PROGRAM SPECIFICATION AND DEVELOPMENT IN STANDARD ML

Donald Sannella and Andrzej Tarlecki[1]
Department of Computer Science
University of Edinburgh

Abstract
An attempt is made to apply ideas about algebraic specification in the context of a programming language. Standard ML with modules is extended by allowing axioms in module interface specifications and in place of code. The resulting specification language, called Extended ML, is given a semantics based on the primitive specification-building operations of the kernel algebraic specification language ASL. Extended ML provides a framework for the formal development of programs from specifications by stepwise refinement, which is illustrated by means of a simple example. From its semantic basis Extended ML inherits complete independence from the logical system (institution) used to write specifications. This allows different styles of specification as well as different programming languages to be accommodated.

1 Introduction

Beginning with [Gut 75] and [ADJ 76], work on the algebraic approach to program specification has focused on developing techniques of specifying programs (abstract data types in particular) and more recently on formalising the notion of refinement as used in stepwise refinement (see e.g. [Ehr 79] and [EKMP 82]). The ultimate goal of this work is to provide a formal basis for program development, which would e.g. support a methodology for systematic development of programs from specifications by means of verified refinement steps. However, comparatively little work has been done on applying the results to programming, with a few exceptions such as CIP-L [Bau 81], IOTA [NY 83] and Anna [LHKO 84].

This paper represents our first attempt at applying our ideas about algebraic specifications in the context of a programming language. The approach to algebraic specification this is based on, described in [SW 83], [Wir 83], [ST 84] and [ST 85], features a specification formalism heavily based on the notion of *behavioural equivalence*, which makes it possible to adopt a very simple definition of refinement. Another feature of this approach is that it is independent of the underlying logical system (or *institution* [GB 83]). This means that we are not bound to using any particular logic in specifications and so can readily adapt from the simple world of algebraic specifications (with total functions, equational axioms, etc.) to the more complex world of programming languages (with nonterminating programs, exceptions, etc.).

The programming language we choose to work with is Standard ML [Mil 84] with modules [MacQ 84]. ML is a statically-scoped functional language which features a flexible but completely secure polymorphic type system [Mil 78]. Large ML programs can be structured into *modules* with explicitly-specified interfaces.

We begin in section 2 by outlining an algebraic semantics of Standard ML modules under some simplifying assumptions. We then proceed to extend the language by permitting axioms to be used in module interface specifications and in place of code; this makes it possible to use the same language to write high-level specifications, programs, and everything in between. This extended language we call Extended ML. The semantics of Extended ML is presented in sections 3 and 4. In section 5 we explain how programs may be formally developed in Extended ML, and in section 6 a simple example is given. In section 7 we briefly explain in what sense

[1] On leave from Institute of Computer Science, Polish Academy of Sciences, Warsaw.

Extended ML is independent of the underlying logical system, and how this allows us to discharge the simplifying assumptions made in section 2.

Although for readability this paper does not contain all the formal details, we would like to emphasise that the strength of this work is that it rests on mathematically firm foundations as detailed in [SW 83], [Wir 83], [ST 84] and [ST 85]. We regard this as absolutely essential; a theory of formal program development does not inspire much confidence otherwise. The technical details (mainly the formal semantics of Extended ML in terms of the specification-building operations described in [ST 84] and proof rules for refinement in the context of behavioural equivalence on the basis of those in [ST 85]) will appear in a longer version of this paper.

2 Standard ML with modules

In this paper, we will restrict our attention to the applicative subset of Standard ML (i.e. without assignment and exceptions). We will further assume that no use is made of polymorphic types and that all functions are first-order and total. Except for the purpose of simplifying the presentation, none of these restrictions are necessary, as discussed in section 7. The reader need not be acquainted with the particular features and syntactic details of Standard ML itself, except to keep in mind the assumptions stated above -- it is sufficient to know that a Standard-ML program defines a set of types and functions. The examples (particularly the one in section 6) will make use of Standard ML as described in [Mil 84].

Of particular interest to us are the extensions to Standard ML for modular programming proposed in [MacQ 84]. The proposal requires that interfaces (*signatures*) and their implementations (*structures*[2]) be defined separately. Every structure has a signature which gives the names of the types and functions 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 also reflected in its signature. *Modules* are "parameterised" structures; if we apply a module to a structure we get a structure. A module has an input signature describing structures to which it may be applied, and an output signature describing the result of an application. A module may have several parameters, but since this is the same as having a single large parameter we will restrict attention when convenient to the single-parameter case (the same approach is taken in [MacQ 84]). 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*.

[2]In the old version of [MacQ 84], structures were called *instances*

An example of a simple program in Standard ML with modules is the following:

```
signature POSig =
   sig type elem
       val le: elem × elem → bool
   end

signature SortSig =
   sig structure PO: POSig
       val sort: PO.elem list → PO.elem list
   end


module SortMod (PO: POSig) : SortSig
   inherit PO
   { *** insert is a local function *** }
   val rec insert(a,nil) = [a]
         | insert(a,b::l) = if PO.le(a,b)
                            then a::b::l
                            else b::insert(a,l)
   and rec sort nil = nil
         | sort(a::l) = insert(a,sort l)
end

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

Now, *SortMod(IntPO).sort* [11,5,8] = [5,8,11].

In this example, the types of the functions *sort* and *insert* in the module *SortMod* are inferred by the typechecker; the type of *sort* must be as declared in the signature *SortSig*. Note that the "types" (i.e. signatures) of structures and modules are given explicitly.

We can describe the semantics of signatures, structures and modules in Standard ML in terms of the following mathematical objects:

Definition: An *algebraic signature* Σ is a pair $\langle S,\Omega\rangle$, where S is a set of *sorts* and Ω is a set of *operations* in which each $\omega\in\Omega$ has a *type* of the form $s_1\times\ldots\times s_n\rightarrow s$, for $s_1,\ldots,s_n,s\in S$.

Definition: Let $\Sigma=\langle S,\Omega\rangle$ be an algebraic signature. A Σ-*algebra* A consists of a *carrier set* $|A|_s$ for every sort $s\in S$, and a total function $\omega_A:|A|_{s_1}\times\ldots\times|A|_{s_n}\rightarrow|A|_s$ for every operation $\omega\in\Omega$ of type $s_1\times\ldots\times s_n\rightarrow s$.

A Standard-ML signature *Sig* denotes an algebraic signature Σ. This may be identified with the class of Σ-algebras which will be denoted Mod[*Sig*] (the *models* of *Sig*). A structure *Str*:*Sig* corresponds to a Σ-algebra A, i.e. an algebra in Mod[*Sig*]. A module $M(X{:}Sig_p){:}Sig_r$ denotes a (total) function taking every algebra in Mod[Sig_p] to an algebra in Mod[Sig_r]. From now on we will ignore the distinction between a syntactic object (e.g. *Sig*, *Str*) and the mathematical object it denotes (e.g. Σ, A), referring to both by the same name.

To make the semantics of modules more precise: in the module

```
module M(X: Σp) : Σr
    <body>
```

the code in *<body>* can make reference to the names in Σ_p (prefixed by *X.* — see next paragraph) and to types and operations defined earlier in *<body>*. This code introduces some new types and operations Σ_{local} and defines them in terms of types and operations from Σ_p (and those defined earlier). The resulting algebraic signature is $\Sigma_p \cup \Sigma_{local}$. The new types and operations must include those of Σ_r, that is $\Sigma_r \subseteq \Sigma_{local}$. If parts of the parameter are to be included in the result, then *<body>* must include the statement ***inherit*** *X* which introduces the types and operations in Σ_p to Σ_{local}. Now, for any structure $A \in \mathrm{Mod}[\Sigma_p]$, *<body>* defines the expansion of the Σ_p-algebra *A* to a $(\Sigma_p \cup \Sigma_{local})$-algebra A_{exp}. *M*(*A*) denotes the reduct of A_{exp} to a Σ_r-algebra, that is, just A_{exp} with the types and operations not in Σ_r omitted.

We deal with hierarchical structures by allowing sorts and operations in algebraic signatures to have names like *X.**name*** where *X* is a structure name and ***name*** is in the signature of *X*. The statement

```
structure X: Σ
```

in a Standard-ML signature introduces *X.**name*** to the signature for each ***name*** in Σ. The statement

```
structure X = Y
```

in a Standard-ML structure or module introduces *X.**name*** for each (type or operation) ***name*** in the signature of *Y*, and gives it the same interpretation as ***name*** had in *Y* (note that ***inherit*** *X* $\equiv_{def}$ ***structure*** *X* = *X*). Moreover, we need to deal with equivalence classes of names rather than names themselves to handle sharing; for example, the sharing declaration

```
sharing X = Y
```

causes *X.**name*** and *Y.**name*** to be identified for each ***name*** in the signature of *X* (which is required to be the same as the signature of *Y*). This allows us to refer to the same object using both names.

For example, the Standard-ML signature

```
signature Σ =
    sig type t
        val c: t
    end
```

denotes the algebraic signature ⟨{[t]},{[c]:[t]}⟩, where we use [...] to list the elements of an equivalence class.

```
signature Σ' =
    sig structure X: Σ and Y: Σ
        sharing X.t = Y.t
        val f: X.t → X.t
    end
```

denotes ⟨{[X.t,Y.t]}, {[X.c]:[X.t,Y.t],[Y.c]:[X.t,Y.t],[f]:[X.t,Y.t]→[X.t,Y.t]}⟩.

```
module M(X: Σ, Y: Σ sharing X.t = Y.t) : Σ'
    inherit X, Y
    val f(a) = if a=X.c then Y.c else X.c
end
```

denotes a function taking any two Σ-algebras *A* and *B* with the same carrier for sort *t* to a Σ'-algebra with both *X.c* and *Y.c* defined as in *A* and *B* (respectively) and *f* defined as in the above code. If

```
structure A: Σ =
    struct type t is bool
           val c = true
    end

structure B: Σ =
    struct type t is bool
           val c = false
    end
```

then *M*(*A*,*B*) has *X.t* = *Y.t* = *bool*, *X.c* = *true*, *Y.c* = *false* and *f* is negation.

3 Extended ML

In the previous section we outlined an algebraic characterization of the notions of signature, structure and module as they appear in Standard ML. In this framework a signature may be viewed as a specification of a class of structures. However, since Standard ML is just a programming language, the information provided by a signature is rather limited. It is sufficient for complete type checking, which is very important and useful in practice to eliminate many simple programming errors, but it is not sufficient, for example, for proving program correctness or for program documentation (except for just giving types of operations). To make signatures more useful for program development and specification we have to extend them to include axioms (for now, sentences of first-order logic, but see section 7) which put constraints on what the operations are supposed to do. We propose to make this extension with the aim of doing formal development and proofs of Standard ML programs. We will call the new language Extended ML.

An example of a signature in Extended ML is the following (we assume that *IntListSig* is a signature with a type *intlist* of lists of integers and a "membership" operation *isin*:*int*×*intlist*→*bool*):

```
signature IntListChooseSig =
    sig structure List : IntListSig
        val choose: List.intlist → int
        axiom (∀l:List.intlist)
                  (l≠List.nil ⇒ List.isin(choose(l),l))
    end
```

Note that the above Extended-ML signature corresponds to what is usually (see e.g. [ADJ 76]) called a specification, i.e. an algebraic signature with some axioms. However, in [ADJ 76] and elsewhere, e.g. [Ehr 79], [EKMP 82], [ETLZ 82], [GM 83], the meaning of a specification was taken to be (up to isomorphism) the *initial* algebra in the class of algebras having the algebraic signature from the specification and satisfying the axioms. An algebraic approach to program modularisation based on these ideas is presented in [Ehrig 84]. We want to consider a different interpretation here: an Extended-ML signature determines a larger class of algebras, all with the same algebraic signature but not necessarily all isomorphic. We postpone the exact definition of this class to the next section. If $\mathfrak{S}$ is an Extended-ML signature then Σ denotes its corresponding algebraic signature and $\mathrm{Mod}[\mathfrak{S}]$ denotes its class of models ($\mathrm{Mod}[\mathfrak{S}]\subseteq\mathrm{Mod}[\Sigma]$).

For any Standard-ML signature we may define a Standard-ML structure over this signature by giving code which defines the data types and operations specified in the signature. The same in Extended ML, but note that Standard-ML code is just a special kind of specification which happens to be executable. In an Extended-ML structure $\mathcal{S}$ over Extended-ML signature $\mathfrak{S}$ we allow arbitrary axioms (over the algebraic signature Σ) to be used as "code". This again determines a class of algebras which we denote by $\mathrm{Mod}[\mathcal{S}]$; namely, $\mathrm{Mod}[\mathcal{S}]$ is the class of all models of $\mathcal{S}$, i.e. those Σ-algebras which satisfy the axioms given in $\mathcal{S}$. For an Extended-ML structure $\mathcal{S}$ to have an Extended-ML signature $\mathfrak{S}$, which we denote $\mathcal{S}:\mathfrak{S}$, means that $\mathrm{Mod}[\mathcal{S}]\subseteq\mathrm{Mod}[\mathfrak{S}]$.

For example, assuming that *List* is a structure with signature *IntListSig*:

```
structure IntListChoose: IntListChooseSig =
    struct
      inherit List
      val choose: List.intlist → int
      axiom choose(List.nil) = 0
      and   (∀l:List.intlist)
                (l≠List.nil ⟹ List.isin(choose(l),l))
    end
```

The situation with Extended-ML modules is similar, although slightly more complicated. Again, in place of code we permit arbitrary axioms to be used. Recall that in Standard ML a module

module $M(X\colon \Sigma_p) : \Sigma_r$
  <body>

denotes a function M:$\mathrm{Mod}[\Sigma_p]\to\mathrm{Mod}[\Sigma_r]$ such that for any Σ_p-algebra A, $M(A)$ is the Σ_r-reduct of the expansion of A defined by *<body>*. In Extended ML, since *<body>* contains axioms rather than code, it need not define an expansion of A unambigously (there may be many different expansions of A which satisfy *<body>*; there may be even none). Thus, in Extended ML a module

module $\mathcal{M}(X\colon \mathfrak{S}_p) : \mathfrak{S}_r$
  <body>

determines in the same way as in Standard ML a function $\mathcal{M}_{\mathbf{base}}$ mapping each Σ_p-algebra to a class of Σ_r-algebras. For Extended-ML module $\mathcal{M}$ as above we require that for any algebra $A\in\mathrm{Mod}[\mathfrak{S}_p]$, $\mathcal{M}_{\mathbf{base}}(A)\subseteq\mathrm{Mod}[\mathfrak{S}_r]$. $\mathcal{M}_{\mathbf{base}}$ extends in an obvious way to a function $\mathcal{M}$ mapping $\mathfrak{S}_p$-structures to $\mathfrak{S}_r$-structures, i.e. for any $\mathfrak{S}_p$-structure $\mathcal{S}$, $\mathrm{Mod}[\mathcal{S}]\subseteq\mathrm{Mod}[\mathfrak{S}_p]$, $\mathcal{M}(\mathcal{S})=\bigcup\{\mathcal{M}_{\mathbf{base}}(A)\mid A\in\mathrm{Mod}[\mathcal{S}]\}$. It follows from the requirement stated above that $\mathcal{M}(\mathcal{S})\subseteq\mathrm{Mod}[\mathfrak{S}_r]$.

4 Behavioural equivalence in program specification

Although according to the previous section signatures and structures in Extended ML both denote classes of algebras (and even have a similar syntax), we want to keep them separate because they play different roles in the process of program development. A signature corresponds to a specification of an abstract data type. It provides information necessary to use the data type, but says nothing about the implementation details. On the other hand, a structure is like an "abstract program" (in the sense of stepwise refinement [Wirth 71]) which implements an abstract data type; it gives some implementation details, although, depending on the stage of development, this information need not be complete.

Since a signature in Extended ML is supposed to be a specification of an abstract data type, it should not distinguish between two algebras which are equivalent from the user's point of view, even if these algebras are "internally" different. This amounts to the requirement that the class of algebras corresponding to a signature should satisfy some kind of "abstractness" condition, i.e. that it should be closed under the relation "equivalent from the user's point of view". [ADJ 76] suggests that "abstract" in "abstract data type" means "up to isomorphism"; however, this is not the abstractness condition for us, since it is easy to give an example of two algebras which are not isomorphic but still are indistinguishable from the user's point of view (e.g. the abstract data type *Stack* can be represented using either a list or else an array with a pointer; the corresponding algebras are not isomorphic but still have exactly the same properties for a user).

We argue (see [SW 83], [ST 84, 85]; cf. [GGM 76]) that the appropriate meaning of "abstract" is "up to behavioural equivalence". The idea is that every abstract data type includes some external (or observable) sorts which are the only ones to which a user has direct access; the remaining sorts may only be manipulated by a user indirectly, via the operations provided by the abstract data type. Two algebras are behaviourally equivalent with respect to a set of observable sorts if and only if they give the same (or, a bit more generally, corresponding) answers to every computation taking inputs of observable sorts and yielding a result of an observable sort. In the algebraic approach such computations correspond to terms of observable sorts with variables of observable sorts. Thus, two algebras are behaviourally equivalent if and only if they satisfy exactly the same equations between such terms. For a more precise definition, technical details and more discussion see [ST 85] (cf. [SW 83], [ST 84]).

A conclusion from the above argument is that an Extended-ML signature denotes the smallest class of algebras which is closed under behavioural equivalence and which contains all the algebras satisfying the axioms given in the signature. Note that this corresponds to an *abstract model specification* [LB 77], where we first define something concretely and then abstract away from the concrete details.

To turn this into a definition we still need to indicate which types in an Extended-ML signature we are going to regard as observable. The natural choice is to take all types which are external to the signature in the sense that they come from somewhere else. These are exactly those types which belong to inherited structures, i.e. in more syntactic terms, these are just the types having names of the form *id.name* for arbitrary *id* and *name*. Note that because of sharing such types may have other names which need not all be of this form. Note also that the standard types like *bool* have such names as well, e.g. *InitialEnv.bool*.

Thus, summing up, any Extended-ML signature $\mathfrak{S}$ determines the class of algebras Mod[$\mathfrak{S}$] consisting of those Σ-algebras which are behaviourally equivalent to some Σ-algebra satisfying the axioms of $\mathfrak{S}$, with respect to the types of Σ belonging to signatures of inherited structures.

For example, assuming that *Triv* is a trivial signature with one type *elem* only:

```
signature StackSig =
   sig structure X: Triv
       type stack
       val empty: stack
       and push: X.elem × stack → stack
       and pop: stack → stack
       and top: stack → X.elem
       and isempty: stack → bool
       axiom (∀a:X.elem,s:stack)(pop(push(a,s)) = s)
       and   (∀a:X.elem,s:stack)(top(push(a,s)) = a)
       and   isempty(empty) = true
       and   (∀a:X.elem,s:stack)
                 (isempty(push(a,s)) = false)
   end
```

Note that the axioms of *StackSig* do not specify the value of e.g. *top(empty)*, so algebras satisfying the axioms need not be isomorphic. Furthermore, because Mod[*StackSig*] is by definition closed under behavioural equivalence with respect to the types *X.elem* and *bool*, it contains not only algebras satisfying the axioms, like the list representation of stacks, but also all algebras which are behaviourally equivalent to them, like the array-with-pointer representation of stacks, which does not satisfy the axiom $(\forall a{:}X.elem, s{:}stack)(pop(push(a,s))=s)$.

5 Program development

From our point of view, program development should proceed as follows:

We begin with some high-level user-oriented specification: this is an Extended-ML signature, say $\mathfrak{S}_0$, typically with a very large class of models. Then, by making a series of design decisions we ultimately arrive at an executable Standard-ML structure, say $\mathcal{S}_{37}$, which satisfies the initial specification (i.e. $\mathcal{S}_{37}{:}\mathfrak{S}_0$). Typically, $\mathcal{S}_{37}$ has exactly one model. Making a design decision is nothing more than restricting the class of models, e.g. by adding an axiom to a specification.

To formalise this we need a notion of implementation:

Definition: A class of algebras C′ *implements* a class of algebras C if C′⊆C. C and C′ must be classes of algebras over the same algebraic signature.

This general definition in a natural way allows us to talk about the implementation of a structure by another structure, of a signature by a structure (which we have been writing as $\mathcal{S}{:}\mathfrak{S}$), of a signature by another signature, or even of a structure by a signature (although we cannot see any obvious need for this last notion).

This also extends to a definition of what it means for a module to implement another module: $\mathcal{M}'(X'{:}\mathfrak{S}'_p){:}\mathfrak{S}'_r$ implements $\mathcal{M}(X{:}\mathfrak{S}_p){:}\mathfrak{S}_r$ if

(1) $\mathfrak{S}_p$ implements $\mathfrak{S}'_p$ (i.e. $\mathcal{M}'$ accepts any argument accepted by $\mathcal{M}$) and

(2) for any structure $\mathcal{S}{:}\mathfrak{S}_p$, $\mathcal{M}'(\mathcal{S})$ implements $\mathcal{M}(\mathcal{S})$.

Note that a sufficient condition for (2) is that the body of $\mathcal{M}'$ implements the body of $\mathcal{M}$; this means that we can implement a module by refining its body, using exactly the same methods as when refining Extended-ML structures.

Now, developing $\mathcal{S}_{37}$ from $\mathfrak{S}_0$ means constructing a series of specifications (signatures or structures) $sp_1, \ldots, sp_{36}$ all over the same algebraic signature such that each one implements the previous one, i.e. $\mathrm{Mod}[\mathfrak{S}_0] \supseteq \mathrm{Mod}[sp_1] \supseteq \ldots \supseteq \mathrm{Mod}[sp_{36}] \supseteq \mathrm{Mod}[\mathcal{S}_{37}]$. Since the implementation relation is transitive (i.e. implementations compose vertically in the terminology of [GB 80]) the correctness of each refinement step guarantees that $\mathcal{S}_{37}$ implements $\mathfrak{S}_0$.

In our framework, a refinement step may be of one of three forms (the fourth, theoretically possible, does not seem very useful as indicated above). The first and simplest case is when we implement a structure $\mathcal{S}_i$ by another structure $\mathcal{S}_{i+1}$. To prove the correctness of such a refinement (i.e. that $\mathcal{S}_{i+1}$ really implements $\mathcal{S}_i$) we need only to show that each axiom of $\mathcal{S}_i$ is a consequence of the axioms of $\mathcal{S}_{i+1}$. The second case is when we implement a signature $\mathfrak{S}_i$ by a structure $\mathcal{S}_{i+1}$. To prove the correctness of this refinement we need to show that every model of $\mathcal{S}_{i+1}$ is behaviourally equivalent to an algebra satisfying the axioms of $\mathfrak{S}_i$. This is difficult in general, but in the next section we give an example of how in practice this may be done. The third case is the implementation of a signature $\mathfrak{S}_i$ by another signature $\mathfrak{S}_{i+1}$. Note that this corresponds to an abstract model specification of a refinement step. Proving such a refinement correct seems even more difficult than in the previous case; note however, that the inherited (i.e. observable) types of $\mathfrak{S}_i$ and $\mathfrak{S}_{i+1}$ are the same and so in practice this may be reduced to proving that any algebra satisfying the axioms of $\mathfrak{S}_{i+1}$ is behaviourally equivalent to an algebra satisfying the axioms of $\mathfrak{S}_i$, as in the previous case.

Note that all these proofs of correctness may and should be supported by a mechanical theorem prover such as the one described in [BoM 79] or LCF [GMW 79].

In the development process as described above we deal with signatures and structures as indivisible entities. We develop a single monolitic structure from a single signature; except that the development may proceed stepwise, there is no notion of "divide and conquer", i.e. splitting the problem into independent subproblems. A way of breaking up the problem is to use modules. In the development process described above we may present any of the specifications shown ($\mathfrak{S}_0$ and $\mathcal{S}_{37}$ as well as the intermediate specifications) as the result of a module application, e.g. $\mathcal{S}_{13}=\mathcal{M}(\mathcal{S},\ldots)$. Note that this splits the problem of implementing $\mathcal{S}_{13}$ into the subproblems:

(1) implementing the module $\mathcal{M}$ itself,
(2) implementing (separately) each of the arguments $\mathcal{S},\ldots$.

It is easy to see that implementations compose horizontally (in the terminology of [GB 80]), i.e. if $\mathcal{S}'$ implements $\mathcal{S}$, ..., and $\mathcal{M}'$ implements $\mathcal{M}$ then $\mathcal{M}'(\mathcal{S}',\ldots)$ implements $\mathcal{M}(\mathcal{S},\ldots)$. This allows the developments of $\mathcal{M}$ and $\mathcal{S},\ldots$ to proceed separately, with the guarantee that the results may be combined to give an implementation of $\mathcal{S}_{13}$ (and so of $\mathfrak{S}_0$ as well). Of course, these (sub)developments may involve themselves further decomposition.

One issue we have so far omitted is the problem of inconsistent specifications. It is easy to write a specification which has no models, and according to our definition such a specification implements any specification over the same algebraic signature. Note, however, that any executable Standard-ML structure is consistent and so if we succeed in implementing a specification by an executable structure then the original specification must have been consistent. This means that checking consistency is not necessary in the development process to ensure correctness; however, an inconsistent specification is a blind alley (worse, it can be refined forever) and so to be cautious it is advisable to check for consistency as far as possible at each stage. Note, however, that even a consistent specification may have no executable implementation and so we cannot in general avoid blind alleys in program development anyway.

6 An example

To show what specifications and program development in Extended ML look like, we give below a perhaps oversimplified but hopefully instructive example. We present the specification and partial development of an interpreter for a very simple programming language with arithmetic expressions, assignment and sequential statements.

We begin with the specification of environments:

```
signature Ident =
  sig type elem
      val eq: elem × elem → bool
      axiom (∀x:elem)(eq(x,x) = true)
      and   (∀x,y:elem)(eq(x,y) = eq(y,x))
      and   (∀x,y,z:elem)
                (eq(x,y)=true & eq(y,z)=true ⇒
                    eq(x,z) = true)
  end

signature EnvSig =
  sig structure Id: Ident
      type env
      val initial: env
      and assign: Id.elem × int × env → env
      and lookup: Id.elem × env → int
      axiom (∀x:Id.elem,n:int,ρ:env)
                (lookup(x,assign(x,n,ρ)) = n)
      and   (∀x,x':Id.elem,n,n':int,ρ:env)
                (Id.eq(x,x')=false ⇒
                    assign(x,n,assign(x',n',ρ))
                      = assign(x',n',assign(x,n,ρ)))
      and   (∀x,x':Id.elem,n,n':int,ρ:env)
                (Id.eq(x,x')=true ⇒
                    assign(x,n,assign(x',n',ρ))
                      = assign(x,n,ρ))
  end
```

EnvSig specifies all algebras which are behaviourally equivalent to algebras satisfying the axioms above, with respect to *Id.elem* and *int* as observable sorts. Note that not all algebras in Mod[*EnvSig*] satisfy the second axiom, but they always satisfy e.g.

```
(∀y,x,x':Id.elem,n,n':int,ρ:env)
    (Id.eq(x,x')=false ⇒
        lookup(y,assign(x,n,assign(x',n',ρ)))
          = lookup(y,assign(x',n',assign(x,n,ρ)))).
```

Having specified environments we can specify (a scheme for) expressions and commands.

```
signature ExpSig =
   sig structure Env: EnvSig
       type exp
       val eval: exp × Env.env → int
   end

signature ComSig =
   sig structure Exp: ExpSig
       type com =
          data assign of Exp.Env.Id.elem × Exp.exp
               | compose of com × com
       val execute: com × Exp.Env.env → Exp.Env.env
       axiom (∀x:Exp.Env.Id.elem,
                e:Exp.exp,
                ρ:Exp.Env.env)
               (execute(assign(x,e),ρ)
                  = Exp.Env.assign(x,Exp.eval(e,ρ),ρ))
       and    (∀c,c':com,ρ:Exp.Env.env)
                 (execute(compose(c,c'),ρ)
                    = execute(c',execute(c,ρ)))
   end
```

The use of ***data*** above not only introduces the type *com* and the operations *assign*: *Exp.Env.Id.elem×Exp.exp→com* and *compose*: *com×com→com*; it also imposes the implicit requirement that every value of type com can be constructed in a unique way using these functions. Formally, this requirement amounts to adding an appropriate *data constraint* as an axiom; for details see [BG 80] (cf. [Rei 80]).

ComSig is independent from the actual syntax of expressions, and so we defined it in terms of the relatively impoverished signature *EnvSig*. The following specification describes a possible syntax for expressions.

```
signature ExpSynSig  =
   sig structure Env: EnvSig
       type exp =
          data const of int
               | var of Env.Id.elem
               | plus of exp × exp
               | cond of exp × exp × exp
       val eval: exp × Env.env → int
       axiom (∀n:int,ρ:Env.env)(eval(const(n),ρ) = n)
       and    (∀x:Env.Id.elem,ρ:Env.env)
                 (eval(var(x),ρ) = Env.lookup(x,ρ))
       and    (∀e,e':exp,ρ:Env.env)
                 (eval(plus(e,e'),ρ)
                    = eval(e,ρ) + eval(e',ρ))
       and    (∀e,e',e'':exp,ρ:Env.env)
                 (eval(cond(e,e',e''),ρ)
                    = eval(e'',ρ)   if eval(e,ρ) = 0
                    = eval(e',ρ)    otherwise)
   end
```

(We use an obvious notation to simplify the syntax of conditional axioms.)

We can now put commands and expressions together to get the final specification of our language.

```
signature LangSig =
   sig structure Com: ComSig and Exp: ExpSynSig
       sharing Com.Exp.exp = Exp.exp
       and     Com.Exp.Env = Exp.Env
       axiom (∀e:Exp.exp,ρ:Exp.Env.env)
                 (Com.Exp.eval(e,ρ) = Exp.eval(e,ρ))
   end
```

Now, to implement LangSig we define:

```
module LangMod (Com: ComSig, Exp: ExpSynSig
                   sharing Com.Exp.exp = Exp.exp
                   and     Com.Exp.Env = Exp.Env)
                     : LangSig
    inherit Com, Exp
    axiom (∀e:Exp.exp,ρ:Exp.Env.env)
              (Com.Exp.eval(e,ρ) = Exp.eval(e,ρ))
end
```

It is easy to see that for any structures *Com:ComSig* and *Exp:ExpSynSig*, *LangMod(Com,Exp)* implements *LangSig*. Moreover, this structure is consistent provided that *Com* and *Exp* have models sharing the type *exp*, the operation *eval* and the substructure *Env* (all names with appropriate prefixes). Thus, we have decomposed the problem of implementing *LangSig* into the subproblems of implementing the parameter signatures and the module *LangMod* itself. The latter task is in fact trivial, as the module introduces neither new types nor new operations. An obvious implementation of *ComSig* is *ComMod(ExpMod(Env))*, where *Env:EnvSig* and *ComMod* and *ExpMod* are defined as follows:

```
module ComMod (Exp: ExpSig) : ComSig
   inherit Exp
   type com =
      data assign of Exp.Env.Id.elem × Exp.exp
           | compose of com × com
   val execute: com × Exp.Env.env → Exp.Env.env
   axiom (∀x:Exp.Env.Id.elem,
            e:Exp.exp,
            ρ:Exp.Env.env)
           (execute(assign(x,e),ρ)
              = Exp.Env.assign(x,Exp.eval(e,ρ),ρ))
   and    (∀c,c':com,ρ:Exp.Env.env)
             (execute(compose(c,c'),ρ)
                = execute(c',execute(c,ρ)))
end

module ExpMod (Env: EnvSig) : ExpSig
   inherit Env
   type exp
   val eval: exp × Env.env → int
end
```

Note that *ComMod* is executable; in the context of the sharing declaration in *LangSig* we can ignore the fact that *ExpMod* is not yet executable.

We could implement *ExpSynSig* in a way analogous to *ComSig* above. However, we would like to take advantage of the fact that Extended-ML signatures specify models up to behavioural equivalence and so our implementation of this part of the language introduces some simple source-code optimisations.

```
module ExpSynMod (Env: EnvSig) : ExpSynSig
   inherit Env
   type exp =
      data constc of int
           | varc of Env.Id.elem
           | plusc of exp × exp
           | condc of exp × exp × exp
   val eval: exp × Env.env → int
   and const: int → exp
   and var: Env.Id.elem → exp
   and plus: exp × exp → exp
   and cond: exp × exp × exp → exp
   axiom (∀n:int,ρ:Env.env)(eval(constc(n),ρ) = n)
   and    (∀x:Env.Id.elem,ρ:Env.env)
              (eval(varc(x),ρ) = Env.lookup(x,ρ))
   and    (∀e,e':exp,ρ:Env.env)
              (eval(plusc(e,e'),ρ)
                 = eval(e,ρ) + eval(e',ρ))
   and    (∀e,e',e'':exp,ρ:Env.env)
              (eval(condc(e,e',e''),ρ)
                 = eval(e'',ρ)  if eval(e,ρ) = 0
                 = eval(e',ρ)   otherwise)
   { *** optimisations start here *** }
   and    (∀e,e':exp)
              (plus(e,e') = e'            if e = const(0)
                          = e             if e' = const(0)
                          = plusc(e,e') otherwise)
   and    (∀e,e',e'':exp)
              (cond(e,e',e'')
                 = e'      if e = const(n) and n ≠ 0
                 = e''     if e = const(0)
                 = e'      if e' = e''
                 = condc(e,e',e'')
                           otherwise)
end
```

We claim that *ExpSynMod* meets its specification, i.e. for any *Env:EnvSig*, *ExpSynMod(Env):ExpSynSig*; in other words *ExpSynMod(Env)* implements *ExpSynSig*.

Sketch of proof: Let $A \in ExpSynMod(Env)$. We have to show that A is behaviourally equivalent to some algebra B satisfying the axioms in *ExpSynSig*, with respect to *Env.env*, *Env.Id.elem* and *int* as observable sorts. By the definition of the semantics of a module, A is a reduct to the algebraic signature of *ExpSynSig* of some algebra A_{exp} which extends *Env* and satisfies the axioms given in the body of *ExpSynMod*. Note that A_{exp} "contains" another algebra of the same signature as A: namely, the algebra in which the operations *const*, *var*, *plus* and *cond* are interpreted as, respectively, *constc*, *varc*, *plusc* and *condc* in A_{exp}. Call this other algebra B. It is easy to see that the axioms in the body of *ExpSynMod* ensure that B satisfies the axioms in *ExpSynSig*.

Now, to show that A and B are behaviourally equivalent it is sufficient to show that the value of any term *term* of an observable sort with variables of observable sorts is the same in A and B. The only nontrivial case here is when *term* is of the form $eval(e,\rho)$, where e is a term of the sort *exp* with variables of sorts *Env.Id.elem* and *int* only. Let e_c denote the term resulting from e by replacing each occurence of *const*, *var* etc. by, respectively, *constc*, *varc* etc. By an easy induction on the complexity of e one may prove that the values of $eval(e,\rho)$ and $eval(e_c,\rho)$ in A_{exp} are the same. To complete the proof it is enough to notice that the value of $eval(e,\rho)$ in A is the same as in A_{exp} and its value in B is the same as the value of $eval(e_c,\rho)$ in A_{exp}. □

Finally, to complete the development we have to implement *EnvSig*. We can start again with a trivial refinement to *EnvMod(Id)*, where *Id:Ident* and

```
module EnvMod (Id: Ident) : EnvSig
   inherit Id
   type env
   val initial: env
   and assign: Id.elem × int × env → env
   and lookup: Id.elem × env → int
   axiom (∀x:Id.elem,n:int,ρ:env)
            (lookup(x,assign(x,n,ρ)) = n)
   and    (∀x,x':Id.elem,n,n':int,ρ:env)
            (Id.eq(x,x')=false ⇒
               assign(x,n,assign(x',n',ρ))
                  = assign(x',n',assign(x,n,ρ)))
   and    (∀x,x':Id.elem,n,n':int,ρ:env)
            (Id.eq(x,x')=true ⇒
               assign(x,n,assign(x',n',ρ))
                  = assign(x,n,ρ))
end
```

This module provides an implementation of environments; however, it is clear that this implementation is not complete in the sense that it does not determine the value of, for example, *lookup(x,initial)*. Thus, we can further refine our specification and implement the module *EnvMod* by

```
module EnvMod' (Id: Ident) : EnvSig
   inherit Id
   type env
   val initial: env
   and assign: Id.elem × int × env → env
   and lookup: Id.elem × env → int
   axiom (∀x:Id.elem,n:int,ρ:env)
            (lookup(x,assign(x,n,ρ)) = n)
   and    (∀x,x':Id.elem,n,n':int,ρ:env)
            (Id.eq(x,x')=false ⇒
               assign(x,n,assign(x',n',ρ))
                  = assign(x',n',assign(x,n,ρ)))
   and    (∀x,x':Id.elem,n,n':int,ρ:env)
            (Id.eq(x,x')=true ⇒
               assign(x,n,assign(x',n',ρ))
                  = assign(x,n,ρ))
   and    (∀x:Env.Id.elem)(lookup(x,initial) = 0)
end
```

It is easy to see that *EnvMod'* implements *EnvMod*, and so *EnvMod'(Id)* implements *EnvSig*. The next refinement of *EnvMod* would probably be to fix a data representation for *env*.

The final step of the development, which we are going to omit, is to implement the signature *Ident* by defining some structure *Id:Ident*.

Summing up, by decomposing our initial specification into smaller pieces using modules and then implementing each of them separately step-by-step we developed an implementation of the signature *LangSig* given by the following structure:

```
LangMod(ComMod(ExpMod(EnvMod'(Id))),
        ExpSynMod(EnvMod'(Id)))
```

The development presented above is, of course, extremely simple. We hope, however, that we have managed to convince the reader that the same ideas may be applied in more complicated situations. For example, we could easily add more kinds of expressions and commands (e.g. while-loops), block structure and input/output. We could replace integers with an arbitrary primitive domain of elementary types and operations. In this more elaborate language more interesting optimisations would be possible.

7 Extended ML over an arbitrary institution

In the previous sections we presented Extended ML based on the standard algebraic framework: we used standard algebraic signatures, standard algebras and standard first-order axioms. However, in practice we may need to use variations of these notions to handle some specific features of ML which we have disregarded here, or to make specifications of some data types easier. For example, to handle polymorphism we would have to deal with polymorphic signatures or with order-sorted signatures and algebras (see [Gog 78], [Gogolla 83]); to handle nontermination we may want to use partial (see [BW 82]) or continuous (see [ADJ 77]) algebras; to deal with exceptions we could use error algebras (see [Gog 77] or [GDLE 82]); finally, we may want to allow (or disallow) different kinds of sentences to be used as axioms. Each of these leads to a formally different logical system for writing specifications in Extended ML.

The notion of *institution* [GB 83] provides a tool for dealing with any of these different logical systems. An institution comprises definitions of signature, model (algebra), sentence and a satisfaction relation satisfying a few consistency conditions. (For a similar but more logic-oriented approach see [Bar 74].)

We can base our definitions of Extended-ML signatures, structures, modules and their semantics on an arbitrary institution, thus avoiding the choice of particular definitions of these underlying notions. Moreover, our notion of implementation and our presentation of program development process remain valid in this general framework as well. For the technical details which underlie such definitions (in particular, a definition of behavioural equivalence in an arbitrary institution) see [ST 84] and [ST 85]. The only concept we used in this paper which did not appear in earlier work on algebraic specification in an arbitrary institution is the notion of executable specification. We can describe this by indicating which sentences of the underlying institution are *executable*. In the standard framework of the institution of first-order logic we use in this paper, we may assume that the executable sentences are just equations of the form *term* = *term'*, where *term* has a certain simple form, as sets of such equations correspond directly to programs in pure applicative ML.

We can define an institution with all the features necessary to specify Standard-ML programs (we need to be able to handle polymorphic higher-order functions which may perform assignments and generate exceptions and which need not terminate). Standard-ML code (or a notational variant) could then be included as sentences of this institution. Extended ML over this institution would be a language suitable for specifying and developing programs in full Standard ML.

8 Conclusions

In this paper we attempted to apply the mathematically well-explored ideas on algebraic specification of [SW 83], [Wir 83], [ST 84] and [ST 85] in the context of the Standard ML programming language. We extended Standard ML by allowing axioms in module interface specifications (signatures) and in place of code; the resulting language is called Extended ML. After describing the algebraic semantics of Extended ML we discussed how it may be used in formal program development. From the specification formalism it is based on, Extended ML inherits complete independence from the logical system (institution) used to write specifications. This also amounts to the independence of Extended ML from the programming language used to write code. With an appropriate change to the underlying institution, Extended "ML" becomes Extended Pascal (i.e. Pascal + modules + specifications), Extended Ada, etc.

There have been few attempts to apply work on algebraic specification to programming in the past. Three exceptions are CIP-L [Bau 81], IOTA [NY 83] and Anna [LHKO 84]. The CIP-L wide-spectrum language and the IOTA language are, like Extended ML, capable of expressing specifications as well as (high- and low-level) programs. But in both cases the semantics of specifications is different. In [Bau 81] although there is a notion of refinement discussed, it is not described formally. In IOTA the notion of refinement is the same as in Extended ML, although because of the different semantics of specifications its application is more restricted there. Anna is mainly oriented towards annotating existing Ada programs, although it is also possible to use it to specify programs prior to their implementation. There is no explicit notion of refinement in Anna. IOTA and Anna lack formal semantics, and all three languages are dependent on some particular logical system and so e.g. CIP-L cannot fully handle higher-order functions, functions with several results, or nondeterministic functions.

One problem with Extended ML is that we have made no provision for building structured specifications as in e.g. Clear [BG 80]. It is possible at the level of modules and structures to break large specifications/programs into small pieces, but it is not possible e.g. to define a signature by means of a "hidden function". This is not a big problem, since our specification formalism (see [ST 84]) provides a rich set of specification-building operations which we have so far only used in defining the semantics of signatures and modules. The only reason at present for forbidding their explicit use in specifications is that they are probably too low-level to be used conveniently. Another point is that in the presence of such powerful specification-building operations as closure under behavioural equivalence (*behavioural abstraction*) the difference between the notions of signature and structure in Extended ML becomes rather fuzzy, since for every signature it is easy to construct a structure having the same class of models. Experience will show if it is worthwhile retaining this terminological and methodological distinction. On the other hand, it is not yet clear whether in practice it is necessary to introduce extra specification-building operations in the context of the powerful modularisation mechanism in Extended ML.

Acknowledgements

The idea of adding axioms to a programming language was suggested by Rod Burstall; this feature was added to the HOPE language [BMS 80] in 1982 for teaching purposes. The present work was stimulated by Robin Milner's work on Standard ML and by David MacQueen's work on modules. Our thanks to Rod Burstall for his constant support and encouragement and to Martin Wirsing for helping to develop the foundations on which Extended ML rests. Support was provided by Edinburgh University and the Science and Engineering Research Council.

9 References

[ADJ 76] Goguen, J.A., Thatcher, J.W. and Wagner, E.G. An initial algebra approach to the specification, correctness, and implementation of abstract data types. IBM research report RC 6487. Also in: Current Trends in Programming Methodology, Vol. 4: Data Structuring (R.T. Yeh, ed.), Prentice-Hall, pp. 80-149 (1978).

[ADJ 77] Goguen, J.A., Thatcher, J.W., Wagner, E.G. and Wright, J.B. Initial algebra semantics and continuous algebras. JACM 24, 1, pp. 68-95.

[Bar 74] Barwise, J. Axioms for abstract model theory. Annals of Math. Logic 7, pp. 221-265.

[Bau 81] Bauer, F.L. *et al* (the CIP Language Group) Report on a wide spectrum language for program specification and development (tentative version). Report TUM-I8104, Technische Univ. München.

[BoM 79] Boyer, R.S. and Moore, J.S. A Computational Logic. Academic Press.

[BW 82] Broy, M. and Wirsing, M. Partial abstract types. Acta Informatica 18, pp. 47-64.

[BG 80] Burstall, R.M. and Goguen, J.A. The semantics of Clear, a specification language. Proc. of Advanced Course on Abstract Software Specifications, Copenhagen. Springer LNCS 86, pp. 292-332.

[BMS 80] Burstall, R.M., MacQueen, D.B. and Sannella, D.T. HOPE: an experimental applicative language. Proc. 1980 LISP Conference, Stanford, California, pp. 136-143.

[Ehr 79] Ehrich, H.-D. On the theory of specification, implementation, and parametrization of abstract data types. Report 82, Abteilung Informatik, Univ. of Dortmund. Also in: JACM 29, 1, pp. 206-227 (1982).

[Ehrig 84] Ehrig, H. An algebraic specification concept for modules (draft version). Report 84-04, Institut für Software und Theoretische Informatik, Technische Univ. Berlin.

[EKMP 82] Ehrig, H., Kreowski, H.-J., Mahr, B. and Padawitz, P. Algebraic implementation of abstract data types. Theoretical Computer Science 20, pp. 209-263.

[ETLZ 82] Ehrig, H., Thatcher, J.W., Lucas, P. and Zilles, S.N. Denotational and initial algebra semantics of the algebraic specification language LOOK. Draft report, IBM research.

[GGM 76] Giarratana, V., Gimona, F. and Montanari, U. Observability concepts in abstract data type specification. Proc. 5th MFCS, Gdansk. Springer LNCS 45.

[Gogolla 83] Gogolla, M. Algebraic specifications with partially ordered sorts and declarations. Fb. 169, Abteilung Informatik, Univ. of Dortmund.

[GDLE 82] Gogolla, M., Drosten, K., Lipeck, U. and Ehrich, H.D. Algebraic and operational semantics of specifications allowing exceptions and errors. Fb. 140, Abteilung Informatik, Univ. of Dortmund.

[Gog 77] Goguen, J.A. Abstract errors for abstract data types. Proc. IFIP Working Conf. on the Formal Description of Programming Concepts, New Brunswick, New Jersey.

[Gog 78] Goguen, J.A. Order sorted algebras: exceptions and error sorts, coercions and overloaded operators. Semantics and Theory of Computation Report No. 14, Dept. of Computer Science, UCLA.

[GB 80] Goguen, J.A. and Burstall, R.M. CAT, a system for the structured elaboration of correct programs from structured specifications. Technical report CSL-118, Computer Science Laboratory, SRI International.

[GB 83] Goguen, J.A. and Burstall, R.M. Introducing institutions. Proc. Logics of Programming Workshop (E. Clarke, ed.), Carnegie-Mellon. University.

[GM 83] Goguen, J.A. and Meseguer, J. An initiality primer. Draft report, SRI International.

[GMW 79] Gordon, M.J., Milner, A.J.R. and Wadsworth, C.P. Edinburgh LCF. LNCS 78.

[Gut 75] Guttag, J.V. The specification and application to programming of abstract data types. Ph.D. thesis, University of Toronto.

[LB 77] Liskov, B.H. and Berzins, V. An appraisal of program specifications. Computation Structures Group memo 141-1, Laboratory for Computer Science, MIT.

[LHKO 84] Luckham, D.C., von Henke, F.W., Krieg-Brückner, B. and Owe, O. Anna: a language for annotating Ada programs (preliminary reference manual). Technical report 84-248, Computer Systems Laboratory, Stanford University.

[MacQ 84] MacQueen, D.B. Modules for Standard ML. Research report, Bell Laboratories; an earlier version appeared in Proc. 1984 ACM Symp. on LISP and Functional Programming, Austin, Texas.

[Mil 78] Milner, R.G. A theory of type polymorphism in programming. Journal of Computer and System Sciences 17, pp. 348-375.

[Mil 84] Milner, R.G. A proposal for Standard ML. Proc. 1984 ACM Symp. on LISP and Functional Programming, Austin, Texas.

[NY 83] Nakajima, R. and Yuasa, T. (eds.) The IOTA Programming System: A Modular Programming Environment. Springer LNCS 160.

[Rei 80] Reichel, H. Initially restricting algebraic theories. Proc. 9th MFCS, Rydzyna. Springer LNCS 88, pp. 504-514.

[ST 84] Sannella, D.T. and Tarlecki, A. Building specifications in an arbitrary institution Proc. Intl. Symp. on Semantics of Data Types, Sophia-Antipolis, France. Springer LNCS 173, pp. 337-356.

[ST 85] Sannella, D.T. and Tarlecki, A. On observational equivalence and algebraic specification. To appear in: Proc. 10th Colloq. on Trees in Algebra and Programming, Berlin (March 1985).

[SW 83] Sannella, D.T. and Wirsing, M. A kernel language for algebraic specification and implementation. Report CSR-131-83, Dept. of Computer Science, Univ. of Edinburgh; extended abstract in: Proc. Intl. Conf. on Foundations of Computation Theory, Borgholm, Sweden. Springer LNCS 158, pp. 413-427.

[Wir 83] Wirsing, M. Structured algebraic specifications: a kernel language. Habilitation thesis, Technische Univ. München.

[Wirth 71] Wirth, N. Program development by stepwise refinement. CACM 14, pp. 221-227