The Semantics of Extended ML: A Gentle Introduction

Stefan Kahrs*
Laboratory for Foundations of Computer Science, Edinburgh University
Edinburgh, Scotland

Donald Sannella†
Laboratory for Foundations of Computer Science, Edinburgh University
Edinburgh, Scotland

Andrzej Tarlecki‡
Institute of Informatics, Warsaw University, and
Institute of Computer Science, Polish Academy of Sciences,
Warsaw, Poland

Abstract

Extended ML (EML) is a framework for the formal development of modular Standard ML (SML) software systems. Development commences with a specification of the behaviour required and proceeds via a sequence of partial solutions until a complete solution, an executable SML program, is obtained. All stages in this development process are expressed in the EML specification language, an extension of SML with axioms for describing properties of module components.

This is a report on the current state of the semantics of the EML specification language as it nears completion. EML is unusual in being built around a "real" programming language having a formal semantics. Interesting and complex problems arise both from the nature of this relationship and from interactions between the features of the language.

1 Introduction

Extended ML (EML) is a framework for the formal development of modular Standard ML (SML) software systems which are correct with respect to a specification of their required behaviour. The long-term goal of work on EML is to provide a practical framework for formal development together with an integrated suite of computer-based specification and development support tools and complete mathematical foundations to substantiate claims of correctness. A short-term subgoal is to complete the formal definition of the semantics of the EML specification language [14], in order to provide a basis for further research

*This research was supported by SERC grants GR/E78463, GR/H73103 and GR/J07303.

†This research was supported by SERC grants GR/E78463, GR/J07693, a SERC Advanced Fellowship, and the COMPASS Basic Research working group.

‡This research was supported by SERC grants GR/H76739 and GR/E78463, an EC-funded COST fellowship, and KBN grant PB 1247/P3/93/04.

on foundations and tools. This paper is a report on the current state of this definition as it nears completion.

SML is a widely-used functional programming language. Apart from useful features it shares with a number of similar languages (a flexible type system with polymorphic types, function definition by patterns, etc.) it has two special characteristics which make it very well-suited to the enterprise mentioned above. First, it provides state-of-the-art modularisation facilities for building large software systems by defining and combining self-contained generic program units. Such facilities seem to be a prerequisite for the use of formal development methods on examples of significant size. The main emphasis of EML is on development "in the large", relying heavily on linguistic support from the SML module facilities and incorporating ideas from foundational work on specification and formal development of modular systems [37], [33], [30], [36]. Second, the syntax and semantics of SML is formally defined [22]. This makes it possible (at least in principle) to reason formally about the behaviour of SML programs, as required for proofs of correctness with respect to a specification of requirements. The size and complexity of the semantics is such that fully formal use of it, e.g. to prove correctness of an optimizing transformation, would be quite a difficult task. Nevertheless, the semantics is small and elegant enough that such use seems not to be completely out of the question.

The idea of building a fully-fledged specification and formal development framework around a "real" programming language seems to be novel to EML. Somewhat related is work on the Anna language for annotating Ada programs with assertions concerning their intended behaviour [19]; but this is not intended for formal development of software from specifications (although see [17]), and as far as we are aware there is no formal semantics of Anna nor any intention to formally relate Anna to the semantics of Ada [2]. Similar comments apply to Larch [10], which has been used in connection with various programming languages. Attempts to apply Larch to the specification of SML modules have recently begun [39], but this work is still at an early stage and many problems remain to be solved. Real programming languages are inevitably complex, and any serious attempt to give a formal treatment of such a language and a development framework based on it is an ambitious goal bringing a host of problems which do not arise when considering toy programming languages or when considering specification and formal development in abstract terms.

Another novelty of this work is in its treatment of the specification of a number of "difficult" facets of computation, all of which arise in SML. These include polymorphic types, higher-order functions, exceptions and non-termination. In spite of the fact that these are common features of modern programming languages, they are rarely addressed by approaches to specification. There have been attempts to treat each of these features in isolation, but not in combination with one another. It is precisely in the interaction between such features that some of the most difficult issues arise.

The structure of the paper is as follows. Section 2 gives a short introduction to the main features of SML and EML in order to set the scene for the rest of the paper. We have resisted the temptation to dwell at length on aspects of EML which are not directly relevant to the topic at hand; for more information, see the papers cited in Section 2. Section 3 discusses the way in which EML relates to and extends SML. Section 4 is an overview of the semantics of EML which attempts to give the reader an overall impression of its structure without

the need to study the details of [14], while touching on the ideas behind many of the most interesting and important points. Section 5 concludes with some remarks about the trials and tribulations involved in writing such a semantics.

2 An overview of EML

The main aim of this section is to provide enough background concerning EML to make the paper self-contained. The first subsection is a summary of the features of the SML programming language, which is the target of EML formal program development and on which EML is based. The next subsection gives an overview of the EML language and formal development framework. A small example is given to demonstrate some of the features of the language, and a final subsection summarizes the main features of the logic used to write axioms.

2.1 SML

The following is necessarily very brief. Readers with no prior knowledge of SML or related languages (Hope, Haskell, etc.) will probably find it necessary to consult e.g. [11] or [24].

SML consists of two sub-languages: the *core* language and the *module* language. The core language provides constructs for programming "in the small" by defining a collection of types and values (including functions) of those types. The module language provides constructs for programming "in the large" by defining and combining a number of self-contained program units coded using the core. To a large extent, these sub-languages can be understood separately from each other, both because the dependency is only one-way (modules contain core constructs, but not vice versa) and because the constructs available in the module language are applicable to the organization of declarations of any kind. SML is an interactive language in which top-level declarations are typechecked, compiled and evaluated one at a time.

The SML core language is a strongly typed functional programming language with a flexible type system including polymorphic types, disjoint union, product and (higher-order) function types, recursive types, and user-defined abstract and concrete types. Conceptually, all values in SML (except those of certain special built-in types, such as `real` and function types) are represented as finite ground terms built from uninterpreted *constructors*. A function is defined by a sequence of equations, each of which specifies the value of the function over some subset of the set of possible argument values. This subset is described by a *pattern* (a term containing constructors and variables only, without repeated variables) on the left-hand side of the equation, which serves both for case selection and variable binding. Certain types are designated by SML as *equality types*; roughly, these are types whose definitions do not involve abstract types or function types. The built-in equality function `=` has type `''a * ''a -> bool`; the type variable `''a` can only be instantiated to equality types (in contrast to `'a` which can be instantiated to any type), preventing values of non-equality types from being tested for equality. Exceptions (possibly carrying values) may be raised by built-in functions (e.g. division by zero) or by user code. Once raised, an exception propagates until it is trapped by a surrounding handler or reaches top level. Typed references are available

with dereferencing and assignment operations. Input/output is handled via streams; input streams are associated with producers (e.g. a keyboard or a file) and output streams are associated with consumers.

The SML module language provides mechanisms that allow large SML programs to be structured into self-contained program units with explicit interfaces. Under this scheme, interfaces (*signatures*) and their implementations (*structures*) are defined separately. Structures contain definitions of types, values and exceptions, and may also contain definitions of lower-level structures (*substructures*). Signatures may be attached to structures; this imposes a requirement for the structure to *match* that signature, meaning that the structure must define types, values, exceptions and substructures with the names indicated by the signature, and the types of values and exceptions as well as the signatures of substructures must correspond to those given in the signature. *Functors* are "parameterized" structures; the application of a functor to a structure yields a structure. A functor has an input signature describing structures to which it may be applied, and an output signature describing the structure which results from such an application. It is possible, and sometimes necessary to allow interaction between different parts of a program, to declare that certain substructures (or just certain types) are identical or *shared*. Structures and functors are referred to collectively as *modules*.

Signatures serve both to impose constraints on the bodies of modules and to restrict the information which is made available externally about the components of module bodies. Roughly speaking, only the information that is explicitly recorded in the signature(s) of a module is available externally. (In fact, this statement is not accurate for SML, but it *is* accurate in the context of EML. See Section 3 for more on this point.) Such information hiding is vital to allow parts of a large software system to be developed and maintained independently.

2.2 EML

EML is a vehicle for the formal development of programs from specifications by means of individually-verified steps. EML is called a *wide-spectrum* language [4] since it allows all stages in the formal development process to be expressed in a single formalism, from the initial high-level specification to the final program itself and including intermediate stages in which specification and program are intermingled. The target of the formal development process is a modular program in SML, and thus (a large subset of) SML is an executable sub-language of EML. Earlier stages in the development of such a program are incomplete modular programs in which some parts are only specified by means of axioms rather than defined in an executable fashion by means of SML code.

Syntactically, the main difference between SML and EML is that EML permits axioms to be included in signatures and in module bodies. Including axioms in signatures allows properties to be specified which are required to hold of any structure matching that signature. The general idea is similar to that of providing *types* of values in signatures in addition to their *names*; the difference is that types (and sharing constraints) can be checked mechanically, while checking that axioms are satisfied requires the use of a theorem prover (and human ingenuity). One reason for including types of values in an SML signature is to provide enough information about the module it describes to enable subsequent code which refers to it to be typechecked and compiled

without making reference to the details of the code in the module body. This is essential for purposes of separate compilation. Similarly, a reason for including axioms in an EML signature is to provide enough information about the module it describes to enable *properties* of such subsequent code to be *proved* without reference to the module body. This separation of an interface from its implementation permits different implementations (satisfying the axioms in the interface) to be developed and used later without affecting the correctness of the rest of the system, and enables implementations for different modules to be developed independently.

Axioms in module bodies may be used to describe components for which executable definitions (in the form of SML code) are not yet available. Syntactically, one gives a declaration containing the place-holder expression "`?`", followed by axioms referring to the undefined object. For example:

```
val x:int = ?
axiom x>7 andalso isprime x
```

Module bodies containing axioms may be regarded as unfinished or incomplete *abstract programs* in which some decisions have already been taken but others, such as choice of algorithms, remain open. The intention is that at a later stage in the development of the program, the question mark will be replaced by code that satisfies the axioms. In the first version of a module declaration, a question mark may be used as a place-holder for the entire module body.

In EML, each structure comes equipped with a signature (this is optional in SML) containing the information which is available externally concerning the structure body. As in SML, the body is required to match this signature. In addition to the name/type matching required in SML, the body must be *correct*: the axioms in the signature must be satisfied by any model of the body (that is, by any structure containing the code in the structure body and satisfying any axioms it includes). Obviously, a proof is generally required to establish correctness. Following ideas concerning the use of axioms to specify encapsulated abstractions (see e.g. [26], [9], [32]), the axioms in the signature need not actually be satisfied "literally": it is enough if they are satisfied "up to behavioural equivalence". Briefly, this means that for any model of the structure, there must be some structure satisfying the axioms in the signature from which the model cannot be distinguished by performing computations that yield observable results (i.e. results of base types such as `bool`). Similar remarks apply to functor declarations, which must contain both an input signature (also required in SML) and an output signature (optional in SML); in this case, all models of the functor body which extend "literal" models of the input signature are required to satisfy the output signature up to behavioural equivalence. See Section 4.3 for some further details, and see [34] for more on the role of behavioural equivalence in the context of EML.

Formal development of a system begins with an initial high-level specification of the problem to be solved, in the form of an EML module declaration having a question mark in place of its body. If the module is parameterized (i.e., is a functor) the input signature specifies the facilities (types, values, exceptions, and structures) to be taken as given, in addition to the built-ins of SML. The output signature of the module specifies the additional facilities required. These signatures will normally contain axioms. At later stages of development, this module declaration will be refined by providing it with a body

which is correct in the sense described above. This may contain axioms, and may make reference to further structures or functors that are themselves not yet defined in an executable fashion. The development process is finished once all functor and structure bodies on which the original "goal" module depends are *complete*, meaning that all question marks and axioms in module bodies have been replaced by executable SML code. At this point, erasing all axioms from signatures (or, much more usefully, regarding them as complete and formally checked documentation) yields an executable SML program. This is correct with respect to the initial specification since correctness is maintained by each development step.[1]

The EML formal development methodology defines a number of ways of gradually refining an unfinished module declaration towards a complete and correct version. A common way to proceed is to decompose the problem into simpler problems by specifying a number of new modules and defining the module at hand as a composition of these. The task of providing a body for each of these new modules becomes a refinement task in its own right which can be tackled separately from the others. Such steps give rise to proof obligations which must be proved in order to ensure that correctness is preserved; these proof obligations can be generated mechanically from the "before" and "after" versions of the module at hand. See [34], [35], [15] and [28] for further details, and see [34], [27] and [29] for examples of EML-style formal software development.

2.3 An example in EML

The example in Figure 1 illustrates some of the language features of EML. It is an implementation of evaluation for a rewrite system, based on some simple abstract properties one would expect for arbitrary rewrite systems, (enriched) λ-calculi, etc. This takes the form of a functor, where properties required of the argument and properties of the result are specified by EML axioms. The functor itself is coded in the executable subset of EML, so this is an example of what might emerge from a formal development which began with a specification of the problem consisting of the same functor with its body replaced by the place-holder "`?`".

The idea of the example is as follows. Rewrite systems operate on some set of terms; each term is either a normal form (`NF`) or contains a redex that can be contracted. A (one-step) strategy picks a redex in a term and returns the redex together with the context of its occurrence in the term, given as a function. The functor `Reduce` provides a function `eval` which repeatedly contracts redexes selected by the given strategy until a term in normal form is obtained. A copy of the argument structure `L` is included as a substructure `T` of the result in order to provide convenient access to the type of terms. `T` inherits the signature of `L` (`TERMSIG`).

The signature `TERMSIG` imposes certain requirements on the behaviour of `NF` and `strategy`: the axiom `forall t => (NF t) proper` is true if the evaluation of `NF t` neither fails to terminate nor raises an exception; for `strategy`

[1] To be completely accurate, it must be mentioned that the compilation of the resulting program is not guaranteed to terminate: EML copes gracefully with non-terminating functions, as explained below, but not with non-terminating *declarations*. The guarantee of correctness is subject to this proviso.

```
signature TERMSIG =
    sig
        type term
        val contract: term -> term
        val NF: term -> bool
        axiom forall t => (NF t) proper
        val strategy: term -> term * (term -> term)
        exception noredex
        axiom forall t =>
            if NF t then (strategy t) raises noredex
            else ((strategy t) proper andalso
                    let val (u,f) = strategy t
                    in f u == t andalso
                       (f (contract u)) proper
                    end)
    end;

signature EVAL =
    sig
        structure T: TERMSIG
        val eval: T.term -> T.term
        axiom forall t =>
            (eval t) terminates implies T.NF(eval t)
    end;

functor Reduce (L: TERMSIG) :
    sig include EVAL; sharing L=T end =
    struct structure T = L
           fun eval t =
               if L.NF t then t
               else let val (redex,context) = L.strategy t
                    in eval (context (L.contract redex))
                    end
    end;
```

Figure 1: An example: evaluation for a rewrite system

there are even stronger conditions, for example that the redex created by `strategy` can be properly contracted, and that `strategy t` raises an exception if and only if `t` is in normal form. Typical for EML is here the mixture of logical connectives and programming language constructs.

The functor `Reduce` gives us an evaluation function `eval`, as specified in the "included" signature `EVAL`, for any rewrite system matching `TERMSIG`. From the interface of `TERMSIG` and the implementation of `eval` we can show that it will never raise an exception (although it may fail to terminate). The sharing equation, an SML feature, is needed to ensure that the type `T.term` used in the type of `eval` is the same as the type `L.term` provided by the argument of `Reduce`, so evaluation is for the kind of terms defined by the argument and not for some other kind of terms. It also makes `eval` applicable to terms other than the ones that can be built using structure `T` only. This is quite desirable, as structure `T` contains no functions for building terms, except by contraction of other terms; normally, the argument of `Reduce` (or structures on which it depends) will contain such functions, in addition to those required by `TERMSIG`.

2.4 The language of EML axioms

The syntax used to write axioms in the above example should have been sufficiently self-explanatory to make the intended meaning clear. However, the logical system used is not a conventional one; it is necessarily much more complex than (for example) many-sorted equational logic or first-order predicate logic because of the need to deal with all the features of SML programs. For example, consider an equation asserting that the values of two expressions, *exp* and *exp′*, are equal. What if either *exp* or *exp′* (or both) fail to terminate? What if one raises an exception (or in the terminology of the SML definition, evaluates to a *packet*)? What if *exp* and *exp′* are of a function type? And in the case of universally and existentially quantified formulae, what is the meaning of quantification over a polymorphic type?

The syntax of EML axioms is designed to be a natural extension of the syntax of EML boolean expressions, with the meaning of the new constructs chosen to be as simple and natural as possible under the circumstances. We have attempted to maximize expressive power, and to avoid making certain common specification idioms unduly awkward to write.

Any expression of type `bool` may be used as an axiom in EML. Such use amounts to an assertion that the expression evaluates[2] to the value `true` rather than evaluating to the value `false`, or evaluating to a packet, or failing to terminate. Hence, the basic connectives are those of SML: `andalso`, `orelse`, and `not`, with the additional connective `implies`. The first two of these have the same "sequential" interpretation as they do in SML (and analogously for `implies`), so for example the expression `true orelse` *exp* evaluates to `true` even if *exp* produces a packet or fails to terminate.

A "logical" equality predicate `==` complements the "computational" equality `=` provided by SML. The expression *exp*`==`*exp′* is well-formed whenever *exp* and *exp′* have the same type, in contrast to *exp*`=`*exp′* which requires this to be an equality type. Logical equality is extensional equality in "logical-relation style"

[2] Actually, *verificates* — see Section 4.3.

[23] on function types, meaning that if f, f' are both of type $\tau \to \tau'$ then f`==`f' is defined as

`forall (x:`τ`,x':`τ`) => x==x' implies (`f` x)==(`f'` x')`

— see below for the meaning of quantification. It is also "extensional" for packets and non-termination: exp`==`exp is `true` even if exp produces a packet or fails to terminate. For any expression exp, additional atomic formulae are:

exp `terminates`, which is `true` if exp produces a normal value or a packet, and `false` if it fails to terminate;

exp `proper`, which is `true` if exp produces a normal value, and `false` if it produces a packet or fails to terminate; and

exp `raises` $excon$,[3] which is `true` if exp raises the exception $excon$ and `false` if it produces a normal value or raises a different exception. If exp fails to terminate then so does exp `raises` $excon$.

Universal and existential quantification is provided over all SML types; function types are included here so this gives a form of higher-order logic, although since quantification ranges over values that are *expressible* in SML, it is not true higher-order quantification. The meaning of quantification over polymorphic types is a tricky issue. An easy choice would be to require explicit quantification of type variables, as in System F [8], but this seems contrary to the spirit of SML in which all such quantification is implicit. The best balance seems to be struck by viewing a quantified formula as having a defined value only if it has that value for all instances (including polymorphic instances) of the type of the bound variable. More explicitly, this amounts to the following four cases:

- In order for `forall` x`:`τ `=>` exp to be `true`, the expression $exp[x := v]$ must be `true` for every expressible value v of every instance of τ.
- In order for `exists` x`:`τ `=>` exp to be `true`, there must be an expressible value v of type τ such that $exp[x := v]$ is `true`. (Note that this is stronger than requiring such a v of some instance of τ.)
- In order for `forall` x`:`τ `=>` exp to be `false`, there must be an expressible value v of type τ such that $exp[x := v]$ is `false`.
- In order for `exists` x`:`τ `=>` exp to be `false`, the expression $exp[x := v]$ must be `false` for every expressible value v of every instance of τ.

Note that the third and fourth cases above are obtained from the second and first cases respectively using the de Morgan laws ($\forall x.\varphi = \neg\exists x.\neg\varphi$, and $\exists x.\varphi = \neg\forall x.\neg\varphi$). The value of a quantified expression is left undefined if none of the above applies, so for example `forall` x`:`τ `=>` exp has no value if $exp[x := v]$ is `false` for some expressible value v of some instance of τ, but there is no expressible value v of type τ itself such that $exp[x := v]$ is `false`.

An example of a formula involving polymorphic quantification that is `true` for some type instances but `false` for others is the following:

[3] In fact, this is a special case of a slightly more general form.

```
forall (x,xs) => [x] @ xs == xs @ [x]
```

where `@` is concatenation of lists and `[x]` is a singleton list containing `x`. One might expect the value of this formula to be `false`, since this is what happens when (for example) `x:int` and `xs:int list`. But when `x:unit` (the type having just one value, written `()`) and `xs:unit list`, the value of the formula is `true` since lists of type `unit list` are uniquely determined by their length. As a consequence, this formula has no value whatsoever. Fortunately, such odd examples occur rarely. A positive example of a polymorphic formula that holds is

```
forall xs => exists ys => xs @ ys == ys @ xs
```

because for any list type, the empty list has the property required for `ys`. The type quantification is left implicit.

A similar but slightly different semantics for quantifiers is considered by Kazmierczak in [16].

Datatype declarations in SML can be seen as carrying logical content. For example, consider the declaration:

```
datatype t = c1 | c2 of int
```

Apart from declaring a new type `t` which is different from all previously-defined types, a constant value `c1:t` and a function value `c2:int->t`, this makes the following assertions (the terminology is due to [5]):

"No junk": The only values of type `t` are `c1` and `(c2` n`)` for integer values n:

```
forall x:t => (x == c1 orelse exists n:int => x == c2 n)
```

"No confusion": The values produced by different constructors are different, and each constructor function is injective and total:

```
forall n:int => not(c1 == c2 n)
forall (n:int,n':int) => (c2 n == c2 n' implies n == n')
forall n:int => c2 n proper
```

"No junk" corresponds to an induction principle for the new datatype; in the case of recursive datatype declarations, this is necessarily a higher-order formula. Both conditions are necessary for the use of constructors in patterns. EML provides a new form of declaration which has syntax similar to that of datatype declarations, but which only asserts "no junk":

```
spantype ''a set = empty | singleton of ''a
                         | union of ''a set * ''a set
```

Here we are specifying that all sets are either empty, or singletons, or unions of such sets, but we are not saying (for example) whether union is commutative or not; if such a property is required, an axiom can be added to impose it. In this paper, the term "axiom" refers to spantypes (although they are syntactically quite different from axioms) as well as to ordinary axioms.

3 The relationship between SML and EML

The EML language was very deliberately designed as a language for specifying modular SML software systems. In contrast to much related work, the intention was *not* to create a completely general-purpose specification language. One of the main guiding principles of the design was to make EML a *minimal* extension to SML. The addition of axioms was clearly necessary to enable module properties to be specified, but we have attempted to keep the syntax of axioms simple and have resisted the temptation to add features or to repair minor defects in the design of SML. For example, EML does not include parameterised specifications (functions from signatures to signatures), despite the fact that these are commonly provided by other specification languages. We have not yet seen a compelling need to add parameterised specifications to EML. In fact, it has become clear to us [30] that what is really important in formal software development is the ability to specify parameterised program modules (i.e. SML functors), and EML already has this facility: one uses an EML functor declaration having a question mark in place of a body.

There are at least four senses in which EML is a minimal extension of SML. First, the syntax of EML minimally extends the syntax of SML. As already stated, the main syntactic extension is the addition of axioms. Second, the semantics of EML is based directly on the semantics of SML, as will be explained in detail in the next section. This is to ensure consistency with SML "by construction" — the fact that significant portions of the two semantic definitions are identical would make a proof of consistency considerably simpler than otherwise. Our initial attempts to give a semantics of EML took quite a different and much more "algebraic" route [31]; we have temporarily abandoned this approach, in part because of the difficulty of ensuring consistency with the existing definition of SML (but see [16]). A third and related point is that the extension to the semantics of SML is such that the semantics of the SML fragment of EML is preserved, making EML a "conservative" extension of SML. This is vital to ensure that the end-product of EML formal development can be compiled and run using existing implementations of SML without modification. Finally, we have attempted to preserve the *spirit* of SML in the extensions insofar as this is possible. This is a necessarily vague statement, but there was already an example of this in Section 2.4 where we eschew the use of explicit quantification of type variables in axioms because such quantification is always left implicit in SML.

In spite of the above, EML is not quite an extension of SML; it is an extension of a large subset of SML. This subset is obtained by excluding the imperative features of SML (references, assignment, and so-called imperative type variables) and input/output, by requiring structure declarations and functor declarations to include explicit signatures, and by adopting a more restrictive view of the role of signatures as interfaces. The first restriction is made for the sake of simplicity, and for philosophical reasons which will be familiar to advocates of functional programming [3]. The second restriction seems appropriate in a specification and formal development framework in which signatures play a central role, in contrast to a programming language where the need to supply explicit signatures may be viewed as an unnecessary inconvenience. The only structure declarations that are exempt from this restriction are those in which the signature is already available from the structure used in the body

of the declaration, as in the case of the structure declaration in the body of `Reduce` in Figure 1. The final restriction is to enforce the principle that only the information which is explicitly recorded in the signature(s) of a module is available externally, as mentioned in Section 2.1. This is necessary since the SML module system does not otherwise fully insulate the clients of a module from choices in the representation of types in the body, and therefore does not properly support separate development of the components of a modular system. See [34] for more on the methodological technicalities behind this restriction.[4] None of these changes makes EML incompatible with SML, as any program in the SML fragment of EML (which therefore satisfies these restrictions) is a well-formed SML program. However, certain SML programs cannot be developed using EML.

There is one additional restriction imposed by EML which causes certain pathological but well-formed SML programs to be regarded as incorrect. This is demonstrated by the following example:

```
signature SIG =
    sig
        type t
        local val x:t in end
    end;

structure S:SIG =
    struct
        datatype t = foo of t
    end
```

This is well-formed according to SML but fails to *verificate* in EML because `S.t` is a type with no values! (Recall that values in SML are represented as finite ground terms built from constructors; since the only constructor for type `S.t` is `S.foo:S.t->S.t`, there are no finite ground terms of type `S.t`.) The point here is that `local val x:t in end` in `SIG` imposes a logical constraint, namely that `t` has at least one value, which is disregarded by SML but cannot be correctly disregarded by EML. Apart from this minor restriction and the restrictions mentioned above, EML does not limit the freedom of the SML programmer in the sense that well-formed SML programs satisfying these restrictions (even "ugly" ones) are also well-formed according to EML. Of course, it is clear that it will be easier to reason about the correctness of some SML programs than others, in EML or any other framework.

Compatibility between SML and EML is a more delicate matter than simply insuring compatibility for the SML fragment of EML. For example, the dynamic semantics of EML (see Section 4.2), which defines the result of evaluating EML "code" insofar as this is possible, raises the exception `NoCode` when producing a result would involve evaluating a specification construct such as a quantifier or question mark. Refinement steps involve the replacement of question marks by expressions. This would lead one to expect that successive refinement steps should cause the dynamic semantics to raise `NoCode` exceptions less frequently,

[4] The original design of the SML module system [20] proposed an additional kind of structure, a so-called *abstraction*, for which the stricter interpretation of signatures taken in EML would apply. This was unfortunately not included in SML as defined in [22] although some SML implementations provide it as a non-standard extension [1].

without affecting the "ordinary" values produced. In order to achieve this, it is essential to define `NoCode` as a special exception which cannot be trapped by any surrounding handler. Consider the following (contrived) refinement example:

```
val x = (? handle _ => 2)  ~~>  val x = (1 handle _ => 2)
```

In SML, *exp* `handle _ => 2` evaluates to `2` if *exp* raises any exception. If this were the case in EML, then the above refinement would change the result of evaluating `x` from `2` to `1`. By treating `NoCode` as a special non-trappable exception (which involves a change to the dynamic semantics of the SML fragment of EML!) the result changes from `[NoCode]` to `1`, as desired.

By way of disclaimer, it should be noted that the assertions above concerning such matters as compatibility between the semantics of SML and EML should be formally regarded as conjectures which we strongly believe to be true but which have not yet been formally proved; the same goes for similar assertions in the remainder of the paper.

4 An Overview of the EML semantics

As mentioned earlier, one of the most important features of SML is that it has a fully formal definition (modulo some minor faults [13]). Not only is its syntax formally defined, which is quite common, but also the meaning of SML programs is determined unambiguously by a formal mathematical semantics [22], [21]. This is given in the form of so-called *natural semantics* [12] (or structural operational semantics [25]) via deduction rules which determine a meaning for each SML phrase. We will present a number of such rules below, hopefully giving the reader the flavour of the entire semantics.

The semantics of SML consists of some two hundred rules, grouped to reflect both the structure of the language and the envisaged phases of program interpretation. Thus, on one hand, the semantics of SML divides into the semantics for the core layer of the language and the semantics for its module system. Then, the semantics for the core and the semantics for modules are split into two parts: the *static semantics*, which describes the type-checking phase of program interpretation, and the *dynamic semantics*, which describes the actual evaluation of programs. In addition, the *derived forms* of the language are described by translation to phrases of the *bare language*.

The dependencies between various parts of the semantics are kept to a minimum, to facilitate understanding of the quite complex language definition. As expected, the static semantics for modules relies on the static semantics for the core. Similarly for the dynamic semantics: the dynamic semantics for modules relies on the dynamic semantics for the core. However, no part of the semantics for the core depends on the semantics for modules, and the static semantics and the dynamic semantics are independent[5]. All the parts are joined at the top level, where the overall semantics for SML *programs* involves both type-checking (the static semantics) and evaluation (the dynamic semantics).

[5] Although this statement is technically accurate, a successful "run" of the static semantics is needed to ensure that the dynamic semantics yields expected meanings. In this sense the dynamic semantics depends on the static semantics. A precise statement of this "soundness" property may be found in [38].

The semantics of EML inherits its basic form and structure from the semantics of SML. It is given as a natural semantics and consists of a number of deduction rules grouped to reflect the structure of the language and the various aspects of the interpretation of EML phrases. As in the SML semantics, the semantics for EML core and modules are given separately, each of them incorporating static semantics and dynamic semantics. The meaning of the derived forms of EML is given by translation to the bare language — the description of this translation is considerably more detailed than the corresponding part of the SML semantics, since we have decided to capture formally all the technicalities, whereas the definition of SML relies on a somewhat informal English description.

In addition we also have a *verification semantics* for EML, again split into the verification semantics for the core and for modules. In a way, the verification semantics for EML modules is the essence of Extended ML. It is here that the correctness of modules w.r.t. their interfaces is formally defined. We consider a (well-typed) EML program to be *correct* if the verification semantics produces a meaning for it. If the verification semantics *fails* for this program, that is, no verification meaning for the program may be derived, the program is considered incorrect. Incorrect programs may still be "run" (according to their dynamic semantics) — but the results are not guaranteed to meet the requirements expressed in the module interfaces.

The dependencies between the various parts of the EML semantics are somewhat more complicated than in SML. As in SML, the semantics for modules depends on the semantics for the core, while the semantics for the core does not depend on the semantics for modules. The static semantics and the dynamic semantics are independent. However, the new part of the semantics, the verification semantics, depends on both the static and the dynamic semantics. As explained in Section 2.4, the interpretation of axioms depends on typing information (for example, the type of the bound variable must be known to interpret the meaning of a universally quantified formula) — hence the dependency on the static semantics. The dependency on the dynamic semantics results from the need to interpret axioms describing evaluation properties of expressions (for example, stating that an expression terminates). We should hasten to add that neither the static nor the dynamic semantics depends on the verification semantics, as should be expected. Finally, as for SML, all the parts of the semantics are joined at the top level, where the overall semantics of EML "programs" is given.

In the rest of this section we present fundamental ideas important for each part of the semantics — see [14] for the complete semantics. We skim through the static and the dynamic semantics, as the issues involved there are much the same as in the semantics of SML — we hope, however, to give the flavour of these parts. More attention is paid to the verification semantics, as this is the really new (and most interesting) part of EML.

4.1 Static semantics

The static semantics of EML describes the process of static *elaboration* of EML phrases. This includes, for example, checking that all the objects used have been declared in the current environment and, most significantly, that phrases are *well-typed*.

Perhaps most typically, the rules of the static semantics for expressions allow one to derive judgements of the form $C \vdash \mathit{exp} \Rightarrow \tau$. This is to be read: in the context C, the expression *exp* can elaborate to the type τ (or *exp* can have type τ). Here, contexts are triples, where the most essential component is a *static environment* storing typing information about the objects declared in the current environment. We have $C \vdash$ `[1]` $\Rightarrow$ `int list` and $C \vdash$ `[]` $\Rightarrow$ `int list` (for any[6] context C). Note, however, that we also have $C \vdash$ `[]` $\Rightarrow \alpha$ `list`, where α `list` is the type of lists over arbitrary type α. The *polymorphic* generalisation of this type is written as $\forall\alpha.\,\alpha$ `list`. It is formed when an expression of type α `list` is bound to an identifier (provided α is not fixed by the context). $\forall\alpha.\,\alpha$ `list` may be instantiated to any type of the form τ `list`.

Declarations are slightly more complicated: the static semantics elaborates a declaration to a static environment, containing typing information about the objects introduced by the declaration. The corresponding judgements are of the form $C \vdash \mathit{dec} \Rightarrow E$, and for example we have $C \vdash$ `val a = 5` $\Rightarrow \{$`a` $\mapsto$ `int`$\}$. Examples involving function declarations are no more complicated: we have $C \vdash$ `val f = fn x => [x]` $\Rightarrow \{$`f` $\mapsto$ `int` $\to$ `int list`$\}$, as well as $C \vdash$ `val f = fn x => [x]` $\Rightarrow \{$`f` $\mapsto \forall\alpha.\,\alpha \to \alpha$ `list`$\}$.

The judgements mentioned above may be formally derived using the rules of the static semantics. A typical example of such a rule, involving the elaboration of both declarations and expressions, is the following rule for expressions with local declarations (this is a simplified version of the rule!):

$$\frac{C \vdash \mathit{dec} \Rightarrow E \qquad C \oplus E \vdash \mathit{exp} \Rightarrow \tau}{C \vdash \texttt{let}\ \mathit{dec}\ \texttt{in}\ \mathit{exp}\ \texttt{end} \Rightarrow \tau}$$

This is to be read: if in the context C the declaration *dec* elaborates to the static environment E and in the context C extended by the static environment E the expression *exp* elaborates to the type τ, then in the context C the expression `let` *dec* `in` *exp* `end` elaborates to the type τ. Notice that the result of the elaboration of *dec* does not appear in the overall result. For example, using this rule we can derive $C \vdash$ `let val f = fn x => [x] in f 5 end` $\Rightarrow$ `int list` (for any context C).

The static semantics for modules proceeds in much the same way as that for the core, but the semantic values built are more complex. For example, structure expressions elaborate to static environments, which store typing information about the objects declared within the structure, together with a unique name attached to the structure to keep track of sharing. The corresponding judgements have the form $B \vdash \mathit{strexp} \Rightarrow (m, E)$, where B is a *static basis*, containing a context and a set N of *structure names*, like m, used so far. Here is a typical rule, for the encapsulation of a structure-level declaration of objects to form a new structure:

$$\frac{B \vdash \mathit{strdec} \Rightarrow E \qquad m \notin (N \text{ of } B) \cup \text{names}\, E}{B \vdash \texttt{struct}\ \mathit{strdec}\ \texttt{end} \Rightarrow (m, E)}$$

The hints above on the static semantics apply to SML as well as to EML. However, as mentioned before, there are some differences. For example (cf.

[6] We tacitly assume that contexts, environments, etc., used in the small running examples throughout this section map the built-in type constructors and values of EML to their expected meanings, as described in the initial basis for SML, cf. [22].

Section 3) we have designed typing for EML modules to be stricter than for SML, and this change is properly reflected by the static semantics for EML modules. Let us consider a simple structure declaration:

```
structure S: sig type t; val c:t end =
    struct type t = int; val c = 17 end
```

In SML, the signature constraint in this particular example has *no effect*: the static environment assigned to the structure identifier `S` maps `t` and `c` to `int`. A signature constraint in SML, if present, is used only to check that the structure matches the signature and to hide auxiliary structure components. In EML, signature constraints have an additional purpose: they also hide information about structure components — only the information provided in the signature can be exploited when using the structure. In particular, in the above example, the EML static semantics binds `S` to a static environment which maps `t` and `c` to a new, otherwise unknown type. Consequently, in the context of the above structure binding, in EML we cannot form expressions like `S.c+2` — this is not well-typed in EML (but it is well-typed in SML). This behaviour of EML is compatible with SML in the sense that every successful elaboration in EML will also succeed in SML.

Another difference is that in EML we have a new part of the semantics, the verification semantics, which relies on the type information gathered during static elaboration. We need some mechanism to export this information from the static to the verification semantics of EML, also covering cases in which the intermediate types for some parts of EML phrases do not appear in the overall result, as for example the type of `f` in the elaboration of `let val f = fn x => [x] in f 5 end`, which we considered earlier. This is done by requiring that all the types used in a static elaboration of a phrase are accumulated in an additional component of the result of elaboration: a tree of *type guesses*. One can think of this as an annotation of the entire parse tree for the phrase with results of the static analysis. The presence of type guesses somewhat complicates both the form of judgements and the rules of the static semantics. For instance, the above rule for expressions with local declarations in fact looks as follows[7]:

$$\frac{C \vdash \mathit{dec} \Rightarrow E, \gamma \qquad C \oplus E \vdash \mathit{exp} \Rightarrow \tau, \gamma' \qquad \text{tynames}\,\tau \subseteq T \text{ of } C}{C \vdash \texttt{let}\ \mathit{dec}\ \texttt{in}\ \mathit{exp}\ \texttt{end} \Rightarrow \tau, \gamma \cdot \gamma'}$$

Here, the tree of type guesses γ accumulates the types used in the elaboration of *dec* to the static environment E in the context C, γ' accumulates the types used in the elaboration of *exp* to the type τ in the context $C \oplus E$, and consequently $\gamma \cdot \gamma'$ accumulates the types used in the elaboration of `let` *dec* `in` *exp* `end` to the type τ in the context C.

An additional problem is that the static semantics may "choose" different types for some parts of a phrase without affecting the overall result (the differences would be visible in the tree of intermediate type guesses though).

[7] The third premise, which requires that the type of *exp* does not use any new type names not mentioned in the original context, is not present in the corresponding rule of the SML definition. The type system is unsound without this requirement, because type names introduced by different `let` expressions can accidentally become equal. This was an oversight in the definition of SML [22] which was not fixed in [21].

As mentioned above, the type of `fn x => [x]` may be either $\texttt{int} \to \texttt{int list}$ or $\alpha \to \alpha\ \texttt{list}$ (among others). Moreover, since `f 5` elaborates to $\texttt{int list}$ both in the context assigning $\texttt{int} \to \texttt{int list}$ to `f` and in the context assigning $\forall\alpha.\, \alpha \to \alpha\ \texttt{list}$ to `f`, the elaboration of `let val f = fn x => [x] in f 5 end` may proceed either via the judgement $C \vdash \texttt{val f = fn x => [x]} \Rightarrow \{\texttt{f} \mapsto \texttt{int} \to \texttt{int list}\}$, or via $C \vdash \texttt{val f = fn x => [x]} \Rightarrow \{\texttt{f} \mapsto \forall\alpha.\, \alpha \to \alpha\ \texttt{list}\}$, in each case yielding $C \vdash \texttt{let val f = fn x => [x] in f 5 end} \Rightarrow \texttt{int list}$, but with different intermediate type guesses. The type chosen for `f` may influence the result of the verification semantics (well, not in this trivial case, but for example if `f` was involved in an axiom like `forall (x, y) => f x = f y`, which unexpectedly happens to be true if `f` is typed as $\texttt{unit} \to \texttt{unit list}$ — see Section 2.4). To resolve the potential ambiguity, we have to decide which of the possible types should be "exported". The obvious choice is the most general, *principal* type [6] ($\forall\alpha.\, \alpha \to \alpha\ \texttt{list}$ for `f` here), and so an appropriate principality requirement is imposed on type guesses, much as in the SML static semantics for modules the principality requirement is imposed on signatures. The existence of principal types and signatures is a fundamental property of the SML type system (see [21] for a precise statement and proof) which is retained by EML.

The requirement of principality is essentially an infinitary condition which states that *any* type that can be used in the static elaboration of a phrase is an instance of the principal type the elaboration is required to choose. In the semantics of SML it is imposed for example in the following rule:

$$\frac{C \text{ of } B \vdash dec \Rightarrow E \quad E \text{ principal for } dec \text{ in } (C \text{ of } B)}{B \vdash dec \Rightarrow E}$$

which states that if a declaration *dec* elaborates as a core declaration to a static environment E that is moreover principal for *dec* in the given context, then *dec*, as a structure-level declaration, elaborates to E (notice the crucial distinction between the elaboration of *dec* as a core declaration and as a structure-level declaration). Infinitary requirements of this kind, hidden behind somewhat informal (but precise enough) English descriptions, occur in a very few places in the semantics of SML. They are, however, rather more common in the semantics of EML; for example, they naturally arise in the semantics of quantifiers or extensional equality, see Section 2.4. We have decided to make such requirements explicit and formalise the use of infinitary conditions via *higher-order rules*. For instance, the above SML rule may be expressed as follows:

$$\frac{C \text{ of } B \vdash dec \Rightarrow E \qquad \dfrac{C \text{ of } B \vdash dec \Rightarrow E'}{E \succ E'}}{B \vdash dec \Rightarrow E}$$

Here, the second premise is a rule, which is true as a premise if it is admissible as a rule. The meta-variable E' is scoped at this premise, making it universally quantified for the local rule. Thus, the premise requires each E' to which *dec* may elaborate to be an instance of E. Consequently, the new rule means exactly the same as its original version quoted above from the semantics of SML.

Actually, the semantics of EML uses here yet a different rule, which imposes the principality requirement not just on the resulting static environment, but

on the entire elaboration as accumulated in the tree of type guesses:

$$\frac{C \text{ of } B \vdash dec \Rightarrow E, \gamma \qquad N = \text{names}\,\gamma \setminus N \text{ of } B \qquad \frac{C \text{ of } B \vdash dec \Rightarrow E', \gamma'}{(N)\gamma \succ \gamma'}}{B \vdash dec \Rightarrow E, \gamma}$$

The last premise of this rule requires that any tree of type guesses corresponding to an elaboration of *dec* in the given context may be obtained from the tree of type guesses γ by instantiating new type variables introduced in the corresponding elaboration of *dec*. As explained above, this requirement, stronger than just principality of the resulting environment, is necessary for the semantics of EML.

Higher-order rules, which come with an additional scoping mechanism for meta-variables, considerably increase the expressive power of the formalism. They have to be used with care, as the formalism no longer guarantees that the rules inductively define all the true judgement of the semantics. In particular, "impredicative" dependencies between premises and conclusions in higher-order rules must be avoided.

4.2 Dynamic semantics

The dynamic semantics of SML, as for any other programming language, is the key part of its description. After all, the main reason for writing programs is in order to evaluate them, and this is what the dynamic semantics describes. One might think, however, that a dynamic semantics for a program development framework like EML is somewhat pointless: the dynamic semantics for the programs produced by formal development is provided by the definition of SML, and can be used to evaluate them. One reason to nevertheless provide a separate dynamic semantics for EML is that the verification semantics, the main part of the EML semantics, relies on the dynamic semantics, for example to determine the value of the `terminates` predicate — hence, the dynamic semantics is needed here for the sake of completeness of the formal definition of EML. Another important reason is that we want to formally define a basis for early practical experiments with incomplete programs. EML programs, even those which are incomplete and contain specification constructs, are viewed as "partially executable". The idea is that any such program should be executable insofar as this is possible, and that evaluation should proceed as in SML for the parts which contain only SML code. The dynamic semantics of EML formalises this.

The dynamic semantics describes the *evaluation* of language phrases. In particular, for expressions, the dynamic semantics allows one to derive judgements of the form[8] $E \vdash exp \Rightarrow v$, stating that in the (dynamic) *environment* E, the expression *exp* evaluates[9] to the value v, where environments store the values of objects currently defined. For example, we have $\{\texttt{a} \mapsto 27\} \vdash \texttt{a*37} \Rightarrow 999$. Environments are built by declarations, with corresponding judgements of the form $E \vdash dec \Rightarrow E'$ expressing the fact that in the environment E the declaration *dec* evaluates to the environment E', which stores the values of objects

[8] This is an approximation used here for presentation purposes only; more details will be provided below.

[9] $E \vdash exp \Rightarrow v$ literally means that in E, *exp* can evaluate to v, but since evaluation is deterministic, v is uniquely determined (if it exists).

declared in *dec*. For instance, we have $E \vdash$ `val a = 27` $\Rightarrow \{\mathtt{a} \mapsto 27\}$ (for any environment E). Formally, judgements are derived using the rules of the dynamic semantics, with a typical example being the following rule for expressions with local declarations:

$$\frac{E \vdash dec \Rightarrow E' \qquad E + E' \vdash exp \Rightarrow v}{E \vdash \mathtt{let}\ dec\ \mathtt{in}\ exp\ \mathtt{end} \Rightarrow v}$$

Using this rule, we can for example derive directly from the judgements above that $E \vdash$ `let val a = 27 in a * 37 end` $\Rightarrow 999$.

Evaluation of expressions involving functions is just as simple. One has to remember though that values of function types are not functions in the usual sense but rather *closures*, which result from the encapsulation of expressions defining function bodies [18]. Closures are expanded when applied to arguments, and a rather elaborate scheme of self-expansion is used to model recursion (see [22] for details). The possibility of non-termination is reflected by the fact that using the rules of the dynamic semantics one cannot derive values for all the expressions of the language. For example, there is no value v for which the judgement $E \vdash$ `let fun loop() = loop() in loop() end` $\Rightarrow v$ can be derived, as expected.

Another complication arises from the fact that SML (and hence EML) expressions may raise exceptions. In this case, the result of evaluation is a *packet* (an exception name possibly together with a value). Consequently, the formal judgements of the dynamic semantics for expressions may also have the form $E \vdash exp \Rightarrow p$ (in the environment E the expression *exp* evaluates to the packet p). To express the two possibilities jointly, we write $E \vdash exp \Rightarrow v/p$, and use the semantic rules to determine which form is derivable for a particular expression. The possibility of a phrase raising an exception is often left implicit in the semantic rules, relying on the so-called "exception convention" to ensure that packets are propagated by the rules of the dynamic semantics. Thus, the above rule for expressions with local declarations induces implicitly, by the exception convention, the following rule:

$$\frac{E \vdash dec \Rightarrow E' \qquad E + E' \vdash exp \Rightarrow p}{E \vdash \mathtt{let}\ dec\ \mathtt{in}\ exp\ \mathtt{end} \Rightarrow p}$$

(and similarly for packets arising from evaluation of *dec*). Of course, some semantic rules must be exempted from the exception convention. Most notably, the rules that describe how exceptions may be trapped (how packets may be handled) deal with packets explicitly.

Another aspect of dealing with exceptions is that the set of exception names used is determined dynamically — a new exception name is generated each time an exception declaration is evaluated (this new exception name is used as the meaning of the exception identifier declared). Consequently, the set of exception names generated so far must be stored. In SML this set is one of the components of the current *state* — and since its other components are used to describe the imperative features of SML programs, this is the only component of states in the dynamic semantics of EML (apart from the *specification flag*, see below). This means that states are necessary in EML, and the real form of semantic judgements describing evaluation of expressions is $s, E \vdash exp \Rightarrow v/p, s'$ (in the state s and the environment E, the expression *exp* evaluates to the value

v or packet p with the resulting state s'). The so-called "state convention" allows one to formulate many rules without mentioning states explicitly, using the order of premises to determine how states resulting from evaluation of one phrase are passed to another. Thus, in particular, the above rule for expressions with local declarations expands to the following:

$$\frac{s, E \vdash dec \Rightarrow E', s' \qquad s', E + E' \vdash exp \Rightarrow v, s''}{s, E \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Rightarrow v, s''}$$

The rules resulting from the use of the exception convention are affected similarly.

The above remarks apply to SML as well as to EML — the overall ideas on how programs are evaluated are the same. What is new in EML is that it contains some phrases which, intuitively, cannot be evaluated. Typical examples here are objects defined by declarations where no code is provided (the lack of code being represented by `?`) or phrases containing constructs for building formulae, such as `==`, `terminates`, or `forall`. Even though the dynamic semantics of EML simply skips axioms, these non-executable constructs may be encountered in evaluation of EML expressions. When this is the case, a special exception `NoCode` is raised. `NoCode` cannot be handled explicitly in programs, as explained in Section 3. However, to enable execution of completed parts of EML programs, `NoCode` is trapped by the dynamic semantics of EML at the declaration level and a special value `Incomplete` is used to mark its presence in the evaluation of an object declaration. An attempt to use the value `Incomplete` causes `NoCode` to be raised again. Here are a few examples:

$E \vdash$ `(fn x : int => x - 1) == (fn x : int => x + 1)` $\Rightarrow$ [`NoCode`]
$E \vdash$ `val x : int = ?` $\Rightarrow \{$`x` $\mapsto$ `Incomplete`$\}$
$\{$`x` $\mapsto$ `Incomplete`$\} \vdash$ `x + 27` $\Rightarrow$ [`NoCode`]
$\{$`x` $\mapsto$ `Incomplete`, `y` $\mapsto$ `Incomplete`$\} \vdash$ `27 * 3` $\Rightarrow 81$
$E \vdash$ `let val x : int = ?; val y = x + 1; val a = 27 in a * 3 end` $\Rightarrow 81$

This yields a rather subtle difference between the dynamic semantics of EML and both the dynamic semantics of SML (which simply does not deal with these special new constructs of EML) and the verification semantics of EML (where, in a sense, these constructs are properly dealt with). To make this explicit, we have added to EML states a new component, a *specification flag*, which is raised by the dynamic semantics whenever one of these special new constructs is encountered. When the specification flag is not raised during the evaluation of a phrase, the results provided by the dynamic semantics of EML coincide with the results of the dynamic semantics of SML[10] as well as with the results of the verification semantics for the core of EML (see Section 4.3 below). However, when the specification flag is raised, then the dynamic semantics of SML cannot yield a result, and the verification semantics of EML may yield a different result (or fail to yield a result at all).

The dynamic semantics for EML modules follows the dynamic semantics for SML modules in the same manner as the dynamic semantics for the EML core sketched above follows the dynamic semantics for the SML core. Thus, in

[10] Somewhat informally, we mean here the semantics of SML literally applied to EML phrases, hence in particular with no rules applicable for the special new constructs of EML.

particular, EML structure expressions evaluate to environments, but evaluation need not terminate and may modify the state. Moreover, evaluation proceeds in a *basis*, a "richer" environment which apart from the values of objects stored as in the dynamic environment for the core may also store functors and signatures. The corresponding judgements have the form $s, B \vdash \mathit{strexp} \Rightarrow E, s'$. The specific EML constructs are treated as sketched above: axioms are disregarded, evaluation of non-executable expressions raises the `NoCode` exception and may result in the value `Incomplete` being stored in the environment. In particular, environments resulting from evaluation of EML structures may contain objects with `Incomplete` stored as their value.

4.3 Verification semantics

Although we provide a dynamic semantics for EML, the main stress in a framework like EML is not so much on running programs (their dynamic evaluation) but rather on the verification of correctness assertions that are present in EML phrases. Consequently, we view the verification semantics as the main part of the formal description of EML. The essence of this semantics is to check whether structures and functors *match* their signatures, which in particular means that they satisfy the axioms given in the signatures. This is described by the verification semantics for EML modules. Verification of an EML phrase does not result in a binary statement saying whether the phrase is correct or not. Some more detailed information about the contribution of the phrase to the meaning of the whole program must be determined as well. We will say that the verification semantics describes how EML phrases *verificate*[11] to semantic objects.

One crucial idea of the EML methodology is that not only should developed modules be correct w.r.t. their specifications, but also this should follow solely from properties stated in module interfaces. Consequently, the verification semantics must express the information hiding implicit in this EML understanding of the role of module interfaces. Incompleteness of information is represented by the fact that EML module phrases verificate to sets of semantic objects, rather than just to a single semantic object as in the dynamic semantics. For instance, in a given basis, EML structure expressions verificate to sets of environments[12], with the corresponding formal judgements having the form $B \vdash \mathit{strexp} \Rightarrow \mathcal{E}$. Typically, in a complete EML structure expression (containing only SML code) without substructures, the resulting set of environments will contain exactly one element: the environment determined by the SML code. But there are several reasons why this set might not be a singleton. Most obviously, there may be unresolved choices within *strexp*. For example, a structure-level declaration like `val a : int = ?` results in a set of environments, each mapping `a` to a different integer. Then, inconsistency within *strexp* may cause the resulting set to be empty. For example, an axiom like `axiom a>5 andalso a<3` results in the

[11] An obvious alternative is "verify", but this carries connotations we would like to avoid.

[12] To be quite precise, we should point out that just as in the dynamic semantics of EML it was necessary to consider environments together with state s, in the verification semantics of EML structure expressions verificate to sets of elements that are pairs of an environment and a state. Fortunately, this does not bring much additional complication, and for the purposes of the presentation here we disregard states in the further discussion of the verification semantics.

empty set of environments. Notice, however, that this is different from a failure to verificate at all! Finally, and perhaps most crucially for the methodological aspects of the verification of EML programs, if *strexp* contains a substructure or uses another structure then the interface attached to it is used to filter the information available, hiding the more detailed information given in its body. Consequently, the "verification meaning" of the structure is the set of environments matching its interface, rather than the particular environment given by its body.

This last point is perhaps best explained by looking at the verification of a single structure declaration `structure S :` *sigexp* `=` *strexp*. To verificate this, one proceeds as follows (we leave the basis in which the verification takes place implicit):

1. First, verificate the signature expression *sigexp*, obtaining a *verification interface* Σ. This stores the names of objects specified in the signature together with static information about them. Moreover, axioms given in the signature are stored in an appropriate form — see below for more details.

2. Then, verificate the structure expression *strexp*, obtaining a set of environments $\mathcal{E}$ as discussed above.

3. The next step is where the real verification takes place: check that each environment $E \in \mathcal{E}$ matches the interface Σ. This involves checking whether the axioms incorporated in Σ are satisfied by E. Section 2.4 presents the particular forms of axioms and their intended meaning, which we return to below.

4. The result is the set of all environments binding `S` to an environment that matches the interface Σ. Notice that this "includes" but is in general larger than the set $\mathcal{E}$ of environments obtained from the verification of *strexp*.

If any of the above steps fails (this may happen in step 2, for example if *strexp* contains an incorrect substructure declaration, or in step 3, if the verification requirement formulated there does not hold) then the structure declaration `structure S :` *sigexp* `=` *strexp* is incorrect and hence its verification fails as well. This is different, however, from the case in which the result is the empty set. The latter is possible when *sigexp* is inconsistent, and hence *strexp* (which satisfies it) is inconsistent as well.

Here is (a simplified version of) the rule which embodies the above verification procedure:

$$\frac{B \vdash \mathit{sigexp} \Rightarrow \Sigma \qquad B \vdash \mathit{strexp} \Rightarrow \mathcal{E} \qquad \text{for each } E \in \mathcal{E},\ E \text{ matches } \Sigma}{B \vdash \texttt{structure S :}\ \mathit{sigexp}\ \texttt{=}\ \mathit{strexp} \Rightarrow \{\, \{S \mapsto E'\} \mid E' \text{ matches } \Sigma\}}$$

A few comments are necessary here. First, we omit a formal definition of the condition stating that an environment matches an interface. Second, for the presentation here we have used an *ad hoc* (but self-explanatory) notation to present a rule with an infinite set of premises, where moreover the number of these depends on a semantic object mentioned in another premise. The formal semantics uses a higher-order rule to express this more precisely. Finally, this is

a very simplified version of a rule that does not actually appear in the semantics, but may be derived using more elementary rules for structure bindings and for structure declarations.

To take a simple example, consider the following structure declaration:

```
structure S: sig val a: int; axiom a>0 andalso a<5 end =
    struct val a: int = ?; axiom a>1 andalso a<4 end
```

The verification of the structure expression in this declaration results in the set of environments $\{E_2, E_3\}$ where we write E_i for $\{\mathtt{a} \mapsto i\}$. It is then checked that each of these environments does indeed match the interface, and in particular satisfies the axiom given there. The resulting set of environments assigning an interpretation for the structure `S` contains not only $\{\mathtt{S} \mapsto E_2\}$ and $\{\mathtt{S} \mapsto E_3\}$, but also $\{\mathtt{S} \mapsto E_1\}$ and $\{\mathtt{S} \mapsto E_4\}$, since the set of environments matching the interface is exactly $\{E_1, E_2, E_3, E_4\}$.

If we modify the interface as follows:

```
structure S: sig val a: int; axiom a>0 andlalso a<3 end =
    struct val a: int = ?; axiom a>1 andalso a<4 end
```

then the check that each of the environments resulting from the verification of the structure expression (E_2 and E_3) matches the interface fails (since E_3 does not satisfy the modified axiom). Thus, the verification of this structure declaration fails. Intuitively, the structure declaration is incorrect.

Summing up, the outcome of a successful verification of a structure-level declaration is a set of environments, each expressing a possible meaning of the declared objects. Further verification proceeds for each of these possibilities separately, as expressed by the following rule for sequential composition of structure-level declarations (again, a very simplified version is used, with an *ad hoc* notation to represent dependencies between objects):

$$\frac{B \vdash \mathit{strdec}_1 \Rightarrow \mathcal{E}_1 \qquad \text{for each } E \in \mathcal{E}_1,\ B \oplus E \vdash \mathit{strdec}_2 \Rightarrow \mathcal{E}_2[E]}{B \vdash \mathit{strdec}_1\,;\mathit{strdec}_2 \Rightarrow \{E_1 + E_2 \mid E_1 \in \mathcal{E}_1, E_2 \in \mathcal{E}_2[E_1]\}}$$

The above rule appropriately respects the dependencies between consecutive structure declarations. Consider the following example:

```
structure S: sig val a: bool end =
    struct val a: bool = ? end;
structure T: sig val b: bool; axiom b = S.a end =
    struct val b: bool = S.a end
```

The verification of these two declarations will result in the set of environments containing $\{\mathtt{S} \mapsto S_t, \mathtt{T} \mapsto T_t\}$ and $\{\mathtt{S} \mapsto S_f, \mathtt{T} \mapsto T_f\}$, where $S_t = \{\mathtt{a} \mapsto \mathtt{true}\}$, $T_t = \{\mathtt{b} \mapsto \mathtt{true}\}$, $S_f = \{\mathtt{a} \mapsto \mathtt{false}\}$ and $T_f = \{\mathtt{b} \mapsto \mathtt{false}\}$. However, the resulting set of environments does not contain for example $\{\mathtt{S} \mapsto S_t, \mathtt{T} \mapsto T_f\}$ even though the interface for `S` does not determine the value of `a` (nor does the structure body in this case). The point is that the verification of the declaration of `T` proceeds in the context of an arbitrary but fixed interpretation for `S.a`, for each of the open possibilities separately.

On the other hand, removing the explicit information about the dependency from the interface for `T` changes the result:

```
structure S: sig val a: bool end =
    struct val a: bool = ? end;
structure T': sig val b: bool end =
    struct val b: bool = S.a end
```

Now, the result of the verification of these two declarations will consist of four environments: $\{\texttt{S} \mapsto S_t, \texttt{T'} \mapsto T_t\}$ and $\{\texttt{S} \mapsto S_f, \texttt{T'} \mapsto T_f\}$ as before, but also $\{\texttt{S} \mapsto S_t, \texttt{T'} \mapsto T_f\}$ and $\{\texttt{S} \mapsto S_f, \texttt{T'} \mapsto T_t\}$. Even though the verification of the structure expression in the declaration of `T'` results in the set of only two environments (as before), this information is filtered out by the interface provided in the binding. Consequently, a further declaration

```
structure U: sig val c: bool; axiom c = S.a end =
    struct val c: bool = T'.b end
```

is incorrect and does not verificate.

All the small examples above were extremely simple and an intuitive understanding of EML axioms as presented in Section 2.4 was sufficient to interpret them. In general, however, the situation may be much more complex, and matching an EML structure against an EML signature involves a number of rather subtle points. Perhaps the most obvious is the fact that the axioms in the signature must be interpreted relative to the type instantiation determined by the structure. For example, in

```
signature A = sig type t
                  axiom exists x:t => true
              end
```

the axiom requires the type `t` to be non-empty and its satisfaction depends on the particular realisation of `t` in the structure we match against `A`.

Another important point is that signatures in both SML and EML allow the use of hidden functions and hidden types. For the dynamic semantics hidden objects are of no concern, but they do matter in the verification semantics, because their interpretation may influence the verification of axioms. For example, a structure matching the following signature

```
signature B = sig local val b: int
                        axiom b>0
                  in val c: int
                     axiom c>b+1
                  end
              end
```

need not include a value `b` (but has to include an integer value `c`, of course). However, to successfully verificate the axiom `c>b+1`, such a value `b` has to be guessed so that both the "hidden" axiom `b>0` and then the "visible" axiom `c>b+1` are satisfied (in this example, this would not be possible unless the value of `c` is greater than 2). In a certain sense, the hidden declarations are existentially quantified (see [7]). To take appropriate care of such cases the axioms in verification interfaces are stored in a rather more elaborate form of *generalised axioms*.

The above presentation of the verification of structure declarations extends to the verification of functor declarations in the obvious way.

In this sketch of the verification semantics for EML modules we have entirely omitted the issue of behavioural equivalence mentioned in Section 2.2. Unfortunately, we have not yet put the relevant technicalities into the current version of the semantics. However, we do not anticipate major problems with this. First, a concept of behavioural equivalence between EML structures (environments) will be needed. In any basis, this will be defined to require that any well-formed expression (possibly built in the context of an additional declaration of a local structure) of observable type has the same value in behaviourally equivalent structures. The appropriate set of observable types to choose seems to be the set of all equality types (for the verification of functor bodies, the types in parameter interfaces, which may be instantiated by equality types, should also be treated as observable). Then, the only further change in the verification semantics for structure declarations will be to replace the requirement that all environments resulting from the verification of a structure expression match the structure interface by the requirement that each of these environments is behaviourally equivalent to an environment which matches the interface.

The verification semantics for the EML core is quite similar to its dynamic semantics. The basic ideas are the same, and for example expressions verificate to values or to packets (since exceptions may be raised), possibly changing the current state. This is captured by judgements of the form $s, M \vdash exp \Rightarrow v/p, s'$, where M is a *model*, a richer context in which the EML core phrases are verificated. Similarly for declarations, where judgements have the form $s, M \vdash dec \Rightarrow E, s$. In contrast to the verification semantics for modules, the verification of EML core phrases yields single objects, as in the dynamic semantics. There are, however, some crucial differences.

First, the specification constructs added in EML, such as `==`, `terminates`, `forall`, are now viewed as special operators with their own verification rules (recall that an attempt to evaluate them in the dynamic semantics simply raises `NoCode`, a special exception reserved for this purpose). The rules of the verification semantics capture the meaning of these constructs as sketched in Section 2.4. It is important to realise that in most cases verification of these constructs depends in an essential way on static information inherited from the static semantics and incorporated in models.

Then, in contrast to the dynamic semantics, axioms are not ignored. When the verification semantics encounters an axiom declaration, it attempts to verificate the axiom body and proceeds further only if the result obtained is the value `true`. Otherwise, verification fails. This does not necessarily mean that the structure declaration in which this axiom occurs is incorrect. Rather, it implies only that a particular choice of resolving all the open possibilities in the structure body, the choice currently under consideration by the verification semantics, is not successful and does not yield a realisation of the structure satisfying this axiom. The crucial point which makes this work is the interpretation of question marks. In the verification semantics for the EML core the interpretation of question marks is provided by an extra component of the model. These question mark interpretations are guessed in an arbitrary way by the verification semantics for modules at the point where a core declaration is viewed as a structure-level declaration. Only those environments resulting from a successful verification of the declaration for some guess of the interpretation of question marks contribute to the result of the verification of this declaration at

the structure level. This is captured by the verification rule given below, again in a somewhat simplified form. Rather informally, we write $M[B, QI]$ for the model obtained by extracting the appropriate components of the verification basis B and adding the question mark interpretation QI.

$$\frac{}{B \vdash dec \Rightarrow \{E \mid \text{for some } QI, M[B, QI] \vdash dec \Rightarrow E\}}$$

As in the static semantics (see the rule imposing principality discussed in Section 4.1) the declaration *dec* is viewed here as a core declaration in the judgement $M[B, QI] \vdash dec \Rightarrow E$, and as a structure-level declaration in $B \vdash dec \Rightarrow \{E \mid \ldots\}$.

Here is a simple example of a structure expression:

```
struct
    val a: int = ?
    axiom a>5 andalso a<8
    val b = a+2
end
```

(The question mark in the declaration of `a` should perhaps be indexed to avoid potential confusion with other question marks elsewhere.) The verification semantics for the declaration enclosed in `struct ... end` tries to verificate its enclosed sequence of core declarations for each possible interpretation of the question mark, one interpretation $\{\texttt{?} \mapsto i\}$ for each integer i. It is clear that the verification succeeds only for the interpretations $\{\texttt{?} \mapsto 6\}$ and $\{\texttt{?} \mapsto 7\}$, yielding environments $E_6 = \{\texttt{a} \mapsto 6, \texttt{b} \mapsto 8\}$ and $E_7 = \{\texttt{a} \mapsto 7, \texttt{b} \mapsto 9\}$ respectively. The result of the verification of the declaration is thus $\{E_6, E_7\}$, and this set of environments is taken as the result of verification of the entire structure expression.

In the same way as our quantification is based on expressible values (see Section 2.4) question marks interpretations QI map question marks to expressions, not to values. In this way ill-formed values are avoided, and moreover, the interpretation of each question mark may depend on the context in which it appears. The latter point means that in the verification of a function declaration like

```
fun f x = let val c = ? in g c end
```

question mark interpretations may replace the `?` by expressions containing free occurrences of `x`.

The treatment of question marks in type expressions is somewhat different. The static semantics guarantees that whatever replacement a question mark interpretation provides (preserving certain attributes), the *success* of static analysis, and hence well-formedness of the program, is not affected. However, the exact *results* of static analysis are affected, and this has to be taken into account in the verification semantics, by interpreting the types derived during static analysis with respect to some *realisation*. Realisations are functions on semantic objects that assign concrete types to formal type parameters.

5 Final remarks

We have tried in this paper to provide a readable exposition of the semantics of EML, a framework for formal specification and development of SML programs. We have not discussed here in any detail the methodological assumptions and theoretical underpinnings underlying the design of this framework — these have been presented elsewhere. We have also refrained from discussing merits of the design of the SML programming language.

Work on the EML semantics is nearly finished: the complete formal definition [14] is at the proof-reading stage. Because the definition is still subject to change, there is a small possibility that some of the details in the above presentation will turn out to be slightly inaccurate with respect to the final version. But we are confident that the basic principles presented in this paper are correct and stable, and accurately reflect the intentions incorporated in the design of the framework.

The problems we are wrestling with are those inherent in the enterprise of engineering a sizable completely formal definition of a realistic, practically useful formalism. All the different aspects of this formalism interact with each other, and their mutual relationship is a delicate matter which has to be handled with care and extreme attention to detail. We should perhaps quote here the example of the formal definition of SML on which we build. The original definition of SML went through three major revisions before it was finally officially published as [22]. As a result of the study of the semantics by a larger body of users, this was then followed by a number of subsequent changes included in [21]. And even now, some inaccuracies, weak points and minor mistakes in the definition are still being discovered [13]. Nevertheless, as a whole, the SML semantics is considered (certainly by us) to be an excellent example of the precise definition of a realistic programming language, with very few practical examples of formal design achieving a comparable level of accuracy and mathematical precision.

Thus, the main problems with producing the formal definition of EML are the problems of size, necessarily involving a struggle with many tedious details. We have tried to illustrate this point in the paper. This does not mean that all the issues addressed in the semantics are mathematically trivial: on the contrary, in our view some of the specific decision in the semantics, especially those related to the formal definition of the logic of axioms, are of independent interest, and deserve further separate study.

The next major step, once the semantics is finished, is to develop a sound proof theory, which would provide the user with some formal proof rules and proof tactics to verify the correctness conditions arising in the process of program development. Given the complexity of SML and hence of EML, it may be difficult to come up with appropriate proof rules. Furthermore, checking the formal soundness of these rules w.r.t. the semantics given will be a formidable task on its own.

Defining the formal semantics of a framework like EML, or indeed of a programming language like SML, is not a futile exercise. Most obviously, it provides a common unambiguous reference for all the users of the formalism. Perhaps even more importantly, such a semantics constitutes a basis for all further work on the framework: sound development methodologies, proof techniques, support tools (including the compiler for the programming language) must all be

based on and checked against precise semantics if they are to be trustworthy in practical applications. Defining the formal semantics of a language involves taking a very close look at all the details of the language and of the complex interactions between its features. Such a detailed examination of a language is a good way (perhaps the only way) of uncovering both major and minor problems that would otherwise escape notice.

Acknowledgements: Thanks to Fabio da Silva for early collaboration on the static and dynamic semantics of EML and to Edmund Kazmierczak and anonymous referees for helpful comments on a draft of this paper. We owe special thanks to Robin Milner, Mads Tofte and Robert Harper for their work on the semantics of SML, without which the research described here would not have been possible.

References

[1] A. Appel and D. MacQueen. Standard ML of New Jersey, version 0.93. AT&T Bell Laboratories (1993).

[2] E. Astesiano *et al.* The draft formal definition of ANSI-MIL/STD 1815A Ada. Deliverable 7 of the CEC-MAP project (1986).

[3] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. **Comm. of the Assoc. for Computing Machinery** 21(8):613–641 (1978).

[4] F.L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner. **The Munich Project CIP, Vol. 1: The Wide Spectrum Language CIP-L**. Springer LNCS 183 (1985).

[5] R. Burstall and J. Goguen. An informal introduction to specifications using Clear. In: **The Correctness Problem in Computer Science** (R. Boyer and J.S. Moore, eds.), 185–213. Academic Press (1981).

[6] L. Damas and R. Milner. Principle type schemes for functional programs. **Proc. 9th Annual ACM Symp. on Principles of Programming Languages**, 207–212 (1982).

[7] J. Farrés-Casals. Verification in ASL and Related Specification Languages. Ph.D. thesis; Report CST-92-92, Univ. of Edinburgh (1992).

[8] J.-Y. Girard, Y. Lafont and P. Taylor. **Proofs and Types.** Cambridge University Press (1989).

[9] J. Goguen and J. Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. **Proc. 9th Intl. Colloq. on Automata, Languages and Programming**, Aarhus. Springer LNCS 140, 265–281 (1982).

[10] J. Guttag and J. Horning. Report on the Larch shared language. **Science of Computer Programming** 6(2):103–134 (1986).

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

[12] G. Kahn. Natural semantics. In: **Programming of Future Generation Computers** (K. Fuchi and M. Nivat, eds.), 237–258. North-Holland (1988).

[13] S. Kahrs. Mistakes and ambiguities in the definition of Standard ML. Report ECS-LFCS-93-257, Univ. of Edinburgh (1993).

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

[15] E. Kazmierczak. Modularizing the specification of a small database system in Extended ML. **Formal Aspects of Computer Science** 4(1):100-142 (1992).

[16] E. Kazmierczak. Model theory for Extended ML. Draft report, Univ. of Edinburgh (1992).

[17] B. Krieg-Brückner. PROgram development by SPECification and TRAnsformation. **Technique et Science Informatiques** (1990).

[18] P. Landin. The mechanical evaluation of expressions. **Computer Journal** 6:308–320 (1964).

[19] D.C. Luckham, F.W. von Henke, B. Krieg-Brückner and O. Owe. **Anna, a Language for Annotating Ada Programs: Reference Manual**. Springer LNCS 260 (1987).

[20] D. MacQueen. Modules for Standard ML. In: Report ECS-LFCS-86-2, Univ. of Edinburgh (1986).

[21] R. Milner and M. Tofte. **Commentary on Standard ML**. MIT Press (1991).

[22] R. Milner, M. Tofte and R. Harper. **The Definition of Standard ML**. MIT Press (1990).

[23] J. Mitchell. Type systems for programming languages. In **Handbook of Theoretical Computer Science, Vol. B** (J. van Leeuwen, ed.). North Holland (1990).

[24] L. Paulson. **ML for the Working Programmer**. Cambridge Univ. Press (1991).

[25] G. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Aarhus University (1981).

[26] H. Reichel. Behavioural equivalence: a unifying concept for initial and final specification methods. **Proc. 3rd Hungarian Computer Science Conference**, 27–39 (1981).

[27] D. Sannella. Formal program development in Extended ML for the working programmer. **Proc. 3rd BCS/FACS Workshop on Refinement**, Hursley Park. Springer Workshops in Computing, 99–130 (1991).

[28] D. Sannella. Static and logical correctness conditions in formal development of modular programs. Draft report, Univ. of Edinburgh (1993).

[29] D. Sannella and F. da Silva. Case studies in Extended ML. Draft report, Univ. of Edinburgh (1993).

[30] D. Sannella, S. Sokołowski and A. Tarlecki. Toward formal development of programs from algebraic specifications: parameterisation revisited. **Acta Informatica** 29:689–736 (1992).

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

[32] D. Sannella and A. Tarlecki. On observational equivalence and algebraic specification. **Journal of Computer and System Sciences** 34:150–178 (1987).

[33] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: implementations revisited. **Acta Informatica** 25:233–281 (1988).

[34] D. Sannella and A. Tarlecki. Toward formal development of ML programs: foundations and methodology. **Proc. Joint Conf. on Theory and Practice of Software Development**, Barcelona. Springer LNCS 352, 375–389 (1989).

[35] D. Sannella and A. Tarlecki. Extended ML: past, present and future. **Proc. 7th Workshop on Specification of Abstract Data Types**, Wusterhausen. Springer LNCS 534, 297–322 (1991).

[36] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: model-theoretic foundations. **Proc. Intl. Colloq. on Automata, Languages and Programming**, Vienna. Springer LNCS 623, 656–671 (1992).

[37] O. Schoett. Data Abstraction and the Correctness of Modular Programming. Ph.D. thesis; Report CST-42-87, Univ. of Edinburgh (1987).

[38] M. Tofte. Operational Semantics and Polymorphic Type Inference. Ph.D. thesis; Report CST-52-88, Univ. of Edinburgh (1988).

[39] J. Wing, E. Rollins and A. Zaremski. Thoughts on a Larch/ML and a new application for LP. Report CMU-CS-92-135, Carnegie Mellon University (1992).