Finite Sets: A Case Study on Formal Program Development in the Extended ML Framework

Cláudia J. A. da Silva
Grupo de Computação, Fundação Instituto Tecnológico de Pernambuco (ITEP)
Av. Prof. Luiz Freire, 700, CEP 50.730
Recife, Pernambuco, Brasil. e-mail: claudia@gctc.itep.br.

Fabio Q. B. da Silva
Departamento de Informática, Universidade Federal de Pernambuco,
Av. Prof. Luiz Freire, s/n, CP 7851, CEP 50732-970,
Recife, Pernambuco, Brasil. e-mail: fabio@di.ufpe.br.

Abstract

We study formal development of functional programs from algebraic specifications in the Extended ML framework. We present a case study on the modular specification and refinement of (finite) set operations in the Extended ML wide-spectrum specification/programming language. Our main objective is to present the module facilities and development methodology of Extended ML, their application to a practical problem, and their suitability for the formal development of (modular) Standard ML programs.

Sumário

Estudamos o desenvolvimento formal de programas funcionais, usando especificações algébricas, em Extended ML. Apresentamos um estudo de caso em especificação modular e refinamento de operações de conjuntos (finitos) usando a linguagem de especificação/programação Extended ML. Nosso principal objetivo é apresentar a linguagem de módulos e a metodologia de desenvolvimento Extended ML, suas aplicações em problemas práticos e sua adequação para o desenvolvimento formal de programas (modulares) em Standard ML.

1 Introduction

As noted in [14], "the ultimate goal of work on program specification is to establish a practical framework for the systematic production of correct programs from requirement specifications via a sequence of verified development steps". We present a modular development of basic operations on sets from a requirement specification of their behaviour in the Extended ML framework [16, 11]. Our goal is to present an extensive use of the modular facilities of the Extended ML language and show how these facilities simplify the program specification and formal development.

Extended ML (EML) is a wide-spectrum specification/programming language for the formal development of Standard ML (SML) [4, 5] programs by means of mathematically verified development steps. Both EML and SML languages have formal semantics describing every aspect of the languages. Therefore, in the EML framework a complete degree of formalisation in the development process is achieved, which can be expressed as follows: the resulting program is considered to be correct with respect to its initial requirement specification if and only if each development step is *proved to be correct* in a formal calculus consistent with the formal semantics of EML and SML.

The EML language is based on the module system of SML, and therefore strongly supports *modular specifications* and the development of *modular programs*. Both aspects are essential for the formal development of large software systems. Modular specifications simplify the complexity of the development steps, by localising design decisions and verification proofs to small, self-contained program units. On the other hand, it is widely accepted that a methodological use of modularity in programming makes large systems easier to maintain and improves the reusability of parts of developed systems. Together, modular specifications of modular programs allow *specifications*, and consequently their *verification proofs*, to be more easily maintained and reused.

Although the design of modular systems have been widely covered in the literature, the problem of modularising formal specifications has received little attention until recently. In the EML framework, modular specifications are natural, and provide powerful abstractions, as already noted in [8]. Furthermore, the use of *observational equivalence* [13] as a correctness criterion in the development process introduces a high degree of flexibility and generality in this framework.

The EML language and development methodology has been extensively studied in the literature. In this paper, EML is briefly introduced by means of examples. The reader unfamiliar with EML, or wishing to understand its mathematical underpinings, is referred to the literature cited in Section 2.2. Our goal in this paper is to show a particular case study and investigate its problems in the context of Extended ML, rather than providing a detailed introduction to the language.

In Section 2, we review the main features of EML and SML. We then present the EML development methodology in Section 3. Sections 2 and 3 are based on previous work [14]. The reader familiar with EML language and methodology might want to skip Sections 2 and 3. In Section 4, we show a complete development of basic set operations using the EML methodology. Finally, Section 5 draws some conclusions and present suggestions for further work.

2 The Extended ML Specification Language

Extended ML (EML) is a wide spectrum specification/programming language for formal development of programs in the functional language Standard ML [4, 5]. EML is called wide spectrum because it can express all stages of a development process in a single framework: the high-level *specification*, the executable Standard ML *program*, and *abstract code* which contains both non-executable specifications and Standard ML programs.

EML is a proper extension of the programming language Standard ML (SML). Before introducing this extension we present the main aspects of SML that are necessary for understanding later examples. This presentation is necessarily short, and is given mainly to make this work self contained. The reader is referred to [4, 9] for didactical accounts of SML, and to [5] for the language's formal definition.

2.1 An Overview of Standard ML

SML has two distincts sub-languages: the *Core* and the *Module* languages. The SML Core language provides features for programming "in the small". The Core is an eager functional programming language, with polymorphic types, a strong type system which allows static type inference, user defined concrete and abstract data types, a mechanism to raise and handle run time exceptions, and imperative features like references and assignment. Programs in the SML Core language resembles other functional programming languages, e.g., HOPE [1]. The following example illustrates a small subset of the SML Core features, including the pre-defined data type of polymorphic lists.

Let us represent finite sets of integer numbers as lists of integers, and implement SML functions for constructing sets and to test for set membership.

```
type intset = int list
val empty : intset = nil
fun add(a, s) = a :: s
fun member(a, empty) = false
  | member(a, b::s) = (a = b) orelse member(a, s)
```

The constant `empty` represents the empty set and is implemented by the (pre-defined) constant `nil` (the empty list). The function `add` simply builds sets using the list function `::` (read *cons*). The function `member` is defined by case analysis on the structure of sets. The first clause states that the empty set has no elements, and the second clause says that `a` is a member of a set `b::s` if `a` is equal `b` or else `a` is a member of `s`. Many SML features are not reviewed here, including record types, high-order functions, imperative features, exceptions, concrete and abstract data types (the module language provides a more flexible way of encapsulating data).

The SML *Module language* provides features for programming "in the large". Using these features large SML programs can be structured in small self-contained programs (called *structures*) with an interface (called a *signature*). Interfaces may be explicitly given by the programmer or inferred by the type inference system. *Functors* are parameterised structures with an explicit *input signature* and an *output signature*. Applying a

functor to a structure matching its input signature yields a structure matching its output signature. Hereafter, we use the term *module* to refer to both functor and structure, whenever it does not cause ambiguities.

In the following example, we will define sets of objects of an arbitrary type, provided this type admits equality on its objects[1]. This is accomplished by the definition of a functor together with its input and output signatures.

```
signature ELEM = sig  eqtype elem  end

signature SET =
 sig
  structure Elem : ELEM
  type set
  val member : Elem.elem * set -> bool
  val add    : Elem.elem * set -> set
 end

functor Set(X : ELEM) : sig include SET
                            sharing Elem = X
                        end =
 struct
  structure Elem = X
  type set = Elem.elem list
  val empty = nil
  fun add(a, s) = a :: s
  fun member(a, nil) = false
    | member(a, b :: s) = (a = b) orelse member(a, s)
 end

structure IntElem : ELEM = struct  type elem = int  end

structure IntSet = Set(IntElem)
```

This defines a functor **Set** with input signature **ELEM** and output signature **SET**. When applied to a structure matching **ELEM**, e.g. **IntELem**, **Set** yields a structure matching **SET**. The Module language also has a strong type system that allows signature inference. For the definition of **Set** to be correctly typed, its body must define at least the objects that are specified in the output signature: a sub-structure called **Elem** matching the signature **ELEM**; a constant **empty**; and functions **add** and **member**, with their types as specified in **SET**. The sharing constraint forces the (actual) parameter structure **X** and the resulting sub-structure **Elem** to be the same object.

The function **IntSet.add** constructs sets of integer numbers using the constant **empty** and the integer constants. The function **IntSet.member** can be used to test membership on sets constructed by **IntSet.add**.

Signatures play a dual rôle in the Module language. On the one hand, they act as an interface, restricting the external view of internal module components. Therefore, only what is explicitly specified in the signature is visible from outside of a module[2]. On the other hand, signatures impose constraints on which components must be defined in the body of the modules, together with their types, i.e., they define the minimal set of components that must be defined in the module's body.

For instance, in the above example the signature **SET** requires the existence in **Set** of a sub-structure **Elem**, a constant **empty**, and two functions **add** and **member**. Furthermore, the users of **IntSet** cannot make use of the internal representation of sets as lists. This is to say that **IntSet** implements an abstract data type and the only visible operations on this type are those defined in the signature. The encapsulation provided by the Module language is essential to the development of large systems structured in small modules, and this is the basis of the EML modular program development methodology, which is reviewed in Section 3.

The above overview of the SML modules is necessarily short. In [9], the interested reader will find a more detailed introduction to the Module language.

[1] An SML type admits equality if it is not an abstract data type or it is not a function type.

[2] This is not true, in general, for SML, but it holds for EML module language.

2.2 An Overview of Extended ML

In this section, we give a brief overview of the Extended ML specification/programming language. We only summarise the EML features that are necessary to the understanding of later examples. The reader unfamiliar with EML language is referred to the literature cited below.

As noted in Section 2.1, in the SML Module language signatures act as interfaces to modules, defining which objects can be externally accessed, together with their types. This information is sufficient to use SML as a programming language. However, a signature does not, in general, provide enough information to be considered as a specification of a programming task. In order to specify a function completely and non-ambiguously we need its type and a formal description of its input/output behaviour.

In EML, *axioms* are allowed in signatures to enhance their information content concerning the input/output behaviour of the functions. That is, axioms provide a formal definition of the components of the modules. Axioms are written in a notation of first-order equational logics in which equality is admitted on all types. For instance, we may extend the signature **SET** with the specification of the functions **add** and **member** as follows:

```
signature SET =
 sig
  structure Elem : ELEM
  type set
  val empty : set
  val add   : Elem.elem * set -> set
  val member : Elem.elem * set -> bool
  axiom forall a => member(a, empty) = false
  axiom forall a, b, x => member(a, add(b, s)) = (a = b) orelse member(a, s)
 end
```

For a structure to match, or *satisfy*, the above signature, it must supply a sub-structure **Elem** matching **ELEM**, a constant **empty** and the functions **add** and **member**, with their types as in **SET**, and furthermore the structure's body must *satisfy* the axioms in the signature. Satisfaction in EML is taken to be up to *observational equivalence*, i.e., the body of the structure does not have to satisfy the axioms in the signature exactly, but only with respect to its observable behaviour. The use of observational satisfaction instead of ordinary logic satisfaction is a point of major importance in the EML framework, which allows a high degree of flexibility in the implementation. However, a more detailed discussion on observational equivalence is outside the scope of this paper. The interested reader might want to see [13] for a detailed account of observational equivalence in the context of EML, and [17] for a more general approach to this issue.

Signatures extended with axioms constitute a specification of a program task, that is, we must construct a functor or structure satisfying this signature. The development of such a functor or structure include stages in which some functions are not yet defined and the body of other structures and functors may contain axioms. In order to allow these stages to be expressed in EML, structure bodies are allowed to include a mixture of *axioms*, *incomplete expressions* and SML *code*. A possible stage in the development of the functor **Set** may be as follows:

```
functor Set(structure X : ELEM) : sig  include SET
                                       sharing X = Elem
                                  end =
 struct
  structure Elem = X
  type set = Elem.elem list
  val empty : set = nil
  val add : Elem.elem * set -> set = ?
  val memeber : Elem.elem * set -> bool = ?
  axiom forall a => member(a, empty) = false
  axiom forall a, b, x => member(a, add(b, s)) = (a = b) orelse member(a, s)
 end
```

The place-holder expression ? is used above to declare functions which are not yet in an executable form. In SML, the type of an expression, and in particular a function, can be inferred from context information. In EML, the type of non-executable functions declared using ? must be explicitly given, as in the above declarations of

add and `member`. The expression ? can also be used to declare a type whose representation is not yet defined, and in place of functors or structures bodies. The latter use of the place-holder expression is useful to specify the initial programming task. For instance:

```
functor Set(X : ELEM) : sig include SET
                            sharing X = Elem
                        end = ?
```

The above functor specification is an EML specification of a *programming task*. Program development aims to suply an executable body to replace the structure expression ?.

The EML language is the extension to SML as described above: axioms are allowed in signatures to specify the behaviour of module components; the place-holder expression ? is allowed in place of a type expression, a value expression or a structure expression; and axioms are allowed in the body of structures and functor to specify the behaviour of variables and functions declared using ?. Hereafter, we will assume that all axioms are implicitly universally quantified, whenever this does not cause ambiguities.

It is important to emphasise that the choice of the logics and the notation to write axioms is, at some extent, arbitrary. The EML semantics is parametric on such a choice. The particular choice of first-order equational logics is convenient since an SML variable or function declaration is a particular form of equation that happens to be executable. Therefore, axioms and declarations are written and interpreted in an uniform framework.

An extensive literature exists covering many aspects of the EML language and methodology: [11, 10] give detailed introductions to EML in a tutorial form; the original semantics of EML, based on the theory of institutions [3], appears in [15, 16]; [12] defines the syntax and some aspects of the semantics of EML; the full semantics, including signature matching extended to deal with observational satisfaction of axioms is given in [6]; [7] provides a gentle introduction to this formal semantics.

3 The Extended ML Development Methodology

In this section, we review the EML development methodology. The starting point of a (formal) development process in EML is a functor of the form:

```
functor F(X : SIGin) : SIGout = ?
```

where `SIGin` and `SIGout` are EML signatures, possibly containing axioms. From this initial specification we should work towards a functor in which an (executable) SML structure expression replaces ?. The development is carried out in various *formally verified development steps*, which ensures that the final executable functor is correct with respect to its initial specification.

In general, any Extended ML functor specification having a body consisting of ? or a non-trivial body with non-executable components is regarded as a programming task. In the former case, this task is to replace ? by a non-trivial (and potentially non-executable) body that satisfies the functor's signature. In the latter case, the task is to replaces axioms in the functor's body by SML function or value declarations.

The EML development methodology describes three ways to proceed from a specification or abstract program towards a (more concrete) program:

Decomposition step: decompose a functor in other functors, which will then be regarded as new programming tasks.

Coding step: provide a non-trivial functor body to replace ?, which may contain axioms and components declared using ?.

Refinement step: replace a non-trivial functor body by another (more concrete) body in which some axioms and declarations using ? are replaces by concrete SML declarations.

Decomposition steps are used to structure the development into smaller tasks, and can be considered as programming, or development, "in the large". Coding and refinement can therefore be considered as programming, or development, "in the small". The use of each development step is illustrate in a case study in Section 4.

Each development step gives rise to one or more correctness conditions, or *proof obligations*. These conditions or obligations can be derived automatically from a development step. A development is considered to be correct if each proof obligation generated during the development is formally verified, or *discharged*. In general, a proof

obligation is a relation between the state of the development before and after a development step. The generic form of a proof obligation is an *observational* satisfaction relation:

$$SP_1 \cup \ldots \cup SP_n \models_{OBS} SP$$

where $SP_i, 1 \leq i \leq n$, are signatures or structure expressions after the development step has taken place, SP is the specification before the development step, and OBS is a set of the (observable) types of SP (actually a subset of the types of SP). Verifying or discharging a proof obligation involves showing that the axioms in SP logically follows from the axioms and declarations in the union of the left hand side of the relation, up to observational equivalence with respect to OBS.

Proofs of observational equivalence are notoriously difficult (see [17] for a detailed discussion on this problem). Since ordinary logical satisfaction is stronger than observational equivalence, it is sufficient to prove that $SP_1 \cup \ldots \cup SP_n \models SP$, if it holds. For the examples of next section, ordinary logical satisfaction will be used, since it holds for each proof obligation during the development process.

Let us now detail each development step and show the proof obligations they give rise:

Decomposition step: given a functor specification

```
functor F(X : SIGin0) : SIGout0 = ?
```

We can decompose `F` into two[3] functors `G` and `H`, as follows:

```
functor G(X : SIGin1) : SIGout1 = ?
functor H(X : SIGin2) : SIGout2 = ?
```

where the new definition of `F` is as follows:

```
functor F(X : SIGin0) : SIGout0 = G(H(X))
```

The new definition of `F` must be a well-formed EML functor definition, regarding syntax and type checking. Furthermore, this decomposition generates three proof obligations:

$$\texttt{SIGin0} \models_{OBS} \texttt{SIGin2} \qquad \texttt{SIGout2} \models_{OBS} \texttt{SIGin1} \qquad \texttt{SIGout1} \models_{OBS} \texttt{SIGout0}$$

Coding step: given a functor specification:

```
functor F(X : SIGin) : SIGout = ?
```

a coding step provides a non-trivial body to replace `?`:

```
functor F(X : SIGin) : SIGout = strexp
```

where `strexp` must be a well formed EML structure expression. This coding step generates one proof obligation:

$$\texttt{SIGin} \cup \texttt{strexp} \models_{OBS} \texttt{SIGout}$$

Refinement step: given a functor specification:

```
functor F(X : SIGin) : SIGout = strexp
```

a refinement step replaces `strexp` by another (more concrete) structure expression:

```
functor F(X : SIGin) : SIGout = strexp'
```

where `strexp'` must be a well formed EML structure expression. This refinement step generates one proof obligation:

$$\texttt{SIGin} \cup \texttt{strexp'} \models_{OBS} \texttt{strexp}$$

According to the methodology described above, every step in a development has the following structure:

- *Design Decision*: a choice of which development step to use and how to execute the chosen step.
- *The step proper*: the new state of the development that arises after the development step.
- *Verification*: the proofs that discharge the obligations generated by the development step.

The following case study illustrates the use of the EML development methodology.

[3] In general, we can decompose `F` in $n > 0$ functors; this generalisation is trivial and can be found in the EML literature cited above.

4 A Case Study: Finite Sets

In this section we present the steps of the development of an SML program from the initial requirements specification to the final executable program. Our objective is to illustrate the EML development methodology and show how it can assist in structuring the development of large software systems. The program consists of several functions that work on an abstract data type of finite sets, called `set`. We will provide a constant value `empty` to represent the empty set and the following functions:

- `singleton`: convert an element of type `elem` into a set of type `set` with only this element in it.
- `add`: adds an element to a set, if it is not already there.
- `delete`: deletes an element from a set.
- `union`: the union of two sets.
- `intersection`: intersection of two sets.
- `difference`: the set difference between two sets.
- `member`: verifies if an element belongs to a set.
- `members`: returns an `elem list` that contains all the elements of a set.
- `cardinality`: returns the number of elements that are in a set.

Two other functions, `inlist` and `twice`, are also part of this program. The function `inlist` verifies if an element belongs to a list, the function `twice` verifies if an element occurs twice in a list. These functions are used to simplify the specification of `cardinality`, as will be shown below.

4.1 Developing the SML Program

The development steps that construct one implementation of the program (informally) described above are given. The development has a form of a (finitely branching) tree, the *development tree*, whose nodes represent design decisions. The development tree for this case study is shown in Figure 1.

Step 0 *Specification.* The initial formal specification of the program described above is given by an Extended ML functor specification that has an input signature `ELEM` and an output signature `SET` as follows:

```
signature ELEM =
sig
  eqtype elem
  val leq : elem * elem -> bool
  axiom leq(a,b)
  axiom leq(a,b) andalso leq(b,a) implies (a=b)
  axiom leq(a,b) andalso leq(b,c) implies leq(a,c)
end;

signature SET =
sig
  type elem
  type set
  val empty : set
  val singleton : elem -> set
  val union : set * set -> set
  val add : set * elem -> set
  val intersection : set * set -> set
  val member : elem * set -> bool
  val members : set -> elem list
  val cardinality : set -> int
  val difference : set * set -> set
  val delete : set * elem -> set
  val inlist : elem * elem list -> bool
  val twice : elem * elem list -> bool
  axiom inlist(a,nil) = false
  axiom inlist(a,b::l) = (a=b) orelse inlist(a,l)
  axiom twice(a,nil) = false
  axiom twice(a,b::l) = (a=b) andalso inlist(a,l) orelse twice(a,l)
  axiom member(a,empty) = false
```

```
  axiom member(b,singleton a) = (a=b)
  axiom member(a,union(s,t)) = member(a,s) orelse member(a,t))
  axiom member(a,s) = inlist(a,members s)
  axiom twice(a,members s) = false
  axiom cardinality s = length(members s)
  axiom add(s,a) = union(s,singleton a)
  axiom member(a,intersection(s,t)) = member(a,s) andalso member(a,t)
  axiom member(a,difference(s,t)) = member(a,s) andalso not(member(a,t))
  axiom delete(s,a) = difference(s, singleton a)
end;

functor Set(X : ELEM) : SET = ?
```

The specification of **cardinality** uses the built in function **length**, whose behaviour we can assume to be correct. The use of **twice** guarantees that there are no duplicates in a list generated using **members**, and therefore simplifies the implementation of **cardinality**. It is important to notice that other specifications of **cardinality**, e.g., the usual set theoretical definition, may not require the use of **twice**.

The parameter to the program is a structure that defines a type **elem** that must admit equality, and a function **leq** that defines a partial order on the elements of this type. The implementation of the program admits any choice for **elem** and **leq** that satisfies the axioms in **ELEM**. This abstraction makes the development more general because the choice of representation for **elem** and an actual implementation of **leq** are only necessary by the time the functor **Set** is used.

Step 1 *Design decision: Decomposition.* Implement the functions **add** and **delete** in terms of the functions **union** and **difference**. This step reduces the number of functions that must be effectively implemented, because the implementations of **add** and **delete** will follow trivially from the axioms in **SET**. We need two new functors and a new signature:

```
signature SET' =
sig
  structure Elem : ELEM
  open Elem
  (* the rest of this signature is as SET without the declarations of the type
   elem, the specifications of add and delete, and the axioms that
   mention add and delete *)
end;

functor AddDelete(S : SET') : SET = ?
functor Set'(X : ELEM) : sig include SET'
                             sharing Elem = X
                         end = ?
```

We can then implement **Set** in terms of **AddDelete** and **Set'** as follows:

```
functor Set(X : ELEM) : SET = AddDelete(Set'(X))
```

Verification We must show that **AddDelete(Set'(X))** is a well formed EML structure expression and that the interfaces match, as defined in Section 3, for the decomposition step . The structure expression is trivially well formed. This property can always be automatically checked by a type checker. The proof obligations that arise from the signature matchings are trivially discharged since all signatures match exactly. Hereafter, we will only justify non-trivial proof obligations. In particular, we will assume that each structure expression is well formed.

We now have a choice of which functor to code. This means to follow either the right or the left branch at the root of the development tree of Figure 1.

Step 2 *Design decision: Coding.* Implement the functor **AddDelete** by replacing **?** by a non trivial structure expression.

```
functor AddDelete(S : SET') : SET =
struct
  open S
```

```
  type elem
  fun add(s : set, e : elem) = ? : set
  fun delete(s : set, e : elem) = ? : set
  axiom add(s,a) = union(s,singleton a)
  axiom delete(s,a) = difference(s, singleton a)
end;
```

Verification Typechecks correctly. We have to show that: SET' ∪ body ⊨ SET.
Proof All the axioms in SET appear in the body of the functor or in the signature SET'.

It is important to remember that SML function declarations are a particular form of axiom in which the left hand side is a linear pattern. Therefore, the axioms in the body of AddDelete can be trivially converted into SML function declarations, which is done in the next development step

Step 3 *Design decision: Refinement.* Convert the axioms for add and delete into SML code. Note that the SML code for these functions is very similar to the axioms. This makes the verification phase trivial, and even possible to be performed automatically.

```
functor AddDelete(S : SET') : SET =
struct
  open S
  type elem = Elem.elem
  fun add(s,a) = union(s,singleton a)
  fun delete(s,a) = difference(s, singleton a)
end;
```

Verification Typechecks correctly. We have to show that: SET' ∪ current body ⊨ previous body.
Proof All the axioms in the previous version of the body follows directly from the definition of the functions in the current body.

At this stage, the body of AddDelete contains only executable SML code. We can now start the development of the functor Set'.

Step 4 *Design decision: Decomposition.* At this stage we need to choose a representation for the type set. Some possibilities are: binary trees, balanced binary trees, unordered lists, and ordered lists. We decided to implement set in terms of ordered lists (using leq) of objects of type elem without duplicates[4]. We delay the actual definition of the functions in Set' in terms of the functions on lists to a later stage in the development. At this point, we proceed by another decomposition and we need two new functors and a new signature:

```
signature ORDDUP =
sig
  structure Elem : ELEM
  open Elem
  val inlist : elem * elem list -> bool
  val twice : elem * elem list -> bool
  val appendlist : elem list * elem list -> elem list
  val inter : elem list * elem list -> elem list
  val dif : elem list * elem list -> elem list
  axiom appendlist(nil, t) = t
  axiom appendlist(s, nil) = s
  axiom (a = b) implies appendlist(a::s,b::t) = a::appendlist(s,t)
  axiom leq(a,b) andalso not(a=b) implies appendlist(a::s,b::t) = a::appendlist(s,b::t)
  axiom leq(b,a) andalso not(a=b) implies appendlist(a::s,b::t) = b::appendlist(a::s,t)
  axiom dif(nil, t) = nil
  axiom dif(s, nil) = s
  axiom inlist(a,t) implies dif(a::s,t) = dif(s,t)
  axiom not(inlist(a,h)) implies dif(a::s,h) = a::dif(s,h)
  axiom (s = nil) orelse (t = nil) implies inter(s,t) = nil
  axiom (a = b) implies inter(a::s,b::t) = a::inter(s,t)
  axiom leq(a,b) andalso not(a=b) implies inter(a::s,b::t) = inter(s,b::t)
  axiom leq(b,a) andalso not(a=b) implies inter(a::s,b::t) = inter(a::s,t)
```

[4]The development of this data structure in EML was already part of a library of simple case studies, and we reused previous development and some of its proofs.

```
  axiom inlist(a,nil) = false
  axiom inlist(a,b::l) = (a=b) orelse inlist(a,l)
  axiom twice(a,nil) = false
  axiom twice(a,b::l) = (a=b) andalso inlist(a,l) orelse twice(a,l)
end;

functor Set''(X : ORDDUP) : sig include SET'
                              sharing X.Elem = Elem
                            end = ?
functor OrdDup(Y : ELEM) : sig include ORDDUP
                             sharing Elem = Y
                           end = ?
```

The facts below are trivial consequence of the axioms in ORDDUP, and are used without proof.

Fact 1: The axioms for appendlist ensure that lists built using nil and appendlist contains no duplicates and their elements are ordered by leq.

Fact 2: The axioms for dif and inter ensure that the lists built using these functions are ordered and without duplicates, provided that their parameters have these properties.

We then implement Set' in terms of Set'' and OrdDup as follows:

```
functor Set'(X : Elem) : SET' = Set''(OrdDup(X))
```

Verification Typechecks correctly. All interfaces match exactly.

We have now another branch in the development tree, and must choose between implementing OrdDup or Set''. Experience in the development of this case study has shown that starting with the inner most functor in a decomposition branch makes it simpler to backtrack whenever a problem is found later in the development. However, a more detailed study is necessary to see whether this is a general design rule.

Step 5 *Design decision: Coding.* Implement the functor OrdDup.

```
functor OrdDup(X : ELEM) : sig include ORDDUP
                             sharing Elem = X
                           end =
struct
  structure Elem : ELEM
  open Elem
  fun inlist(a, l : elem list) = ? : bool
  fun twice(a, l : elem list) = ? : bool
  fun appendlist(s : elem list,t : elem list) = ? : elem list
  fun inter(s : elem list,t : elem list) = ? : elem list
  fun dif(s : elem list,t : elem list) = ? : elem list
  (* include here all the axioms that appear in ORDDUP *)
end;
```

Verification Typechecks correctly. We have to show that: ELEM ∪ body ⊨ ORDDUP.

Proof All the axioms in ORDDUP appear in the body of the functor.

Step 6 *Design decision: Refinement.* We proceed refining OrdDup by converting the axioms for inlist and twice into SML code, which can be trivially derived from their axioms. The current version of the body of the functor OrdDup is obtained from the body of OrdDup in Step 5 by substituting the declarations and axioms for inlist and twice by the following function declarations:

```
  fun inlist(a,nil) = false
    | inlist(a,b::t) = (a=b) orelse inlist(a,t)
  fun twice(a,nil) = false
    | twice(a,b::t) = (a=b) andalso inlist(a,t) orelse twice(a,t)
```

Verification Typechecks correctly. We have to show that: ELEM ∪ current body ⊨ previous body

Proof The axioms for inlist and twice in the previous version of the body follow directly from the definition of the functions in the current version of the body. The other axioms in the previous version of the body appear in the current version of the body.

Step 7 *Design decision: Refinement.* We still proceed refining `OrdDup` by converting the axioms for `appendlist`, `inter`, and `dif` into SML code, which can be trivially derived from their axioms and by making the case analysis explicit using conditional expressions. The current version of the body of the functor `OrdDup` is obtained from the body of `OrdDup` in **Step 6** by substituting the declarations and axioms for `appendlist`, `inter` and `dif` by the following function declarations:

```
fun appendlist(nil,t) = t
  | appendlist(s,nil) = s
  | appendlist(a::s,b::t) = if (a=b) then a::appendlist(s,t)
                            else if (leq(a, b)) then a::appendlist(s,b::t)
                                                else b::appendlist(a::s,t)
fun dif(nil,t) = nil
  | dif(s,nil) = s
  | dif(a::s,t) = if inlist(a,t) then dif(s,t) else a::dif(s,t)
fun inter(nil,t) = nil
  | inter(s,nil) = nil
  | inter(a::s,b::t) = (if (a=b) then a::inter(s,t)
                        else if (leq(a, b)) then inter(s,b::t)
                                            else inter(a::s,t))
```

Verification Typechecks correctly. We have to show that: `ELEM` ∪ `current body` ⊨ `previous body`.
Proof The axioms for `appendlist`, `inter`, and `dif` in the previous version of the body follow directly from the definition of the functions in the current version of the body.

`OrdDup` is already an executable SML functor and we can start coding the functor `Set''`. Another possibility would be to optimise the functions in `OrdDup`. For this particular program, optimisation is not an interesting task, but might be relevant for larger programs.

Step 8 *Design decision: Coding.* Implement the functor `Set''`.

```
functor Set''(X : ORDDUP) : sig include SET'
                                sharing Elem = S.Elem
                            end =
struct
  structure Elem : ELEM = S.Elem
  open S
  type elem = Elem.elem
  type set = elem list
  val empty = nil
  fun singleton(a) = a::nil
  fun union(s: set,t : set) = ? : set
  fun intersection(s : set,t : set) = ? : set
  fun member(a, s : set) = ? : bool
  fun members(s : set) = ? : elem list
  fun cardinality(s : set) = ? : int
  fun difference(s : set,t : set) = ? : set
  (* include here all the axioms that are in SET' *)
end;
```

Verification Typechecks correctly. We have to show that: `ORDDUP` ∪ `body` ⊨ `SET'`.
Proof The axioms for `inlist` and `twice` in `SET` appear in the signature `ORDDUP`. The concrete representations of `empty` and `singleton` trivially satisfy the constraints that lists used to represent sets are ordered and contain no duplicates. All the other axioms in `SET` appear in the body of the functor.

In the following development steps, we will refine `Set''` until an executable functor is obtained. We proceed in small steps, coding only one function at each of the steps 10, 11 and 12, to isolate the proof obligations and consequently their proofs.

Step 9 *Design decision: Refinement.* Implement the functions `members`, `member` and `cardinality`. The current version of the body of functor `Set''` is obtained from the body in **Step 8** by substituting the declarations and axioms for `members`, `member` and `cardinality` for the following function declarations:

```
  fun members s = s
  fun cardinality s = length s
  fun member(a,s) = inlist(a,s)
```

Verification Typechecks correctly. We have to show that: `ORDDUP ∪ current body ⊨ previous body`. We have to prove that the following axioms are implied by the axioms and functions in `ORDDUP ∪ current body`.

1. `axiom member(a,empty) = false`
2. `axiom member(b,singleton a) = (a=b)`
3. `axiom cardinality s = length(members s)`
4. `axiom member(a,s) = inlist(a,members s)`
5. `axiom twice(a,members s) = false`

Proofs

1. From the definitions of `member` and `empty`, we conclude that: `member(a,empty) = inlist(a,nil)`. From the axioms for `inlist`, we conclude that `inlist(a,nil) = false`.
2. From the definitions of `singleton` and `member`, we conclude that `member(b,singleton a) = inlist(b, a::nil)`. From the axioms for `inlist` in `ORDDUP`, we conclude that `inlist(b, a::nil) = (a=b)`.
3. Follows directly from the definitions of `members` and `cardinality`.
4. Follows directly from the definitions for `members` and `member`.
5. Using the definition of `members`, we rewrite axiom 5 as `twice(a,s) = false`. It is important to notice that `s` is a set, and could only have been constructed using the set operations `add`, `delete`, `intersection`, `difference`, and `union`. This allows **Fact 1** and **Fact 2** to be used in the proof below.

   Proof This proof is by structural induction on `s`:

   • **Base case:** `s ≡ nil`

   From the definition of `twice`, we conclude that `twice(a,nil) = false`. Therefore axiom 5 follows.

   • **Inductive Step:** `s ≡ b::l`

   Inductive Hypothesis: `twice(a,l) = false`. From the definition of `twice`, we conclude that `twice(a,b::l) = (a=b) andalso inlist(a,l) orelse twice(a,l)`. Then, from the inductive hypothesis, we conclude that `twice(a,b::l) = (a=b) andalso inlist(a,l) orelse false`.

   Case Analysis:

   - `a=b`: since the lists are constructed using `nil`, `appendlist`, `dif` and `inter`, then **Fact 1** and **Fact 2** apply, i.e. lists have no duplicates. Therefore, `inlist(a,l) = false` and the axiom is satisfied.
   - `not(a=b)` then the axiom follows directly.

Step 10 *Design decision: Refinement.* Implement the function `difference`, in terms of the previously defined function `dif`. The current version of the body of the functor `Set''` is obtained from the body of `Set''` in **Step 9** by substituting the declaration and axiom for `difference` by the following function declaration:

```
fun difference(s,t) = dif(s,t)
```

Verification Typechecks correctly. We have to show that: `ORDDUP ∪ current body ⊨ previous body`. We have to prove that the following axiom is implied by the axioms and functions in `ORDDUP ∪ current body`:

```
axiom forall a, s, t => member(a,difference(s,t)) = member(a,s) andalso not(member(a,t))
```

Using the definitions of `difference` and `member`, we rewrite the axiom as

```
axiom forall a, s, t => inlist(a,dif(s,t)) = inlist(a,s) andalso not(inlist(a,t))
```

Proof This proof is by structural induction on `s`:

• **Base case:** `s ≡ nil`

In this case we have `inlist(a,dif(nil,t)) = inlist(a,nil) andalso not(inlist(a,t))`.

Using the axioms for `dif` in `ORDDUP`, we rewrite the above equation as

`inlist(a,nil) = inlist(a,nil) andalso not(inlist(a,t))`.

Then, from the axioms for `inlist` in `ORDDUP`, we conclude that

`false = false andalso not(inlist(a,t))`. Therefore, for `s = nil` the axiom follows.

- **Inductive Step:** `s ≡ b::l`

  Inductive Hypothesis: `forall t => inlist(a,dif(l,t)) = inlist(a,l) andalso not(inlist(a,t))`

  In this case we have `inlist(a,dif(b::l,t)) = inlist(a,b::l) andalso not(inlist(a,t))`.

  - **case 1:** `inlist(b,t) = true`

    Using the axioms for `dif` in `ORDDUP`, we rewrite
    `inlist(a,dif(b::l,t)) = inlist(a,b::l) andalso not(inlist(a,t))`
    as `inlist(a,dif(l,t)) = inlist(a,b::l) andalso not(inlist(a,t))`.
    Then, from the axioms for `inlist` in `ORDDUP`, we conclude that
    `inlist(a,dif(l,t)) = ((a=b) orelse inlist(a,l)) andalso not(inlist(a,t))`.
    Therefore this case follows from the inductive hypothesis.
  - **case 2:** `inlist(b,t) = false`

    Using the axioms for `dif` in `ORDDUP`, we rewrite
    `inlist(a,dif(b::l,t)) = inlist(a,b::l) andalso not(inlist(a,t))` as
    `inlist(a,b::dif(l,t)) = inlist(a,b::l) andalso not(inlist(a,t))`.
    Then, from the axioms for `inlist` in `ORDDUP`, we conclude that
    `((a=b) orelse inlist(a,dif(l,t))) = ((a=b) orelse inlist(a,l)) andalso not(inlist(a,t))`.
    Case Analysis:
    - `a=b`: from the hypothesis we have that `inlist(a,t) = false`, and the axiom follows trivially.
    - `not(a=b)`: the axiom follows from the inductive hypothesis.

Step 11 *Design decision: Refinement.* Implement the function `intersection` in terms of `inter`. The current version of the body of the functor `Set''` is obtained from the body of `Set''` in **Step 10**, by substituting the declaration and axiom for `intersection` for the following function declaration:

```
fun intersection(s,t) = inter(s,t)
```

Verification Typechecks correctly. We have to show that: `ORDDUP ∪ current body ⊨ previous body`.
We have to prove that the following axiom is implied by the axioms and functions in `ORDDUP ∪ current body`:

```
axiom member(a,intersection(s,t)) = member(a,s) andalso member(a,t)
```

Proof (Sketch) The proof follows by mutual structural induction on `s` and `t`, and by case analysis on the list elements. The complete proof is omited here and can be found elsewhere [2].

Step 12 *Design decision: Refinement.* Implement the function `union` in terms of `appendlist`. The current version of the body of the functor `Set''` is obtained from the body of `Set''` in **Step 11** by substituting the declaration and axiom for `union` for the following function declaration:

```
fun union(s,t) = appendlist(s,t)
```

Verification Typechecks correctly. We have to show that: `ORDDUP ∪ current body ⊨ previous body`.
We have to prove that the following axiom is implied by the axioms and functions in `ORDDUP ∪ current body`:

```
axiom member(a,union(s,t)) = member(a,s) orelse member(a,t)
```

Proof (Sketch) This proof follows similarly as the proof in **Step 11** and can also be found in [2]

Now, all the functor and structure bodies are expressed in SML code, therefore we have finished the development of this program. The Development Tree showing the dependency between the development steps is give in Figure 1. The final SML code for the program appears elsewhere [2].

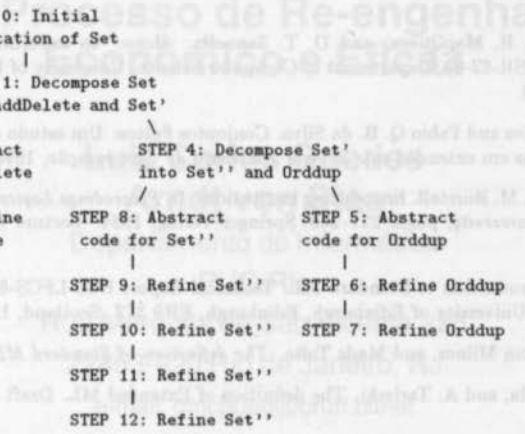

Figure 1: The Development Tree: the dependency between the development steps.

5 Concluding Remarks

We presented a modular development of basic set operations on a data structure of generic finite sets, using the Extended ML specification/programming language and development methodology. The final result of the development process is an SML modular program consisting of various signatures, structures and functors, appears elsewhere [2].

This development illustrated that modular specifications are important to simplify, or even make possible, the development of large systems. Proofs of correctness may be localised in specific stages of the development making it easier to understand the various design decisions. For instance, the proofs of steps 10, 11 and 12 in Section 4.1 are clearer to understand when they are done in separate development steps. This also help in changing the history of the development, e.g., when a "mistake" is found in the original requirement specification or in some later stage.

Another aspect in which modular specifications are important is in isolating the choice of representation for sets and consequently also isolating the proof obligations that arise from such a choice. Therefore, a change in data representation, e.g., to use binary trees, would only require to change the step 4 (including some verifications) of the development, provide we kept the signature `ORDDUP` unchanged. All other parts of the program would remain unchanged, including the verification proofs.

We believe that the development of large software systems require a modular specification/programming language, a powerful development methodology well founded in solid mathematical basis, and a set of software tools to free the programmer from clerical task and also to prevent the possibilities of human errors that would invalidate the entire development process. The EML framework already provide such a language and development methodology. A set of supporting tools are under design and implementation at the Department of Computer Science, University of Edinburgh, Scotland, under the supervision of D. Sannella, and at "Departamento de Informática", UFPE, Brazil, under the supervision of Fabio Q. B. da Silva. These tools include a parser and type-checker for EML specifications and a proof obligation generator, and will form the basis of a complete software development environment for the EML language in the near future. The design and implementation of such an environment is clearly a challenging problem for future research.

Acknowledgements

The authors would like to thank Don Sannella, from University of Edinburgh, Scotland, for many helpful suggestions on previous versions of the case study on Finite Sets. Cláudia J. A. da Silva is supported by a Brazilian government scholarship, CNPq process number 300015/93-3. Fabio Q. B. da Silva is supported by a Brazilian government scholarship, CNPq process number 301557/92-6.

References

[1] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope: an experimental applicative language. Technical Report CSR-62-80, Department of Computer Science, University of Edinburgh, Edinburgh, EH9 3JZ, Scotland, 1980.

[2] Cláudia J. A. da Silva and Fabio Q. B. da Silva. Conjuntos finitos: Um estudo de caso em desenvolvimento formal de programas em extended ml. *Revista Brasileira de Computação*, 1993. To appear.

[3] J. A. Goguen and R. M. Burstall. Introducing institutions. In *Proceedings Logics of Programming Workshop, Carnegie-Mellon University*, pages 221–256. Springer-Verlag, 1984. Lecture Notes in Computer Science, 164.

[4] Robert Harper. Introduction to Standard ML. Technical Report ECS-LFCS-86-14, LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, EH9 3JZ, Scotland, 1986.

[5] Robert Harper, Robin Milner, and Mads Tofte. *The definition of Standard ML*. MIT Press, 1990.

[6] S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML. Draft report, Univ. of Edinburgh, 1993.

[7] S. Kahrs, D. Sannella, and A. Tarlecki. The semantics of Extended ML: a gentle introduction. Draft report, Univ. of Edinburgh, 1993.

[8] Edmund Kazmierczak. Modularizing the specification of a small data base system in Extended ML. Technical Report ECS-LFCS-91-177, LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, EH9 3JZ, Scotland, 1991.

[9] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1992.

[10] D. Sannella. Formal specification of ML programs. In *Jornadas Rank Xerox Sobre Inteligencia Artificial Razionamento Automatizado*, Blanes, Spain, pages 79–98, 1987.

[11] D. Sannella. Formal program development in Extended ML for the working programmer. In *3rd Workshop on Refinement*, Hursley Park, January 1990. BCS/FACS. To appear.

[12] D. Sannella, Fabio Q. B. da Silva, and A. Tarlecki. Syntax, typechecking and dynamic semantics for extended ML (version 2). Technical report, LFCS, Department of Computer Science, University of Edinburgh, 1990. Version 1 appeared as Report ECS-LFCS-89-101, Univ. of Edinburgh (1989).

[13] D. Sannella and A. Tarleck. On observational equivalence and algebraic specification. *Journal of Computer and System Sciences*, 34:150–178, 1987.

[14] D. Sannella and A. Tarleck. Extended ML: present, past and future. Technical Report ECS-LFCS-91-138, LFCS, Department of Computer Science, University of Edinburgh, Edinburgh, EH9 3JZ, Scotland, 1991. Also in Proc. 7th Workshop on Specification of Abstract Data Types, Wusterhausen, GDR (Springer Lecture Notes in Computer Science).

[15] D. Sannella and A. Tarlecki. Extended ML: an institution-independent framework for formal program development. In *Workshop on Category Theory and Computer Programming*, pages 364–389, Guildford, September 1986. Springer Lecture Notes in Computer Science. Vol. 240 (1986).

[16] D. Sannella and A. Tarlecki. Toward formal development of ML programs: foundations and methodology. Technical Report ECS-LFCS-89-71, LFCS, Department of Computer Science, University of Edinburgh, February 1989. Extended abstract in Proc. Intl. Colloq. on Current Issues in Programming Languages, Barcelona, Springer Lecture Notes in Computer Science. Vol. 352, 1989.

[17] Oliver Schoett. Behavioural correctness of data representation. *Science of Computer Programming*, 14:43–57, 1990.