in the support it provides for program development "in the large". Program development "in the small" is supported as well but the mechanisms provided are not very different from those of other approaches.

In the Standard ML module language, a signature acts as an interface to a program unit (structure or functor) which serves to mediate its interactions with the outside world. The signature of a structure describes the types and values which that structure makes available to the outside world. The output signature of a functor has much the same purpose, while the input signature describes what that functor requires from the outside world in order to function as required. Only those internal details of the structure/functor which are mentioned in its signature are visible to the outside world.[2] The remaining internal details may be modified at any time as long as this externally visible behaviour is maintained.

The information in a signature is sufficient for the use of Standard ML as a programming language, but when viewed as an interface specification a signature does not generally provide enough information to permit proving program correctness (for example). To make signatures more useful as interfaces of structures in program specification and development, we allow them to include axioms which put constraints on the permitted behaviour of the components of the structure. An example of such a signature is the following more informative version of the signature `PO` from the last section:

```
signature PO =
   sig
       type elem
       val le : elem * elem -> bool
       axiom le(x,x)
       axiom le(x,y) andalso le(y,x) => x=y
       axiom le(x,y) andalso le(y,z) => le(x,z)
   end
```

This includes the previously-unexpressible precondition which `IntPO` must satisfy if `Sort(IntPO)` is to behave as expected, namely that `IntPO.le` is a partial order on `IntPO.elem`.

Axioms are expressions of type `bool`. Using such an expression as an axiom amounts to an assertion that the value of the expression is `true` for all values of its free variables. Axioms may be built using connectives such as `not`, `andalso`, `orelse` and `=>` and quantifiers such as `exists` and `forall`, and the function `=` may be used to compare values of any type. This is equivalent to

[2]As mentioned at the end of the last section, this is not quite true in Standard ML but it is true in Extended ML.

using first-order equational logic. Of course, the Standard ML code which is obtained at the end of the program development process will not contain quantifiers or use = except on types which admit equality according to Standard ML. The declaration of a type as a `datatype` amounts in logical terms to a principle of structural induction for that type, together with axioms stating that the values of two constructor terms are equal iff the terms are identical.

Formal specifications can be viewed as abstract programs. Some specifications are so completely abstract that they give no hint of an algorithm (e.g. the specification of the inverse of a matrix A as that matrix A^{-1} such that $A \times A^{-1} = I$) and often it is not clear if an algorithm exists at all, while other specifications are so concrete that they amount to programs (e.g. Standard ML programs, which are just equations of a certain form which happen to be executable). In order to allow different stages in the evolution of a program to be expressed in a single framework, we allow structures to contain a mixture of ML code and non-executable axioms. Functors can include axioms as well since they are simply parameterized structures. For example, a stage in the development of the functor `Sort` in the last section might be the following:

```
functor Sort(X : PO) : sig include SORT
                           sharing Elements = X
                       end =
   struct
       structure Elements : PO = X
       datatype sequence =
           nil
         | cons of Elements.elem * sequence
       fun append(nil,s) = s
         | append(cons(a,s1),s2) = cons(a,append(s1,s2))
       fun member(a:Elements.elem,s:sequence) = ? : bool
       axiom member(a,nil) = false
       axiom member(a,cons(a,s)) = true
       axiom a<>b => member(a,cons(b,s)) = member(a,s)
       fun insert(a:Elements.elem,s:sequence) = ? : sequence
       axiom member(a,insert(a,s))
       axiom insert(a,s) = append(s1,cons(a,s2))
                    => append(s1,s2) = s
                       andalso (member(a1,s1) => Elements.le(a1,a))
                       andalso (member(a2,s2) => Elements.le(a,a2))
       fun sort nil = nil
         | sort(cons(a,s)) = insert(a,sort s)
   end
```

In this functor declaration, the function `sort` has been defined in an executable fashion in terms of `insert` which is so far only constrained by axioms (these axioms refer to other functions which will not be required in the final version). Functions and constants which are not defined in an executable fashion are declared using the special place-holder expression `?` as in the example above. This is necessary in order to declare the type of the function or constant which would normally be inferred from an executable definition by the ML system. The same construct can be used to declare a type when its representation in terms of other types has not yet been selected. It is also useful at the earliest stage in the development of a functor or structure when no body has been supplied:

```
functor Sort(X : PO) : sig include SORT
                           sharing Elements = X
                       end = ?
```

The Extended ML language is the result obtained by extending Standard ML as indicated above. That is, axioms are allowed in signatures and in structures, and the place-holder `?` is allowed in place of the expression (type expression, value expression, or structure expression) on the right-hand side of declarations. Explicit signatures are required in structure declarations and explicit output signatures are required in functor declarations (in Standard ML these are optional) and the use of these signatures in typechecking is somewhat stricter than in Standard ML as discussed at the end of Section 2.2.

The examples above and those in the sequel use the notation of first-order equational logic to write axioms. This choice is rather arbitrary since the formal underpinnings of Extended ML are actually entirely independent of the choice of logic (see [ST 86] for the details; a logic suitable for use is called an *institution* [GB 84]). It is natural to choose a logic which has the Standard ML core language as a subset; this way, the development process comes to an end when all the axioms in structure and functor bodies are expressed in this executable subset.

The role of signatures as interfaces suggests that they should be regarded as descriptions of the externally observable behaviour of structures. Consider the following example:

```
signature OBJ =
   sig
       type object
   end

signature STACK =
   sig
       structure Obj : OBJ
       type stack
       val empty : stack
       val push : Obj.object * stack -> stack
       val pop : stack -> stack
       val top : stack -> Obj.object
       axiom pop(push(a,s)) = s
       axiom top(push(a,s)) = a
   end

functor Stack(O : OBJ) : sig include STACK
                             sharing Obj = O
                         end = ?

structure IntStack : STACK = Stack(struct
                                       type object = int
                                   end)
```

The purpose of the axioms in the signature `STACK` is to specify the behaviour of the functions defined by the functor `Stack`. Any implementation of these functions which satisfies the axioms in `STACK` will be valid. This definition of validity seems reasonable, but it turns out to be too restrictive: for instance, the usual representation of stacks using an array with a pointer to the top element of the stack will be invalid since it does not satisfy the first axiom of `STACK`. The reason why this representation causes no difficulties in practice is that there is no way for an external observer to detect the difference between the stacks `pop(push(`a`,`s`))` and s, since equality on stacks is not provided. This implies that this axiom is not to be taken too seriously. In contrast, in `IntStack` (and other instantiations of `Stack`) it is possible to directly observe the value of `top(push(`a`,`s`))` and compare it with the value of a, so the second axiom of `STACK` must be satisfied by any implementation of `Stack`. In fact, the first axiom cannot be disregarded either since it is possible in `IntStack` to directly observe whether or not the following equations hold (for any values of a and s):

$$\begin{array}{rcl}
\texttt{top(pop(push(}a,s\texttt{)))} & = & \texttt{top(}s\texttt{)} \\
\texttt{top(pop(pop(push(}a,s\texttt{))))} & = & \texttt{top(pop(}s\texttt{))} \\
\texttt{top(pop(pop(pop(push(}a,s\texttt{)))))} & = & \texttt{top(pop(pop(}s\texttt{)))} \\
 & \vdots &
\end{array}$$

All of these are consequences of the first axiom and so one would expect them to hold. So the first axiom is important at least insofar as it gives rise to a large number (in fact, an infinite number) of observable properties.

Because of examples like the one above, validity of implementations is defined in Extended ML in terms of satisfaction of axioms "up to behavioural equivalence" with respect to an appropriate set of "observable types". The details of this may be found in [ST 89]. The proper treatment of this issue is one of the most important facets of the design of Extended ML. However, this complication will be disregarded in this paper in the interests of simplicity of presentation; we will pretend that axioms are to be satisfied "literally", rather than only up to behavioural equivalence. Many examples, including the one in Section 5, do not require the extra generality provided by Extended ML's use of behavioural equivalence and so the language and methodology are still quite useful even when this issue is ignored.

4 The formal program development methodology

The starting point of formal development is a high-level requirements specification of a software system. The concept of a Standard ML functor corresponds to the informal notion of a self-contained software system. A functor may be built by composing other functors and so the scale of such a system may vary from small (like the examples in previous sections) to very large. In Extended ML, a specification of a software system is a functor with specified interfaces. The initial high-level specification will be a functor of the form:

```
functor F(X : SIG) : SIG' = ?
```

where `SIG` and `SIG'` are Extended ML signatures containing axioms. At later stages of development, a functor specification may include a body which is not yet composed of executable code. This is still a specification of a software system, but one in which some details of the intended implementation have been supplied.

We will not be concerned here with the difficult problem of how the initial requirements specification is obtained, or how to check that it accurately reflects the needs of the customer for whom the system is being developed. This is definitely a vital issue which needs a great deal more investigation. We assume here that a formal requirements specification in the form indicated above is provided somehow as a starting point, and ignore the step from the informal requirements of the customer to this formal specification. It is clear, however, that the formal requirements specification is the result of negotiation with the customer, and that re-negotiation will be required if it becomes necessary to change that specification in the course of the program development process.

Any non-executable Extended ML functor specification, i.e. a functor specification having a body consisting only of the placeholder `?` or having a non-trivial body which is however not yet composed entirely of executable code, is regarded as a specification of a programming task. The task which is specified is (in the case of `?`) to fill in a body which satisfies the functor interfaces, or (in the case of a body containing axioms) to fill in a body which satisfies the axioms in the current body.

Given a specification of a programming task, there are three ways to proceed towards a program which satisfies the specification:

Decomposition step: Decompose the functor into a composition of "smaller" functors, which are then regarded as separate programming tasks in their own right.

Coding step: Provide a functor body in the form of an abstract program containing type and value declarations and a mixture of axioms and code to define them.

Refinement step: Further refine an abstract program by providing a more concrete (but possibly still non-executable) version which fills in some of the decisions left open by the more abstract version.

Decomposition and coding steps are applicable to functor specifications like the one shown above in which the body consists only of the placeholder `?`, while refinement steps are applicable to functor specifications which already have a body of some kind. Decomposition steps may be seen as programming (or program design) "in the large", while coding and refinement steps are programming "in the small".

Each of the three kinds of step gives rise to one or more proof obligations which can be generated mechanically from the "before" and "after" versions of the functor. Each proof obligation is a condition of the form:

$$exp_1 \cup \cdots \cup exp_n \models SIG$$

where $exp_1, \ldots, exp_n$ are Extended ML signatures or structure expressions and SIG is an Extended ML signature. Discharging such a proof obligation requires showing that the axioms in the signature SIG logically follow from the axioms and definitions in $exp_1, \ldots, exp_n$. A step is correct if all the proof obligations it incurs can be shown to hold. An executable Standard ML program which is obtained via a sequence of correct steps from an Extended ML specification of requirements is guaranteed to satisfy that specification. Of course, there is no need to actually do the proofs when the steps are performed; for example, they may be deferred until it is clear that a particular development path is likely to yield a satisfactory result, or until the entire development process is complete.

The details of each kind of step are given below. The example in Section 5 shows how each kind of step is used in practice to make progress during the process of developing a software system from a specification.

4.1 Decomposing functors

Decomposition step Given an Extended ML functor of the form:

```
functor F(X0 : SIG0) : SIG0' = ?
```

we may proceed by introducing a number of additional functors:

```
functor G1(X1 : SIG1) : SIG1' = ?
           ⋮
functor Gn(Xn : SIGn) : SIGn' = ?
```

and replacing the definition of `F` with the definition:

```
functor F(X0 : SIG0) : SIG0' = strexp
```

where *strexp* is a structure expression which refers to the functors `G1`, ..., `G`n. The developments of `G1`, ..., `G`n may then proceed separately.

The new definition of `F` is required to be a well-formed Extended ML functor definition. A number of proof obligations are incurred, one for each point in the expression *strexp* where two modules come into contact. This includes the point where the result delivered by *strexp* is returned as the result of `F`. In particular:

1. If the parameter structure `X0` is used in *strexp* in a context which demands a structure of signature SIG, then it is necessary to prove that `SIG0` $\models SIG$.

2. If the result of an application of `G`j is used in a context which demands a structure of signature SIG, then it is necessary to prove that `SIG`j`'` $\models SIG$.

3. If any other structure *STR* (explicit structure definition or structure identifier) is used in *strexp* in a context which demands a structure of signature *SIG*, then it is necessary to prove that $STR \models SIG$. □

The best way to understand the above is to consider a simple and very typical schematic example. Let `F` be an Extended ML functor of the form:

```
functor F(X0 : SIG0) : SIG0' = ?
```

We may proceed by introducing two new functors:

```
functor G1(X1 : SIG1) : SIG1' = ?
functor G2(X2 : SIG2) : SIG2' = ?
```

and replacing the definition of `F` with the definition

```
functor F(X0 : SIG0) : SIG0' = G2(G1(X0))
```

This incurs three proof obligations:

1. Any parameter of `F` is a suitable parameter for `G1`: `SIG0` $\models$ `SIG1`
2. Any structure delivered by `G1` is a suitable parameter for `G2`: `SIG1'` $\models$ `SIG2`
3. Any structure delivered by `G2` is a suitable result for `F`: `SIG2'` $\models$ `SIG0'`

Proving that `SIG0` $\models$ `SIG1` is a matter of showing that the axioms in `SIG1` logically follow from the axioms in `SIG0`, and likewise for the other two proof obligations.

In practice, most of the proof obligations incurred by decomposition steps are trivial to discharge by syntactic means since interfaces will almost always match exactly (i.e., in the above schematic example we will nearly always have `SIG0 = SIG1`, `SIG1' = SIG2` and `SIG2' = SIG0'`). In the example in Section 5, there are four decomposition steps which give rise to a total of nineteen proof obligations. Seventeen of these are trivial because the signatures involved match syntactically, and one is trivial because there are no axioms to prove in the consequent signature. The remaining one is also trivial since all the axioms in the consequent signature appear explicitly in the antecedent signature.

4.2 Coding functor bodies

Coding step Given an Extended ML functor of the form:

```
functor F(X : SIG) : SIG' = ?
```

we may proceed by replacing the definition of `F` with the definition:

functor F(X : SIG) : SIG' = *strexp*

where *strexp* is a well-formed Extended ML functor body. This incurs a single proof obligation:

$$\mathtt{SIG} \cup \mathit{strexp} \models \mathtt{SIG'}$$

in addition to any proof obligations arising from the use of structures within *strexp*. □

4.3 Refining abstract code

Refinement step Given an Extended ML functor of the form:

```
functor F(X : SIG) : SIG' = strexp
```

we may proceed by replacing the definition of `F` with the definition:

```
functor F(X : SIG) : SIG' = strexp'
```

where *strexp′* is a well-formed Extended ML functor body. This incurs a single proof obligation:

$$\mathtt{SIG} \cup \mathit{strexp}' \models \mathit{strexp}$$

in addition to any proof obligations arising from the use of structures within *strexp′*. □

The above subsections have set forth three ways to proceed from a specification of a programming task towards a program which satisfies the specification, and the proof obligations which are thereby incurred. Of course, one would not expect the formal development of realistic programs to proceed in practice without backtracking, mistakes and iteration, and the Extended ML methodology does not remove the possibility of unwise design decisions. One problem is that it is often very difficult to get interface specifications right the first time. For example, when implementing a functor by decomposition into simpler functors it may well be necessary to adjust the interfaces both in order to obtain a decomposition which gives rise to "true" (i.e. provable) proof obligations and to resolve problems which arise later while implementing the simpler functors. If a decomposition has been proved correct then some changes to the interfaces may be made without affecting correctness: for example, in any of the simpler functors the output interface may be strengthened or the input interface weakened without problems. It is also possible to modify the interfaces of the functor being decomposed by weakening its output signature or strengthening its input signature. This will preserve the correctness of the decomposition, but since it changes the specification of the functor such changes must be cleared with the functor's clients (higher-level functors which use it and/or the customer). Once we have made such a change to an interface we can also change interfaces it is required to match in order to take advantage of the modification. Then, provided we are able to discharge the proof obligations referring to these interfaces, overall correctness is preserved.

The proof obligations listed above for each kind of development step are actually more strict than necessary. It is possible to loosen them by taking proper account of the ideas concerning behavioural equivalence mentioned at the end of Section 3. This allows each proof obligation above to be replaced by a condition of the form:

$$\mathit{exp}_1 \cup \cdots \cup \mathit{exp}_n \models_{OBS} \mathit{SIG}$$

where $\models_{OBS}$ denotes "behavioural consequence" with respect to a certain set *OBS* of observable types. In principle, this makes the condition easier to satisfy since it only requires the observable consequences of the axioms in *SIG* to follow from the axioms and definitions in $\mathit{exp}_1, \ldots, \mathit{exp}_n$ (see [ST 89] for full details). In practice, convenient methods for proving such conditions have not yet been established and so the proof itself is rather difficult. Since the examples at hand do not require this extra flexibility, we will use the simple but strict form of the conditions listed above.

Standard ML's module language does not permit functors to take other functors as arguments. An extension to permit this is under consideration at the present time, but some of the implications of such an extension on Extended ML have already been considered. From a methodological point of view, this extension adds considerable power; one intriguing point is that it seems to introduce a bottom-up element into Extended ML's top-down program development methodology. A more detailed discussion of this issue may be found in [SST 89].

5 An example

In this section the formal development process presented in the previous section is demonstrated by means of an example. Two different developments are given which begin from the same high-level Extended ML requirements specification and yield different Standard ML programs.

Informal specification A symbol table in a compiler stores identifiers together with attributes of those identifiers which are determined at various stages during compilation. The following functions on symbol tables are required:

- Check whether or not an identifier is present in the symbol table.
- Add a new identifier to the symbol table and set its attributes.
- Look up the attributes of an identifier which is present. If the identifier is not present then return a default value.
- Change the attributes of an identifier which is already present. If the identifier is not present then nothing is changed.

Possible additional functions which would be useful in a compiler for a programming language with nested block structure are the following:

- Enter a new block. All of the identifiers in the symbol table are visible within the new block until replaced by local identifiers with the same name.
- Leave a block. All identifiers which were declared locally within the current block are removed from the symbol table.

These extra functions will not be considered here for the sake of simplicity, although the reader is invited to consider how their inclusion would alter the developments below.

Step 0

The initial formal specification of the required system is given by the following Extended ML functor specification:

```
functor Symtab
    (structure X : ID
     structure Y : ATTRIB
        ) : sig include SYMTAB
                sharing Id = X and Attrib = Y
            end
    = ?
```

where `ID`, `ATTRIB` and `SYMTAB` are Extended ML signatures as follows:

```
signature ID =
    sig
        eqtype id
    end

signature ATTRIB =
    sig
        type attrib
        val null_attrib : attrib
    end
```

```
signature SYMTAB =
    sig
        structure Id : ID
        structure Attrib : ATTRIB
        type symtab
        val empty : symtab
        val add : Id.id * Attrib.attrib * symtab -> symtab
        val change_attrib : Id.id * Attrib.attrib * symtab -> symtab

        val present : Id.id * symtab -> bool
        axiom present(i,empty) = false
        axiom present(i,add(i',a',s)) = (i=i') orelse present(i,s)
        axiom present(i,change_attrib(i',a',s)) = present(i,s)

        val lookup : Id.id * symtab -> Attrib.attrib
        axiom lookup(i,empty) = Attrib.null_attrib
        axiom lookup(i,add(i,a,s)) = a
        axiom i<>i' => lookup(i,add(i',a',s)) = lookup(i,s)
        axiom present(i,s) => lookup(i,change_attrib(i,a,s)) = a
        axiom i<>i' => lookup(i,change_attrib(i',a',s)) = lookup(i,s)
    end
```

Our target language is the executable subset of Extended ML, namely the purely functional subset of Standard ML described in Section 2. A natural consequence of this is that the functions on symbol tables will explicitly take a symbol table as argument rather than working on a single fixed symbol table which is destructively updated by adding identifiers and changing attributes. Those functions which change the symbol table will return the modified symbol table as a result. Thus, values of type `symtab` represent states of the symbol table. The empty symbol table is represented by `empty`, and the functions `add` and `change_attrib` update the state of the symbol table by adding a new identifier (and setting its attributes) and resetting the attributes of an existing identifier, respectively. The functions `present` and `lookup` may be used for querying the current state of the symbol table. These functions check whether or not an identifier is present in the symbol table and look up the attributes of an identifier, respectively.

The parameters to the system are the type of identifiers (which is required to admit equality), the type of attributes, and a default attribute value called `null_attrib`. This means that the system will cater for any choice of these types and this value. Making the type of identifiers a parameter allows identifiers to be character strings (as usual) or something more elaborate. The informal specification does not say anything about the internal structure of attributes except that there must be some default attribute value, so it is natural to provide these as parameters to the system. The function `change_attrib` sets all the attributes of an identifier regardless of their present values; more complicated interpretations of the informal requirements are possible, but this will do for our purposes.

Step 1

Design decision (decomposition) We implement `change_attrib` in terms of `add`. Exactly how this is done is left open for now. (Another possibility, which we will not consider, is to implement `add` using a function `insert` which adds a symbol without setting its attributes.)

We need two new functors:

```
functor Symtab'
    (structure X : ID
     structure Y : ATTRIB
         ) : sig include SYMTAB'
                 sharing Id = X and Attrib = Y
             end
    = ?

functor ChangeAttrib
    (X : SYMTAB'
         ) : sig include SYMTAB
                 sharing Id = X.Id and Attrib = X.Attrib
             end
    = ?
```

where SYMTAB' is exactly like SYMTAB except that the function change_attrib and the axioms which mention it are absent.

Then we can implement Symtab in terms of these functors as follows:

```
functor Symtab
    (structure X : ID
     structure Y : ATTRIB
         ) : sig include SYMTAB
                 sharing Id = X and Attrib = Y
             end
    = ChangeAttrib(Symtab'(structure X = X
                           structure Y = Y))
```

Verification Typechecks okay. All interfaces match exactly so there is nothing to check. □

Step 2

Design decision (coding) Implement the functor ChangeAttrib by coding change_attrib in terms of add in the obvious way.

```
functor ChangeAttrib
    (X : SYMTAB'
         ) : sig include SYMTAB
                 sharing Id = X.Id and Attrib = X.Attrib
             end
    = struct
          open X
          fun change_attrib(i:Id.id,a:Attrib.attrib,s:symtab) = ? : symtab
          axiom present(i,s) => change_attrib(i,a,s) = add(i,a,s)
          axiom not present(i,s) => change_attrib(i,a,s) = s
      end
```

The declaration open X includes the substructures, types and values of X in the result of ChangeAttrib. Thus, it abbreviates the following sequence of declarations:

```
structure Id : ID = X.Id
structure Attrib : ATTRIB = X.Attrib
type symtab = X.symtab
val empty = X.empty
val add = X.add
val present = X.present
val lookup = X.lookup
```

Verification Typechecks okay. We have to show that

$$\mathtt{SYMTAB}' \cup \mathit{body} \models \mathtt{SYMTAB}$$

where *body* is the body of `ChangeAttrib`. The only non-trivial part of this involves the axioms of `SYMTAB` which are not in `SYMTAB'`, namely those which mention the function `change_attrib`:

```
present(i,change_attrib(i',a',s)) = present(i,s)
present(i,s) => lookup(i,change_attrib(i,a,s)) = a
i<>i' => lookup(i,change_attrib(i',a',s)) = lookup(i,s)
```

The second of these follows directly from an axiom in the body of `ChangeAttrib` and an axiom in `SYMTAB'` while the first and third require simple case analyses. □

Step 3

Design decision (refinement) Convert the axioms for `change_attrib` into ML code. The only change required is to make the case analysis in the axioms explicit using `if _ then _ else _`.

```
functor ChangeAttrib
    (X : SYMTAB'
         ) : sig include SYMTAB
                 sharing Id = X.Id and Attrib = X.Attrib
             end
    = struct
          open X
          fun change_attrib(i,a,s) = if present(i,s) then add(i,a,s) else s
      end
```

Verification Typechecks okay. The axioms for `change_attrib` in the previous version of the body follow directly from the function definition in the current version of the body. □

Pause for breath

At this point it is necessary to choose an representation of symbol tables as specified in `Symtab'` in terms of simpler data types. There are many possibilities, including at least the following (see [Sed 88] and similar texts for details):

1. Terms built from the constant `empty` using the constructor function `add`.

2. Sequences with identifiers kept in the order in which they are added.

3. Like (2), but with duplicates removed.

The following five choices require an additional order relation on identifiers to be supplied. Since this involves changing the original specification, it would be necessary to negotiate with the customer to see if this change is acceptable. Alternatively, if the customer is satisfied with a non-generic implementation of symbol tables in which the type of identifiers is fixed as strings of characters, the order relation need not be supplied since there is an appropriate one available.

4. Sequences with identifiers kept in ascending or descending order, with or without duplicates.
5. Like (4), but using an array in place of a sequence, with sequential search.
6. Like (5), but with binary search.
7. Ordered binary trees, with or without duplicates.
8. Balanced trees (e.g. 2–3–4 trees, AVL trees, 2–3 trees etc.).

The following two choices require an additional hash function to be supplied which takes identifiers to some given range of natural numbers. Since this involves changing the original specification, prior consultation with the customer is again required. And again, such a function need not be supplied by the customer if the type of identifiers is fixed as strings of characters.

9. Hash tables with separate chains of collisions kept in the order in which they are added, with or without duplicates.
10. Hash tables with linear probing, with rehashing into a larger table when the table becomes nearly full.

Other possibilities (again requiring modification to the original specification) are: a variation on (9) in which chains of collisions are kept in ascending or descending order; and, a variation on (10) with double hashing. Note that a variation on (10) in which the size of the table is fixed is *not* an option (assuming that the number of possible identifiers is infinite) since `SYMTAB'` requires symbol tables to be capable of storing an arbitarily large number of different identifiers.

In this paper we will look at just two of these possibilities: (1) and (3). The development process therefore splits at this point into two alternative development paths, which will be treated in two separate subsections.

5.1 Symbol tables represented as terms

The simplest way to represent symbol tables in ML is as terms built from the constant `empty` using the constructor function `add`. A similar method is applicable whenever there are no non-trivial equations inferrable between constructor terms of the type being represented. If this method is chosen then the implementation follows almost immediately from the specification of `Symtab'` in Step 1 above.

Step 4

Design decision (coding) Implement the functor `Symtab'` by representing symbol tables directly as terms.

```
functor Symtab'
    (structure X : ID
     structure Y : ATTRIB
        ) : sig include SYMTAB'
                sharing Id = X and Attrib = Y
            end
    = struct
          structure Id : ID = X
          structure Attrib : ATTRIB = Y
          datatype symtab =
              empty
            | add of Id.id * Attrib.attrib * symtab
          fun present(i:Id.id,s:symtab) = ? : bool
          axiom present(i,empty) = false
          axiom present(i,add(i',a',s)) = (i=i') orelse present(i,s)

          fun lookup(i:Id.id,s:symtab) = ? : Attrib.attrib
          axiom lookup(i,empty) = Attrib.null_attrib
          axiom lookup(i,add(i,a,s)) = a
          axiom i<>i' => lookup(i,add(i',a',s)) = lookup(i,s)
      end
```

Verification Typechecks okay. All the axioms in SYMTAB' appear in the body of the functor, so there is nothing to prove. □

Step 5

Design decision (refinement) Convert the axioms for present and lookup into ML code. No change is required to the axioms for present; the only change required to the axioms for lookup is to make the case analysis explicit using if _ then _ else _.

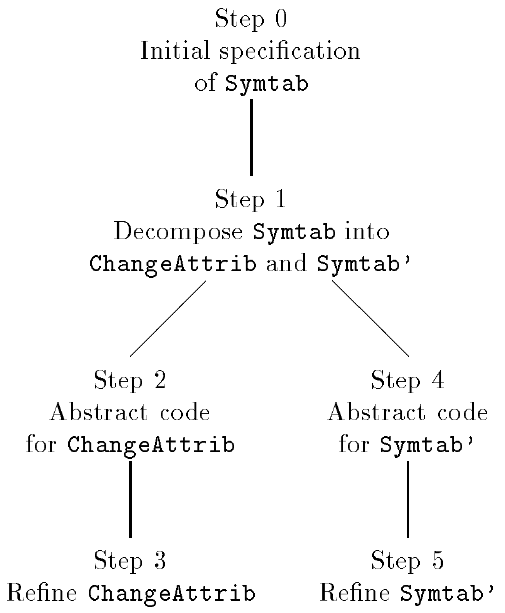
```
functor Symtab'
    (structure X : ID
     structure Y : ATTRIB
        ) : sig include SYMTAB'
                sharing Id = X and Attrib = Y
            end
    = struct
          structure Id : ID = X
          structure Attrib : ATTRIB = Y
          datatype symtab =
              empty
            | add of Id.id * Attrib.attrib * symtab

          fun present(i,empty) = false
            | present(i,add(i',a',s)) = (i=i') orelse present(i,s)

          fun lookup(i,empty) = Attrib.null_attrib
            | lookup(i,add(i',a,s)) = if i=i' then a else lookup(i,s)
      end
```

Verification Typechecks okay. The axioms for present and lookup in the previous version of the body follow directly from the function definitions in the current version of the body. □

All functor bodies are now expressed entirely in Standard ML, so we are finished with this development path. The functors appearing in the final program are given above under steps 1, 3 and 5. The following tree shows the dependencies between the development steps:



5.2 Symbol tables represented as sequences

An alternative to the above is to represent symbol tables using sequences of ⟨identifier × attribute⟩-pairs. Having selected this representation, there are several choices to be made concerning the details:

1. What dictates the order of the entries in the sequence?
   (a) Adding an entry puts it at the front of the sequence.
   (b) Adding an entry only puts it at the front of the sequence if the identifier is not already present.
   (c) The entries are kept in order of their identifiers (with respect to some order on identifiers).
2. Are duplicates removed?
   (a) Duplicates are not removed: each `add` puts an additional entry in the sequence.
   (b) Duplicates are removed.

We will choose the combination of 1(a) and 2(b) here. Recall that Steps 0–3 from the beginning of this section are still relevant to this development path.

Step 4

Design decision (decomposition) We implement `Symtab'` in terms of sequences of ⟨identifier × attribute⟩-pairs. Exactly how the functions of `Symtab'` are expressed in terms of the functions provided on sequences is left open for now.

We need two new functors:

```
functor SeqPairs
    (structure X : ID
     structure Y : ATTRIB
         ) : sig include SEQPAIRS
                 sharing Id = X and Attrib = Y
             end
    = ?

functor Symtab''
    (S : SEQPAIRS
         ) : sig include SYMTAB'
                 sharing Id = S.Id and Attrib = S.Attrib
             end
    = ?
```

where SEQPAIRS is as follows:

```
signature SEQPAIRS =
    sig
        structure Id : ID
        structure Attrib : ATTRIB
        datatype sequence =
            nil
          | cons of (Id.id * Attrib.attrib) * sequence

        val null : sequence -> bool
        axiom null nil = true
        axiom null(cons((i,a),s)) = false

        val hd : sequence -> Id.id * Attrib.attrib
        axiom hd(cons((i,a),s)) = (i,a)

        val tl : sequence -> sequence
        axiom tl(cons((i,a),s)) = s
    end
```

Then we can implement Symtab' in terms of these functors as follows:

```
functor Symtab'
    (structure X : ID
     structure Y : ATTRIB
         ) : sig include SYMTAB'
                 sharing Id = X and Attrib = Y
             end
    = Symtab''(SeqPairs(structure X = X
                        structure Y = Y))
```

Verification Typechecks okay. All interfaces match exactly so there is nothing to check. □

Step 5

Design decision (decomposition) We implement Symtab'' in terms of sequences of ⟨identifier × attribute⟩-pairs where no more than one pair in a sequence has the same identifier. Exactly how the functions of Symtab'' are expressed in terms of the functions provided on such sequences is left open for now.

We need two new functors:

```
functor SeqDup
    (S : SEQPAIRS
         ) : sig include SEQDUP
                 sharing Seq = S
             end
    = ?

functor Symtab'''
    (S : SEQDUP
         ) : sig include SYMTAB'
                 sharing Id = S.Seq.Id and Attrib = S.Seq.Attrib
             end
    = ?
```

where SEQDUP is as follows:

```
signature SEQDUP =
    sig
        structure Seq : SEQPAIRS
        val add : (Seq.Id.id * Seq.Attrib.attrib) * Seq.sequence -> Seq.sequence
        val ismatch : Seq.Id.id * Seq.sequence -> bool
        val remove : Seq.Id.id * Seq.sequence -> Seq.sequence

        (* axioms for add *)
        axiom ismatch(i,s) => add((i,a),s) = Seq.cons((i,a),remove(i,s))
        axiom not ismatch(i,s) => add((i,a),s) = Seq.cons((i,a),s)

        (* axioms for ismatch *)
        axiom ismatch(i,Seq.nil) = false
        axiom ismatch(i,Seq.cons((i',a'),s)) = (i=i') orelse ismatch(i,s)

        (* axioms for remove *)
        axiom not ismatch(i,remove(i,s))
        local
           val member : (Seq.Id.id * Seq.Attrib.attrib) * Seq.sequence -> bool
           axiom member(e,Seq.nil) = false
           axiom member(e,Seq.cons(e',s)) = (e=e') orelse member(e,s)
        in
           axiom i<>i' => member((i',a'),remove(i,s)) = member((i',a'),s)
        end
    end
```

The axioms for add ensure that sequences built from empty and add contain no pairs with duplicate identifiers. The function member is an auxiliary function which is introduced in order to simplify the specification of remove. Since it is declared as local, it need not be implemented in structures which match SEQDUP.

We can now implement Symtab'' in terms of SeqDup and Symtab''' as follows:

```
functor Symtab''
    (S : SEQPAIRS
         ) : sig include SYMTAB'
                 sharing Id = S.Id and Attrib = S.Attrib
             end
    = Symtab'''(SeqDup(S))
```

Verification Typechecks okay. All interfaces match exactly so there is nothing to check. □

Step 6

Design decision (coding) Implement the functor `Symtab'''` by representing `symtab` using sequences, with `add` on symbol tables implemented by `add` (without duplicates) on sequences.

```
functor Symtab'''
    (S : SEQDUP
         ) : sig include SYMTAB'
                 sharing Id = S.Seq.Id and Attrib = S.Seq.Attrib
             end
    = struct
          structure Id : ID = S.Seq.Id
          structure Attrib : ATTRIB = S.Seq.Attrib
          type symtab = S.Seq.sequence
          val empty = S.Seq.nil
          fun add(i,a,s) = S.add((i,a),s)
          val present = S.ismatch

          fun lookup(i:Id.id,s:symtab) = ? : Attrib.attrib
          axiom lookup(i,empty) = Attrib.null_attrib
          axiom lookup(i,add(i,a,s)) = a
          axiom i<>i' => lookup(i,add(i',a',s)) = lookup(i,s)
      end
```

Verification Typechecks okay. We have to show that

$$\texttt{SEQDUP} \cup \mathit{body} \models \texttt{SYMTAB}'$$

where *body* is the body of `Symtab'''`. The only non-trivial part of this involves the axioms for the function `present` in `SYMTAB'`:

```
present(i,empty) = false
present(i,add(i',a',s)) = (i=i') orelse present(i,s)
```

The first of these follows directly from the definition of `present` in the body of `Symtab'''` and an axiom in `SEQDUP`. To prove the second we must first prove the following:

Lemma *The formula*

```
i<>i' => S.ismatch(i,S.remove(i',s)) = S.ismatch(i,s)
```

follows from `SEQDUP` ∪ *body.*

Proof By structural induction on the type `S.Seq.sequence`. Structural induction is valid since this type is declared (in `SEQPAIRS`, which is part of `SEQDUP`) as a `datatype`. □(of Lemma)

The required result then follows by a simple case analysis. □

Step 7

Design decision (refinement) Convert the axioms for `lookup` into ML code. The only change required is to make the case analysis in the axioms explicit using `if _ then _ else _`.

```
functor Symtab'''
    (S : SEQDUP
         ) : sig include SYMTAB'
                 sharing Id = S.Seq.Id and Attrib = S.Seq.Attrib
             end
    = struct
          structure Id : ID = S.Seq.Id
          structure Attrib : ATTRIB = S.Seq.Attrib
          type symtab = S.Seq.sequence
          val empty = S.Seq.nil
          fun add(i,a,s) = S.add((i,a),s)
          val present = S.ismatch
          fun lookup(i,s) = if S.Seq.null s then Attrib.null_attrib
                            else let val (i',a') = S.Seq.hd s in
                                     if i=i' then a'
                                     else lookup(i,S.Seq.tl s) end
      end
```

Verification Typechecks okay. We have to prove that the axioms for `lookup` in the previous version of the body of `Symtab'''` follow from the function definition in the current version of the body and the axioms in `SEQDUP` (this contains `SEQPAIRS`, so the axioms there may be used as well). The relevant axioms from the previous version of `Symtab'''` are:

```
lookup(i,empty) = Attrib.null_attrib
lookup(i,add(i,a,s)) = a
i<>i' => lookup(i,add(i',a',s)) = lookup(i,s)
```

The first of these follows directly from the definition of `lookup` above and an axiom in `SEQPAIRS`. The other two require a simple case analysis. □

Step 8

Design decision (coding) Implement the functor `SeqDup`. At this stage we convert the axioms for `add` and `ismatch` to ML code, but leave `remove` defined by axioms. (An alternative would be to implement `SeqDup` using a more general functor for duplicate-free sequences of arbitrary elements which is parameterized by the type of elements, the type of keys and the function which produces the key of an element, and use this functor for the case where elements are ⟨identifier $\times$ attribute⟩-pairs, keys are identifiers, and the left projection produces the key of an element. But this would require some restructuring since we have already decided to use sequences as specified in `SEQPAIRS` as symbol tables.)

```
functor SeqDup
    (S : SEQPAIRS
         ) : sig include SEQDUP
                 sharing Seq = S
             end
    = struct
         structure Seq : SEQPAIRS = S
         fun ismatch(i,Seq.nil) = false
           | ismatch(i,Seq.cons((i',a'),s)) = (i=i') orelse ismatch(i,s)
```

```
        fun remove(i:Seq.Id.id,s:Seq.sequence) = ? : Seq.sequence
        axiom not ismatch(i,remove(i,s))
        local
           fun member((i:Seq.Id.id,a:Seq.Attrib.attrib),s:Seq.sequence)
                                                               = ? : bool
           axiom member(e,Seq.nil) = false
           axiom member(e,Seq.cons(e',s)) = (e=e') orelse member(e,s)
        in
           axiom i<>i' => member((i',a'),remove(i,s)) = member((i',a'),s)
        end

        fun add((i,a),s) = if ismatch(i,s) then Seq.cons((i,a),remove(i,s))
                           else Seq.cons((i,a),s)
     end
```

Verification Typechecks okay. The axioms for `add` and `ismatch` in `SEQDUP` follow directly from the function definitions in the body of `SeqDup`, and the axioms for `remove` are unchanged. □

Step 9

Design decision (refinement) Supply ML code for the function `remove`. The local declaration of `member` becomes superfluous (and need not be converted to ML code) since the code for `remove` does not refer to `member`.

```
functor SeqDup
    (S : SEQPAIRS
         ) : sig include SEQDUP
                 sharing Seq = S
             end
    = struct
         structure Seq : SEQPAIRS = S
         fun ismatch(i,Seq.nil) = false
           | ismatch(i,Seq.cons((i',a'),s)) = (i=i') orelse ismatch(i,s)
         fun remove(i,Seq.nil) = Seq.nil
           | remove(i,Seq.cons((i',a'),s)) = if i=i' then remove(i,s)
                                             else Seq.cons((i',a'),remove(i,s))
         fun add((i,a),s) = if ismatch(i,s) then Seq.cons((i,a),remove(i,s))
                            else Seq.cons((i,a),s)
      end
```

Verification Typechecks okay. We have to show that

$$\texttt{SEQPAIRS} \cup \textit{body of } \texttt{SeqDup} \models \textit{previous version of body of } \texttt{SeqDup}$$

The only axioms in the previous version of the body of `SeqDup` which do not appear in the present version are those for `remove`:

```
not ismatch(i,remove(i,s))
i<>i' => member((i',a'),remove(i,s)) = member((i',a'),s)
```

Both proofs proceed by structural induction on `s` and case analysis. For the second proof we are allowed to make use of the axioms for the function `member` in the previous version of the body of `SeqDup`. □

Step 10

Design decision (decomposition) We implement `SeqPairs` using a more general functor `Seq` which is parameterized by the type of elements. We will use this for the case where elements are ⟨identifier × attribute⟩-pairs.

We need one new functor:

```
functor Seq
    (E : ELEM
         ) : sig include SEQ
                 sharing Elem = E
             end
    = ?
```

where `ELEM` and `SEQ` are as follows:

```
signature ELEM =
    sig
        type elem
    end

signature SEQ =
    sig
        structure Elem : ELEM
        datatype sequence =
            nil
          | cons of Elem.elem * sequence

        val null : sequence -> bool
        axiom null nil = true
        axiom null(cons(e,s)) = false

        val hd : sequence -> Elem.elem
        axiom hd(cons(e,s)) = e

        val tl : sequence -> sequence
        axiom tl(cons(e,s)) = s
    end
```

Then we can implement `SeqPairs` in terms of this functor as follows:

```
functor SeqPairs
    (structure X : ID
     structure Y : ATTRIB
         ) : sig include SEQPAIRS
                 sharing Id = X and Attrib = Y
             end
    = struct
          structure Id : ID = X
          structure Attrib : ATTRIB = Y
          structure Elem : ELEM =
                struct
                    type elem = Id.id * Attrib.attrib
                end
```

```
            structure Seq : SEQ = Seq(Elem)
            open Seq
        end
```

Verification Typechecks okay. All of the structure declarations in the body of `SeqPairs` are trivially well-formed (in the case of `Elem`, this is because `ELEM` contains no axioms). We therefore have only to show that

$$\left(\begin{array}{l}\texttt{structure Id : ID}\\ \texttt{structure Attrib : ATTRIB}\\ \texttt{structure Elem : ELEM}\\ \texttt{structure Seq : SEQ}\\ \texttt{open Seq}\end{array}\right) \models \texttt{SEQPAIRS}$$

All the axioms in SEQPAIRS follow immediately. □

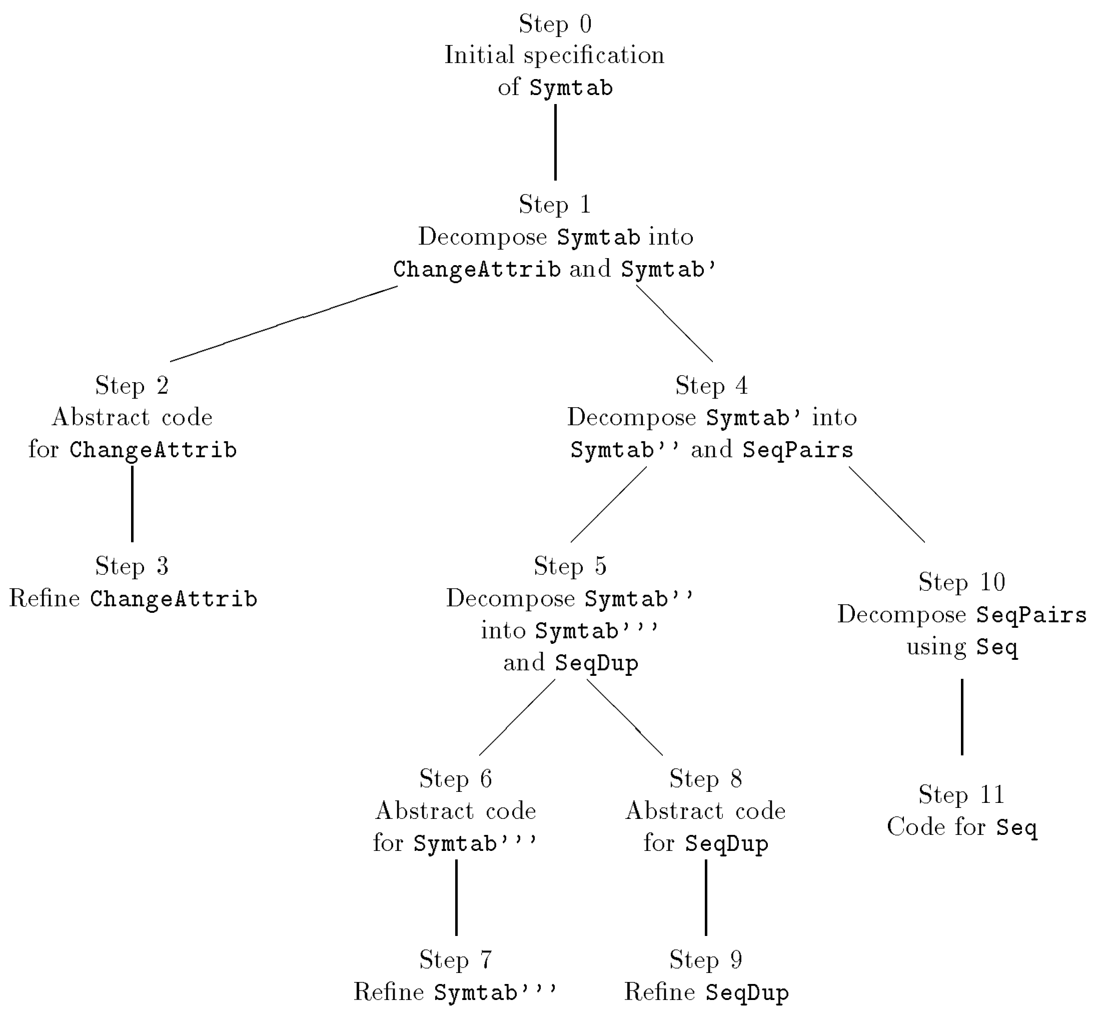
Step 11

Design decision (coding) One would normally expect the functor `Seq` to be available in the library. In case it is not available, the axioms can be converted directly into ML code using an implementation of sequences as terms.

```
functor Seq
    (E : ELEM
         ) : sig include SEQ
                 sharing Elem = E
             end
    = struct
         structure Elem : ELEM = E
         datatype sequence =
             nil
           | cons of Elem.elem * sequence
         fun null nil = true
           | null(cons(e,s)) = false
         fun hd(cons(e,s)) = e
         fun tl nil = nil
           | tl(cons(e,s)) = s
      end
```

Verification Typechecks okay. All the axioms in `SEQ` follow immediately from the function definitions in the body of `Seq`. Strictly speaking, this code is invalid since the function `hd` is not totally defined, but any choice of value for `hd nil` will do (the same is true for `tl nil`, which we have arbitrarily given the value `nil`). The values of `hd nil` and `tl nil` are unimportant since the specified interface of `Seq` does not make any promises concerning them. However, we are operating under the global assumption that all functions are total, which means that we should ensure that some value is returned. □

All functor bodies are now expressed entirely in Standard ML, so we are finished with this development path. The functors appearing in the final program are given above under steps 1, 3, 4, 5, 7, 9, 10 and 11. The following tree shows the dependencies between the development steps:


Step 0
Initial specification of Symtab
Step 1
Decompose Symtab into ChangeAttrib and Symtab'
Step 2
Abstract code for ChangeAttrib
Step 3
Refine ChangeAttrib
Step 4
Decompose Symtab' into Symtab'' and SeqPairs
Step 5
Decompose Symtab'' into Symtab''' and SeqDup
Step 6
Abstract code for Symtab'''
Step 7
Refine Symtab'''
Step 8
Abstract code for SeqDup
Step 9
Refine SeqDup
Step 10
Decompose SeqPairs using Seq
Step 11
Code for Seq

6 Concluding remarks

This paper has presented the Extended ML approach to formal program development in a way which is intended to emphasize the practical aspects of formal program development while avoiding theoretical issues as much as possible. The importance of sound mathematical foundations to support the enterprise of formal program development cannot be over-emphasized, and this is one of Extended ML's strengths, but a formal program development framework should be designed in such a way that the user of the framework is not forced to be aware of these foundations.

One important feature of Extended ML which has not been stressed in this paper is the fact that the Extended ML language and methodology are practically independent of the logic used to write axioms, as well as of the form of signatures and structures (see [ST 86] for details). The notation of first-order equational logic has been used here to write axioms and signatures/structures contain types and values as in ML, but we could have used order-sorted equational logic [GM 87] and imposed a sub-type relation on types as in OBJ3 [GW 88] (although this would have been a awkward choice for producing programs in Standard ML since it is unable to cope with sub-types and coercions). The semantics of Extended ML regards executable code as a special case of axioms; e.g., Standard ML function definitions can be viewed as axioms of first-order equational logic which have the special form:

$$f(p_1) = expr_1 \wedge \cdots \wedge f(p_n) = expr_n$$

where $p_1, \ldots, p_n$ are patterns (terms containing constructors and variables only) and all the variables in $expr_j$ appear in p_j, for all $j \leq n$. As a practical consequence, Extended ML can be used to develop programs in other target programming languages. For example, if we switch to untyped first-order predicate logic and regard Horn clauses as the executable subset of this logic, the result is a language and methodology for developing modular Prolog programs (see [SW 87]) from specifications. Another consequence is that the present restriction to a small subset of Standard ML (excluding higher-order functions, polymorphism, references, exceptions etc.) is only necessary until a logic is developed which is able to cope with all these features adequately. Developing such a logic will not be an easy job by any means, but it is one which can be tackled separately.

The aims of Extended ML are broadly similar to those of work on rigorous program development by the VDM school (see e.g. [Jones 80]). VDM is a method for software specification and development, based on the use of explicitly-defined models of software systems, which has been widely applied in practice. However, it is *rigorous* rather than fully *formal*, and lacks formal mathematical foundations and explicit structuring mechanisms (the RAISE project [BDMP 85] is attempting to fill these gaps). In contrast, work on Extended ML builds on formal mathematical foundations with a strong emphasis on modularity and programming/design in the large; problems of practical usability are addressed, but such concerns are never allowed to take precedence over the need to maintain the soundness of the foundations. At a technical level, two advantages of the Extended ML approach (neither of which have been properly discussed here) are the use of behavioural equivalence which handles the transition between data specification and representation in a more general way than VDM's *retrieve functions*, and the independence from the underlying logical framework and target programming language mentioned above. Extended ML is primarily designed to support the development of programs from property-oriented (axiomatic) specifications rather than model-oriented specifications, but it is able to cope with model-oriented specifications as well via the use of behavioural equivalence.

Much work remains to be done. One of the most glaring omissions at present is the lack of machine-based tools to support formal program development in Extended ML. This is one of the main goals of a SERC-funded project in Edinburgh which began in May 1989. The first step will be a parser/typechecker for Extended ML specifications which will allow specifications to be checked for silly mistakes and produce abstract syntax trees in a form suitable for processing by other tools; this will be available soon. A number of theorem provers are available which are able to cope with the proofs involved in program development examples like the one in Section 5, but once one is adopted it will have to be enriched to cope with the modular structure of specifications along the lines described in [SB 83]. A component is also needed to generate proof obligations from development steps and to keep track of these and of the programming tasks which remain to be tackled. Other plans are sketched in [ST 88]. The support system will be written in Standard ML, which will allow us to experiment with the use of the techniques we advocate in developing the components of the system itself.

Acknowledgements

Much of this paper is a rehash of ideas from [ST 89]. I gratefully acknowledge the work of Andrzej Tarlecki of the Polish Academy of Sciences in our continuing collaboration on Extended ML and on topics in the foundations of algebraic specification and formal program development on which this work is based. Thanks to Edmund Kazmierczak for comments on a draft of this paper. The research reported here has been partially supported by grants from the U.K. Science and Engineering Research Council and the Polish Academy of Sciences.

7 References

[Note: LNCS n = Springer Lecture Notes in Computer Science, Volume n]

[BW 88] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall (1988).

[BDMP 85] D. Bjørner, T. Denvir, E. Meiling and J. Pedersen. The RAISE project: fundamental issues and requirements. Report RAISE/DDC/EM/1/V6, Dansk Datamatic Center (1985).

[BMS 80] R. Burstall, D. MacQueen and D. Sannella. Hope: an experimental applicative language. *Proc. 1980 LISP Conference*, Stanford, California, pp. 136–143 (1980).

[GB 84] J. Goguen and R. Burstall. Introducing institutions. *Proc. Logics of Programming Workshop*, Carnegie-Mellon. LNCS 164, pp. 221–256 (1984).

[GM 87] J. Goguen and J. Meseguer. Order-sorted algebra solves the constructor-selector, multiple representation and coercion problems. *Proc. 2nd IEEE Symp. on Logic in Computer Science*, Ithaca, New York, pp. 18–29 (1987).

[GW 88] J. Goguen and T. Winkler. Introducing OBJ3. Report SRI-CSL-88-9, Computer Science Laboratory, SRI International (1988).

[Har 89] R. Harper. Introduction to Standard ML. Report ECS-LFCS-86-14, Univ. of Edinburgh. Revised edition (1989).

[HMM 86] R. Harper, D. MacQueen and R. Milner. Standard ML. Report ECS-LFCS-86-2, Univ. of Edinburgh (1986).

[HMT 89] R. Harper, R. Milner and M. Tofte. The definition of Standard ML (version 3). Report ECS-LFCS-89-81, Univ. of Edinburgh (1989).

[HW 89] P. Hudak and P. Wadler *et al.* Report on the functional programming language Haskell. Report CSC/89/R5, Univ. of Glasgow (1989).

[Jones 80] C. Jones. *Software Development: A Rigorous Approach.* Prentice-Hall (1980).

[MacQ 86] D. MacQueen. Modules for Standard ML. In: [HMM 86] (1986).

[Reade 89] C. Reade. *Elements of Functional Programming.* Addison-Wesley (1989).

[SB 83] D. Sannella and R. Burstall. Structured theories in LCF. *Proc. 8th Colloq. on Trees in Algebra and Programming*, L'Aquila, Italy. LNCS 159, pp. 377–391 (1983).

[SS 89] D. Sannella and F. da Silva. Syntax, typechecking and dynamic semantics for Extended ML. Report ECS-LFCS-89-101, Univ. of Edinburgh (1989).

[SST 89] D. Sannella, S. Sokołowski and A. Tarlecki. Toward formal development of programs from algebraic specifications: parameterisation revisited. Technical Report, Laboratory for Foundations of Computer Science, Dept. of Computer Science, Univ. of Edinburgh (to appear).

[ST 85] D. Sannella and A. Tarlecki. Program specification and development in Standard ML. *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans, pp. 67–77 (1985).

[ST 86] D. Sannella and A. Tarlecki. Extended ML: an institution-independent framework for formal program development. *Proc. Workshop on Category Theory and Computer Programming*, Guildford. LNCS 240, pp. 364–389 (1986).

[ST 88] D. Sannella and A. Tarlecki. Tools for formal program development: some fantasies. *LFCS Newsletter*, No. 1, pp. 10–15 (1988).

[ST 89] D. Sannella and A. Tarlecki. Toward formal development of ML programs: foundations and methodology. Report ECS-LFCS-89-71, Laboratory for Foundations of Computer Science, Dept. of Computer Science, Univ. of Edinburgh (1989); extended abstract in *Proc. Colloq. on Current Issues in Programming Languages*, Joint Conf. on Theory and Practice of Software Development (TAPSOFT), Barcelona. LNCS 352, pp. 375–389 (1989).

[SW 87] D. Sannella and L. Wallen. A calculus for the construction of modular Prolog programs. *Proc. 1987 IEEE Symp. on Logic Programming*, San Francisco, pp. 368–378 (1987). To appear in *Journal of Logic Programming.*

[Sed 88] R. Sedgewick. *Algorithms*, 2nd edition. Addison-Wesley (1988).

[Tofte 89] M. Tofte. Four lectures on Standard ML. Report ECS-LFCS-89-73, Univ. of Edinburgh (1989).

[Wik 87] Å. Wikström. *Functional Programming Using Standard ML.* Prentice-Hall (1987).