The definition of Extended ML: a gentle introduction*

Stefan Kahrs[†] Donald Sannella[‡] Andrzej Tarlecki[§]

16th October 1995

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 language, an extension of SML with axioms for describing properties of module components.

This is an overview of the formal definition of the EML language. To complement the full technical details presented elsewhere, it provides an informal explanation of the main ideas, gives the rationale for certain design decisions, and outlines some of the technical issues involved. EML is unusual in being built around a "real" programming language having a formally-defined syntax and semantics. Interesting and complex problems arise both from the nature of this relationship and from interactions between the features of the language.

*This is an essentially revised and expanded version of [KST94a], which was based on an earlier, draft version of [KST94b].

[†]Laboratory for Foundations of Computer Science, Edinburgh University, Edinburgh, Scotland; e-mail `smk@dcs.ed.ac.uk`. This research was supported by EPSRC grant GR/J07303.

[‡]Laboratory for Foundations of Computer Science, Edinburgh University Edinburgh, Scotland; e-mail `dts@dcs.ed.ac.uk`. This research was supported by the EC-funded COMPASS Basic Research working group and MeDiCiS Scientific Cooperation Network, and by EPSRC grant GR/J07303 and an EPSRC Advanced Fellowship.

[§]Institute of Informatics, Warsaw University, and Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland; e-mail `tarlecki@mimuw.edu.pl`. This research was supported by the EC-funded COMPASS Basic Research working group and MeDiCiS Scientific Cooperation Network, and by KBN grant 2 P301 007 04.

To appear in *Theoretical Computer Science*.

1 Introduction

Extended ML (EML) is a framework for the formal development of modular Standard ML (SML) software systems that 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. The complete formal definition of the EML language [KST94b] constitutes an important milestone in this programme, necessary to provide a basis for further research on foundations and tools. The length and requisite formality of the definition renders it rather difficult to penetrate. Accordingly, this paper provides an informal overview of the definition, explaining most of the main issues involved and justifying some of the choices taken.

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 that make it very well-suited to the enterprise set out above. First, it provides powerful 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 [Sch87], [ST88], [SST92], [ST92], backed up by a large body of work on algebraic specification and the theory of formal software development (see [BKLOS91] for a comprehensive presentation of the related literature). Second, the syntax and semantics of SML is formally defined [MTH90]. 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 (provided that the specification itself is given a formal meaning as well). 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. An encouraging start in this direction, using the HOL theorem prover, is described in [VG94], [MG94].

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 [LHKO87]; but this is not intended for formal development of software from specifications (although see [Kri90]), 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 [Ast86]. Similar comments apply to Larch [GH93], which has been used in connection with various programming languages having no existing formal semantics. An attempt to apply Larch to the specification of SML modules is reported in [WRZ93], but many difficult problems remain to be solved there. 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 that do not arise when considering toy programming languages or when considering specification and formal development in abstract terms.

A related 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 that are not directly relevant to the topic at hand; for more information, see the papers cited there. Section 3 briefly discusses the way in which EML relates to and extends SML. The main body of the paper is Section 4, 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 [KST94b], while touching on the ideas behind many of the most interesting and important points. Section 5 summarizes some of the decisions involved in the design of EML and concludes with 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. [Har89] or [Pau91].

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 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), by failure of pattern matching, 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 software systems 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 optional output signature describing the structure that 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 collectively are referred to as *modules*.

Signatures serve both to impose constraints on the bodies of modules and to restrict the information that 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 a *wide-spectrum* language (cf. [Bau85]) 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 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. Axioms in a signature specify properties that 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 proof. 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 that refers to it to be typechecked and compiled without making reference to the details of the code in the module body.[1] 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

[1] This aim is not fully achieved in SML because SML signatures are "transparent", so details of the representation of types in a module body that are not mentioned in its signature can be exploited by clients — cf. e.g. [Ler94]. EML signatures are "opaque", see Section 3, so this is not a problem there.

program, the question mark will be replaced by code that satisfies the axioms. A question mark may also be used in place of the type expression on the right-hand side of a type declaration, or even as a placeholder for the entire module body in a structure/functor declaration.

In EML, each structure comes equipped with a signature (this is optional in SML) containing the information that 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. Similar remarks apply to functors, which must be equipped with both an input signature (also required in SML) and an output signature (optional in SML).

Formal development of a system typically 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 that 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.[2]

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 that can be tackled separately from the others. Such steps give rise to proof obligations that must be discharged 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 [ST89], [ST91], [Kaz92a] and [San93] for further details, and see [ST89], [San91] and [SdS95] for examples of EML-style formal software development.

[2]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.

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 that 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` that 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 `contract`, `NF` and `strategy`: the axiom `forall t => (NF t) proper` is true if for all terms `t` the evaluation of `NF t` neither fails to terminate nor raises an exception; the second axiom requires that `strategy t` raises an exception if and only if `t` is in normal form, and that the redex selected by `strategy` otherwise can be properly contracted. Typical for EML is here the mixture of logical connectives and programming language constructs. Incidentally, the arrow `=>` appears in a formula like `forall t => (NF t) proper` for the same reason as it appears in a functional expression like `fn x => x+1`, which is SML's syntax for $\lambda x.x + 1$.

The functor `Reduce` gives us an evaluation function `eval`, as specified in the "included" signature `EVAL`, for any rewrite system given as a structure matching `TERMSIG`. From the parameter interface `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 important, 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

```
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

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 SML boolean expressions, with the meaning of the new constructs chosen to be as simple and natural as possible under the circumstances. Within the limits imposed by these constraints, 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[3] to the value `true` rather than evaluating to the value `false`, or evaluating to a packet, or failing to terminate. 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.

The identification of logical formulae used as axioms with boolean expressions of EML was a major design decision of the language of EML axioms. An alternative would be to introduce an additional type for logical formulae, subsuming boolean expressions via a coercion amounting semantically to the "evaluates to `true`" judgement, with additional logical connectives separate from those supplied by SML for booleans. This would seem to put us on familiar territory with a clear separation between the layer of computations and the layer of logical assertions, but the resulting system would be far from standard. The complications introduced by exceptions and potential non-termination would still be present, albeit at a lower level, and the intricacies involved in quantification (see below) would not disappear.

This identification requires EML to extend the language of SML boolean expressions with constructs corresponding to logical equality, assertions about the outcome of evaluating expressions, and quantification. The syntax of these and (a sketch of) their meaning is as follows — see Section 4.3.1 for some further details concerning their semantics.

The "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 also requires this to be an equality type. If both *exp* and *exp'* produce values, then the result of *exp*`==`*exp'* is `true` if and only if inserting these two values into the same expression always yields the same result. In particular, logical equality on function types is intended to be extensional in "logical-relation style" [Mit90]. Thus, if f, f' are both of type $\tau \to \tau'$ then f`==`f' entails[4]

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

[4] For obscure technical reasons this is not absolutely true for logical equality as defined in [KST94b]. An alternative definition of logical equality, to be included in the next version of the EML definition, will

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

— see below for the meaning of quantification. But the opposite entailment might fail if f, f' involve locally-declared exceptions [PS93]. Logical equality is also "extensional" for packets and non-termination, i.e. *exp*`==`*exp*$'$ is `true` if *exp* and *exp*$'$ both fail to terminate, or both produce the same packet.

The following additional constructs are provided for building axioms that constrain the outcome of computing the value of an expression *exp*:

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*,[5] 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 [GLT89], 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 expression 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:

1. In order for `forall` $x : \tau$ `=>` *exp* to be `true`: for every instance τ' of τ, the expression $exp[x := v]$ must be `true` for every expressible value v of type τ'.

2. In order for `exists` $x : \tau$ `=>` *exp* to be `true`: for every instance τ' of τ, the expression $exp[x := v]$ must be `true` for some expressible value v of type τ'. (Typically[6] it is enough to provide such a v of type τ itself. Ditto for case 3.)

3. In order for `forall` $x : \tau$ `=>` *exp* to be `false`: for every instance τ' of τ, the expression $exp[x := v]$ must be `false` for some expressible value v of type τ'.

4. In order for `exists` $x : \tau$ `=>` *exp* to be `false`: for every instance τ' of τ, the expression $exp[x := v]$ must be `false` for every expressible value v of type τ'.

---

repair this defect.

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

[6] In [KST94b] we made the mistake of assuming that this is always the case, even though it need not be when the value of $exp[x := v]$ depends on the type variables in τ. This error will be corrected in the next version of the EML definition.

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` $x\!:\!\tau$ `=>` *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 some other instance of τ such that $exp[x := v']$ is `false`.

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

```
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 expression to be `false`, since this is what happens when (for example) `x:int` and `xs:int list`. But when `x:unit` (`unit` is a built-in type having just one value, written `()`) and `xs:unit list`, the value of the expression is `true` since lists of type `unit list` are uniquely determined by their length. As a consequence, this expression has no value whatsoever. Fortunately, such odd examples occur rarely! An example of a quantified expression that is `true` is

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

because for any list type, the empty list has the property required for `ys`.

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 [SST92] 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 match 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 [ST86]; we have temporarily abandoned this approach, in part because of the difficulty of ensuring consistency with the existing definition of SML (but see [Kaz92b]). 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 devotees of functional programming [Bac78]. (In hindsight, the inclusion of imperative features would seem to add less complexity than we originally anticipated, because the presence of exceptions leads to some of the same complications.) The second restriction seems appropriate in a specification and formal development framework in which interfaces play a central role, in contrast to a programming language where the need to supply explicit interfaces 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 that 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 [ST89] for more on the methodological technicalities behind this restriction, and see [Tar92], [Ler94] and [HL94] for recent work having similar motivations.[7] 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 that causes certain pathological but well-formed SML programs to be regarded as incorrect. This is demonstrated by the following example:

[7]The original design of the SML module system [MacQ86] proposed an additional kind of structure, a so-called *abstraction*, for which the stricter "opaque" interpretation of signatures taken in EML would apply. This was unfortunately not included in SML as defined in [MTH90] although some SML implementations provide it as a non-standard extension [AM93].

```
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 is ill-formed according to the verification semantics of 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 (even "ugly" ones) satisfying these restrictions are also well-formed according to EML. Of course, it is clear that it will be easier to reason about the correctness of some 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 quantified expression or question mark. To eliminate "programs" that depend on the lack of code, it is essential to define `NoCode` as a special exception that cannot be trapped by any surrounding handler. As another example, special care is taken in the static semantics of EML (see Section 4.1) to ensure that the presence of axioms does not influence the result of typechecking signatures. Then regarding all the axioms in an EML program as comments results in a well-formed SML program.

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 [Kah93]). Not only is its syntax formally defined — this is not unusual — but also the meaning of SML programs is determined unambiguously by a formal mathematical semantics [MTH90], [MT91]. This is given in the form of so-called *natural semantics* [Kah88] (or structural operational semantics

[Plo81]) via deduction rules that 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 language and the semantics for the module language. Then, the semantics for the core and the semantics for modules are each 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 syntax 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, 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.[8] 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).

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, but 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 at this point 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 the definition of EML. This part of the semantics captures the requirement that modules are correct with respect to their interfaces. 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.

[8] 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 [Tof88].

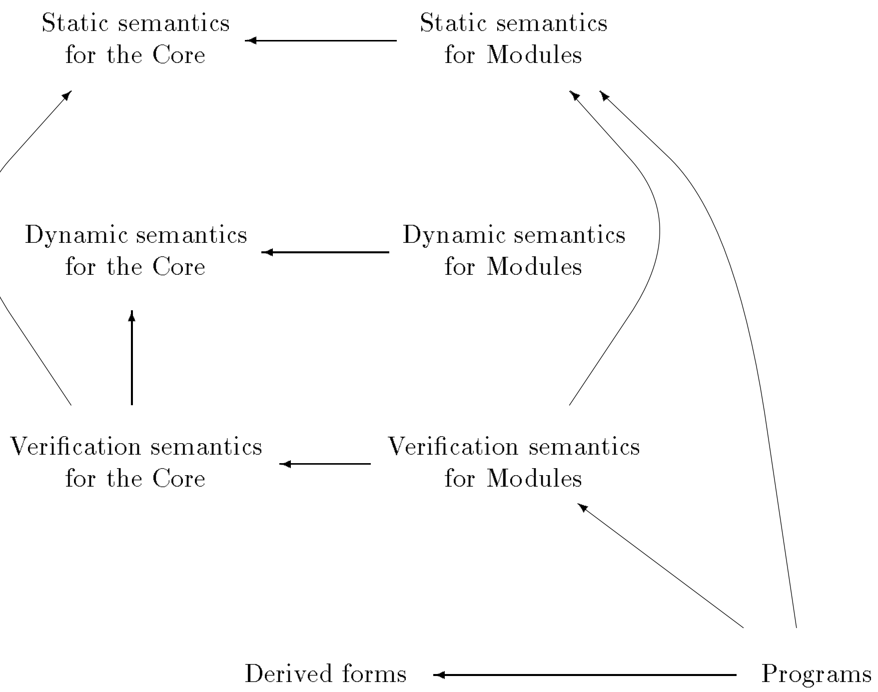

Figure 2: Direct dependencies between parts of the EML semantics

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 determine the meaning of a universally quantified expression) — hence the dependency on the static semantics. The dependency on the dynamic semantics stems from the need to interpret axioms describing evaluation properties of expressions (for example, stating that an expression terminates) and to determine exactly what the expressible values are. 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. Figure 2 is a diagram of the direct dependencies between the various parts of the semantics.

In the rest of this section we present fundamental ideas that are important for each part of the semantics — see [KST94b] for the complete definition. 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 the definition of EML. We go into more of the technical details there, and the reader should be warned that these are somewhat intricate. This should give some idea of how many issues had to be taken into account in the course of work on the definition. The definition of the syntax is not discussed, although certain tricky problems arise there due to unconventional features of SML's syntax.

4.1 Static semantics

The static semantics of EML describes the process of *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[9] $C \vdash exp \Rightarrow \tau$. 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 $C \vdash \texttt{[1]} \Rightarrow \texttt{int list}$ and $C \vdash \texttt{[]} \Rightarrow \texttt{int list}$ (for any[10] context C). Note, however, that we also have $C \vdash \texttt{[]} \Rightarrow \alpha\,\texttt{list}$, where $\alpha\,\texttt{list}$ is the type of lists over arbitrary type α. The *polymorphic* generalisation of this type is written as $\forall\alpha.\,\alpha\,\texttt{list}$. It is formed when an expression of type $\alpha\,\texttt{list}$ is bound to an identifier (provided α is not fixed by the context). $\forall\alpha.\,\alpha\,\texttt{list}$ may be instantiated to any type of the form $\tau\,\texttt{list}$; we write e.g. $\forall\alpha.\,\alpha\,\texttt{list} \succ \texttt{int list}$.

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 $C \vdash dec \Rightarrow E$, and for example we have $C \vdash \texttt{val a = 5} \Rightarrow \{\texttt{a} \mapsto \texttt{int}\}$. Examples involving function declarations are no more complicated: we have $C \vdash \texttt{val f = fn x => [x]} \Rightarrow \{\texttt{f} \mapsto \texttt{int} \to \texttt{int list}\}$, as well as $C \vdash \texttt{val f = fn x => [x]} \Rightarrow \{\texttt{f} \mapsto \forall\alpha.\,\alpha \to \alpha\,\texttt{list}\}$.

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!):

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

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 $\texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end}$ 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 $C \vdash \texttt{let val f = fn x => [x] in f 5 end} \Rightarrow \texttt{int list}$ (for any context C).

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

[10] 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. [MTH90].

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, a structure expression elaborates to a static environment E, which stores typing information about the objects declared within the structure, together with a *structure name* m (a unique internal tag) 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 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. 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 that maps `t` and `c` to a new, otherwise unknown type. Consequently, in the context of the above structure declaration, in EML we cannot form expressions like `S.c+2` — this is not well-typed in EML, although 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 achieved by accumulating all the types used in static elaboration of a phrase in an additional component of the result of elaboration — a so-called *trace* — for use by the verification semantics. One can think of a trace as an annotation of the entire parse tree for the phrase with results of the static analysis of each of its subphrases. The presence of traces 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:[11]

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

[11]The third premise, which requires that the type of *exp* does not use any new type names not mentioned

Here, the trace 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. (Elaborating an expression produces an additional result U, the set of *unguarded type variables*, used to keep track of the scope of explicit type variables. This issue is treated semi-formally in [MTH90].)

An additional problem is that the static semantics may "choose" different types for some parts of an expression without affecting the type of the expression as a whole. 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 `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 $C \vdash$ `val f = fn x => [x]` $\Rightarrow \{\texttt{f} \mapsto \texttt{int} \to \texttt{int list}\}$, or via $C \vdash$ `val f = fn x => [x]` $\Rightarrow \{\texttt{f} \mapsto \forall\alpha.\,\alpha \to \alpha\,\texttt{list}\}$, in each case yielding $C \vdash$ `let val f = fn x => [x] in f 5 end` $\Rightarrow$ `int list`, but with different traces. 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 [DM82] ($\forall\alpha.\,\alpha \to \alpha\,\texttt{list}$ for `f` here), and so an appropriate principality requirement is imposed on traces, 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 [MT91] for a precise statement and proof) that is retained by EML and extends to the existence of principal traces.

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

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

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). In the semantics of EML, such infinitary conditions are formalised by means of *higher-order rules*. For instance, the above

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 otherwise accidentally become equal. See [Kah93].

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 trace:

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

The last premise of this rule requires that any trace corresponding to an elaboration of *dec* in the given context may be obtained from the trace γ by instantiating new type names introduced in the corresponding elaboration of *dec* and (possibly) instantiating type schemes contained in γ. As explained above, this requirement, which is stronger than just principality of the resulting environment, is necessary for the semantics of EML.

The static semantics of the axioms of EML requires little comment. Boolean expressions used as axioms are typechecked exactly as usual. The only subtle point is that an explicit restriction must be imposed to prevent the static analysis of an axiom from influencing the results of the static analysis of the phrase in which it occurs. For example, the signature expression

```
sig
    type t
    val a:t
    axiom a=5
end
```

is not statically well-formed in EML, since the axiom forces the type `t` to share with `int`. The restriction is required to ensure that treating the axioms in an EML program as comments yields a well-formed SML program.

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 usual inductive interpretation of the rules unambiguously defines the true judgements of the semantics. In order for such a reading to be valid, "impredicative" dependencies between premises and conclusions in higher-order rules must be avoided.[12] This problem was already present in the semantics

[12] This was not quite ensured in [KST94b], e.g. in the definition of logical equality. This deficiency will

of SML [MTH90], but was less explicit there since the problematic premises were formulated in terms of concepts defined semi-formally in English and separately from the rules. The requirement of principality was the most visible example of this, and the potential problem is resolved by a theorem in [MT91]. (Capturing principality is the only use of higher-order rules in the static semantics of EML, and their interpretation may be explained in exactly the same way as for principality in SML. Since there is an algorithm that computes principal types, all judgements in the static semantics are decidable for exactly the same reason as in SML.) In the EML semantics, the need for higher-order rules arises much more frequently and prominently than in SML since the verification of axioms naturally involves infinitary premises because of the presence of e.g. quantifiers and logical equality, see Sections 2.4 and 4.3.1. Thus the semi-formal style used in SML seemed inappropriate.

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 and in quantification over expressible values — hence, the dynamic semantics is needed to make the formal definition of EML self-contained. Another important reason is that we want to formally define a basis for experiments with unfinished programs. EML programs, even incomplete ones containing specification constructs, are viewed as "partially executable". The idea is that such programs should be executable insofar as this is possible, and that evaluation should proceed as in SML for the parts that 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[13] $E \vdash exp \Rightarrow v$, stating that in the (dynamic) *environment* E, the expression exp evaluates[14] to the value v, where environments store the values of objects that are currently defined. For example, we have $\{\mathsf{a} \mapsto 27\} \vdash \mathsf{a} * 37 \Rightarrow 999$. Environments are built by declarations, with corresponding judgements of the form $E \vdash dec \Rightarrow E'$ expressing the fact that in

be alleviated in the next version of the EML definition which will admit a well-founded stratification of judgements such that premises of rules belong to strata that are no "higher" than that of the conclusion of the rule itself, and the premises of rule premises in higher-order rules, as well as negated premises, belong to strictly "lower" strata.

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

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

the environment E the declaration *dec* evaluates to the environment E', which stores the values of objects declared in *dec*. For instance, we have $E \vdash$ `val a = 27` $\Rightarrow \{$`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:

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

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 [Lan64]. Closures are expanded when applied to arguments, and a rather elaborate scheme of self-expansion is used to model recursion (see [KST94b], [MTH90] 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 certain 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 $E \vdash \mathit{exp} \Rightarrow p$ (in the environment E the expression *exp* evaluates to the packet p). To express the two possibilities jointly, we write $E \vdash \mathit{exp} \Rightarrow v/p$, 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:

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

(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 (i.e. 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 absence of code being represented by the placeholder `?`) 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 specification constructs may be encountered in evaluation of EML expressions outside axioms. When this is the case, a special exception `NoCode` is raised. `NoCode` cannot be handled explicitly in programs, as mentioned 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 (where [`NoCode`] denotes the packet with exception name `NoCode`):

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

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 the specification 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, the *specification flag*. This flag is raised whenever evaluation encounters a specification construct, or when a closure is produced that depends on a specification construct whose evaluation may be required when the closure is applied to an argument. When the specification flag is not raised during the evaluation of a phrase, the results provided by the dynamic semantics of EML coincide both with the results of the dynamic semantics of SML[15] and with the results of the verification semantics for the core of EML

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

(see Section 4.3.1 below). When the dynamic semantics of EML does not yield a result, the verification semantics cannot yield one either, nor can the dynamic semantics of SML. However, obtaining a result with the specification flag raised provides no reliable information about the behaviour of either the verification semantics or the dynamic semantics of SML: they may yield a different result or no result whatsoever.

The role of the dynamic semantics for EML modules is purely to define a basis for experiments with unfinished programs (see the beginning of this section). The other parts of the semantics do not depend on this part, as Figure 2 indicates. It 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 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 $s, B \vdash strexp \Rightarrow E, s'$. The EML-specific 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.

No higher-order rules are used in the dynamic semantics of EML. It follows that the judgements defined here are semi-decidable, so evaluation is implementable although (of course) it may fail to terminate. This is just the same as in SML.

4.3 Verification semantics

Although we provide a dynamic semantics for EML, the main stress in a specification and formal development framework like EML is rather on the verification of correctness assertions that are present in EML phrases. Consequently, we view the verification semantics as the essence of the formal description of EML. The heart of this part of the semantics is the check that structures and functors *match* their signatures, which in particular means that they satisfy the axioms given in the signatures. Signature matching is described by the verification semantics for modules, and the meaning of axioms is described by the verification semantics for the core. Verification of an EML phrase does not result merely in a binary statement indicating 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*[16] to semantic objects.

Higher-order rules are used throughout the verification semantics. We claim that some of the judgements defined here are not semi-decidable, so (as one would expect) there is no sound and complete proof system for EML.

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

4.3.1 Verification semantics for the core

The verification semantics for the EML core is in many respects 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 state. A difference with respect to the dynamic semantics stems from the fact that verificating an expression requires information that is not available in the expression itself or in the dynamic environment. This information comes from various sources. As mentioned earlier, the interpretation of axioms depends on type information that appears in the trace produced by the static semantics. Expressions are substituted for question marks by reference to the *question mark interpretation* produced by the verification semantics for modules, see Section 4.3.2. The verification semantics thus interprets expressions in the context of a *model* consisting of a dynamic environment (with some type information added), a trace for the expression at hand, and a question mark interpretation; the corresponding judgement has the form $s, M \vdash exp \Rightarrow v/p, s'$. Each state is augmented with (among other things) two *type interpretations*: one is used to interpret types that were defined using question marks in *other* phrases, and the second, produced by the verification semantics for modules, penetrates the abstraction barrier imposed by interfaces for use in the interpretation of logical equality and quantifiers, see below. Similar remarks apply to declarations, where judgements have the form $s, M \vdash dec \Rightarrow E/p, s'$.

The specification constructs of EML, such as `==`, `terminates` and `forall`, are 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.

The verification of logical equality exp_1 `==` exp_2 proceeds in two stages. First, the expressions exp_1 and exp_2 are classified according to whether they (i) verificate to values, (ii) verificate to packets, (iii) fail to evaluate, or (iv) fail to verificate without failing to evaluate. If (iv) holds for either of the two expressions then we have no reliable information about its value (see the discussion of the `terminates` construct below) and exp_1 `==` exp_2 is undefined; otherwise it is always defined. Most typically, if (i) holds for the two expressions, we proceed by comparing their values v_1 and v_2 (see below). The result of verification is determined directly if (iii) holds for the two expressions — then exp_1 `==` exp_2 verificates to `true` — and if they fall into different categories as described by (i), (ii) and (iii) — then exp_1 `==` exp_2 verificates to `false`. If (ii) holds for the two expressions and the exception names in the resulting packets are different, then exp_1 `==` exp_2 again verificates to `false`. Otherwise, values v_1 and v_2 are extracted from the packets.

To resolve the remaining cases, the values v_1 and v_2 obtained from exp_1 and exp_2 as above are *compared*. This comparison is always defined and yields `true` if v_1 and v_2 are indistinguishable, i.e., if there is no expression exp that yields different outcomes in two environments distinguished only by assigning to some new variable `x` in exp the values v_1 and v_2 respectively.

This informal explanation is not as precise as it appears. The phrase "expression exp" may seem innocuous, but it omits one crucial ingredient: a static context C in which exp

is well-formed. There are various choices for C, each giving a distinctive flavour to the comparison. We use a context C in which every constructor is available (disregarding scoping) and associated with its original type (disregarding abstraction barriers). This also determines the two environments in which the value of *exp* is to be obtained: they carry all the values and types mentioned in C, plus the binding of the new variable `x` to v_1 and v_2 respectively. This decision makes it possible to distinguish values even if the current program context is not capable of making such a distinction. A small example:

```
datatype t = A | B | C of int->int
val z = A and y = B
datatype cover = A of int | B
```

In the context produced by the above sequence of declarations, no means are provided to distinguish the values of `z` and `y`: since the declaration of `cover` hides the constructors `A` and `B` of `t`, expressions like `case x of A => true | _ => false`, which distinguish between the values of `z` and `y` before the declaration of `cover`, become ill-formed after it (`t` is not an equality type so expressions like `x=z` are ill-formed). The verification semantics builds a context that restores the constructors `A` and `B` hidden by the declaration of `cover` (without hiding the constructors from *that* declaration) and these two values then become easily distinguishable. The use of this enriched context means that the result of comparison is unaffected by the textual position of the formula. For example, the expression `z==y` will verificate to `false` regardless of whether it occurs before or after the declaration of `cover`.

In spite of the way that a structure's interface signature abstracts away from the details of the structure body, hiding the concrete realisation of its types and other components (see Section 4.3.2 below for details), each model incorporates a particular choice of these details satisfying the axioms in the signature.[17] Comparison of values takes this information into account. Consider the following example:

```
signature TWOVAL =
    sig
        type t
        val c: t
        val d: t
    end;
structure T: TWOVAL = ?
```

Any model will bind `T.t` to some type and `T.c`, `T.d` to values of that type. In a model that happens to bind `T.c` and `T.d` to values that are distinguishable in this model, the expression `T.c==T.d` will verificate to `false`; in a model that binds them to values that are indistinguishable in this model, it will verificate to `true`. If the choice of bindings in the model were not taken into account and only the information in the signature were available for comparison of values, then `T.c==T.d` would always verificate to `true` since no contexts would be available to distinguish between `T.c` and `T.d`.

[17] This information is partly in the type interpretations that are contained in the state.

The result of verificating an expression of the form *exp* `terminates` indicates whether the verification of the expression *exp* terminates or not, *provided* we have reliable information to determine this. This proviso is crucial to avoid the usual paradoxes involving expressions *exp* that contain the termination predicate itself. Reliable information about termination of verification is provided by the dynamic semantics. If in the dynamic environment obtained by removing type information from the current verification environment *exp* evaluates to a value v or packet p without raising the specification flag, then the verification of *exp* will terminate as well (and yield the same value) — the circumstances under which the dynamic semantics raises the specification flag are carefully chosen to ensure this property. Consequently, we can then reliably verificate *exp* `terminates` to `true`. If, however, the evaluation of *exp* results in a value or packet with the specification flag raised, the termination information thus obtained is unreliable and we indicate this fact by raising the special exception `Abuse`. Finally, if there does not exist a successful evaluation of *exp* then *exp* `terminates` verificates to `false`. An important consequence of this definition is that the verification of *exp* `terminates` for expressions *exp* that do not depend on specification constructs is always determined and yields `true` or `false` consistently with the termination behaviour of this expression in the dynamic semantics for SML.

Intuitively, a universally quantified formula `forall` x `=>` *exp* is true if $exp[x := v]$ is true for all values v. Since SML is a typed language, we have to modify this statement by requiring v to have the type that x has. But what is the type of x and how do we obtain all its values?

The answer to the first question is given by the static semantics of EML.[18] However, it is only a *partial* answer, since the type assigned to x (available from the trace) is its principal type, as explained in Section 4.1. For the purposes of quantification instantiation of this type is required as it increases the set of values: for example, α `list` only has the single value `[]` (the empty list), but we get non-empty lists as well when α is instantiated to non-empty types. This explains why it is counter-intuitive to stick solely to the principal type for the purposes of quantification: we want to be able to state properties of non-empty lists without giving a particular instantiation of α, thus for universal quantification over α `list` we have to consider all possible instantiations of α. Consequently, a universally quantified expression `forall` x `=>` *exp* verificates to `true` if $exp[x := v]$ verificates to `true` for all values v of all instances of the type of x, as presented in Section 2.4.

This might suggest that a universally quantified expression should be `false` if $exp[x := v]$ verificates to `false` for *some* value of *some* type instance, and analogously for existentially quantified expressions verificating to `true`. We have, however, decided against the second "some", in part because it leads to certain anomalies as the following example illustrates.

```
signature SIG =
    sig
```

[18]This is not the whole story. The type inferred by the static semantics needs to be modified to take the realisation of types in structures into account, see Section 4.3.2.

```
        val f: int -> int list
        axiom exists xs => (forall y => f y = xs)
    end;
structure S:SIG =
    struct
        val f:''a -> ''a list = ?
        axiom exists xs => (forall y => f y = xs)
    end
```

Both the structure and the signature contain literally the same axiom, and signature matching permits the structure to be more polymorphic than the signature specifies, so we would expect this declaration to verificate (and indeed it does verificate in EML). Had we instead adopted the above suggestion, then the verification of the structure body would admit a model mapping `f` to the polymorphic function `fn z => [z]` since `[()]` is a witness for `xs` in the existential axiom in the structure body with `f` considered over the type `unit -> unit list`. Clearly, for this choice of the function `f`, the axiom in the signature cannot be satisfied, since `f` is considered there over the type `int -> int list`.

Thus, as indicated in Section 2.4, we require witnesses to existential axioms *for each instance* of the type of the quantified variable. Therefore, all models admitted by verification of the structure body above map `f` to functions such that for each equality type τ, a witness `xs:`τ `list` can be provided for which `forall y:`τ `=> f y = xs` verificates to `true` — for example, `f` could be mapped to the function `fn z => []`, with the witness for `xs` being `[]` for each type τ `list` — and then we are guaranteed that there is a witness of type `int list`, as required in `SIG`.

We decided to define the set of all values of a type τ to be the values that can be expressed in the language, i.e. each value considered can be obtained from an expression *exp* of type τ. Again, two aspects of this characterisation have to be made precise: we have to decide in which static context *exp* should have type τ, and we have to choose whether "obtain" refers to the dynamic or verification semantics. For the former, a solution similar to that for logical equality is chosen: we disregard scoping and abstraction barriers and quantify over the values of the type realisation in the model at hand. The following structure declaration verificates, as expected:

```
structure S : sig type t
                  val c: t
                  val p: t -> bool
                  axiom exists x => p x
              end =
    struct type t = int
           val c = 1
           val p = fn y => y=2
    end
```

To verificate the axiom in the signature we use the type `t` as realised (by `int`) in the structure body, and then the axiom clearly holds. Had we instead relied on the type `t` as abstractly characterised by the signature, the axiom would not hold, since the only value of `t` we could construct at this level is given by the constant `c`, and `c` is not a witness for the existential axiom in the signature.

The choice whether we obtain values by evaluation or verification has to be decided in favour of evaluation to avoid vicious circles — after all, the verification of a quantified expression produces a value (of type `bool`) itself. A complication arising from this choice is that we have to check that the evaluation of the expression *exp* used to generate a value does not raise the specification flag. This is necessary to ensure that the verification of *exp* yields the same value. A consequence is that the values considered cannot depend on specification constructs.

The following rule for verificating a universally quantified formula to `true` takes all of the above points into account. This is the actual rule as it appears in [KST94b] — except that quantification over a list of patterns, rather than over a single variable, is allowed there — and so includes states and other details that we have been glossing over in the preceding discussion. We give it here to illustrate how involved the technicalities may become — admittedly, this is one of the most complicated rules in the semantics! The reader should not be alarmed if some of the details are not completely clear; they become clear only in the light of details given in other rules.

$$\frac{\mathrm{Comp}(FE,s)=VE \qquad \dfrac{\begin{array}{c}\gamma \succ \gamma_1 = (C,\tau)\cdot\gamma_2 \quad s^{\#\#}(C)+\mathrm{Stat}\,VE \vdash_{\mathrm{STAT}} \mathit{atexp}^{\bullet} \Rightarrow \tau', \emptyset, \gamma_3 \\ s^{\#\#}(\tau)=\tau' \quad \mathrm{Dyn}(s, FE+VE) \vdash_{\mathrm{DYN}} \mathit{atexp}^{\bullet} \Rightarrow v_{\mathrm{DYN}}, (\top, \mathit{ens})\end{array}}{\exists s'.\ s, (FE+VE, \gamma_1\cdot\gamma_3) \vdash (\texttt{fn}\ x\ \texttt{=>}\ \mathit{exp}^{\bullet})\ \mathit{atexp}^{\bullet} \Rightarrow \texttt{true}, s'}}{s, (FE,\gamma) \vdash \texttt{forall}\ x\ \texttt{=>}\ \mathit{exp}^{\bullet} \Rightarrow \texttt{true}, s}$$

A few comments are required to link the notation used to the discussion above. Comp(FE, s) produces an environment in which every constructor is available, disregarding scoping. The function $s^{\#\#}$ interprets question marks in types taken from traces and penetrates abstraction barriers, thus referring to the type realisation in the model at hand. (Note that this is one of the rules that makes use of the information recorded in the trace, rather than merely building traces and passing them around.) The functions Stat resp. Dyn extract static resp. dynamic information from the verification environment, and $\vdash_{\mathrm{STAT}}$ resp. $\vdash_{\mathrm{DYN}}$ refer to the judgements of the static resp. dynamic semantics. $\top$ in the state resulting from the dynamic evaluation of $\mathit{atexp}^{\bullet}$ means that evaluation does not raise the specification flag. Finally, the bullet in $\mathit{atexp}^{\bullet}$ and $\mathit{exp}^{\bullet}$ restricts to phrases not containing question marks.

4.3.2 Verification semantics for modules

EML module phrases verificate to sets of semantic objects, rather than just to a single semantic object as in the verification semantics for the core. For instance, in a given basis, EML structure expressions verificate to sets of *(verification) environments*,[19] with

[19] In fact, just as in the dynamic semantics of EML it was necessary to consider an environment together with a state, in the verification semantics structure expressions verificate to sets of elements that are pairs of an environment and a state. For presentation purposes we disregard states in the rest of this section.

the corresponding formal judgements having the form $B, \gamma \vdash \mathit{strexp} \Rightarrow \mathcal{E}$. 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, the resulting set may be empty — for example, an axiom like `axiom a>5 andalso a<3` in *strexp* results in the empty set of environments — but notice 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 its attached interface filters the information available, hiding the details given in its body. This is modelled by taking the "verification meaning" of a structure to be the set of all environments matching its interface, rather than the particular environment (or set of environments) 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) signature* 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. Then, check that each environment $E \in \mathcal{E}$ matches the signature Σ. This step is where the real verification takes place: it involves checking whether the axioms incorporated in Σ are satisfied by each $E \in \mathcal{E}$.

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

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 if no environment matches Σ, and the verification of *strexp* results in the empty set of environments. Of course, such a structure would not be of much use!

Here is (a simplified version of) the rule that 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, see below for a discussion of the details involved in matching an environment against a signature. Second, we have elided traces. Third, we use 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 semantics uses a higher-order rule to express this more formally. 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 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 $\{\mathtt{a} \mapsto i\}$.[20] It is then checked that each of these environments does indeed match the signature, 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 signature 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 andalso 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 signature fails, since E_3 does not satisfy the modified axiom. Thus, the verification of this structure declaration fails: the structure declaration is (not surprisingly) incorrect.

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
```

[20]We omit type information in verification environments here and below.

The verification of these two declarations will result in the set of environments containing $\{\texttt{S} \mapsto S_t, \texttt{T} \mapsto T_t\}$ and $\{\texttt{S} \mapsto S_f, \texttt{T} \mapsto T_f\}$, where $S_t = \{\texttt{a} \mapsto \texttt{true}\}$, $T_t = \{\texttt{b} \mapsto \texttt{true}\}$, $S_f = \{\texttt{a} \mapsto \texttt{false}\}$ and $T_f = \{\texttt{b} \mapsto \texttt{false}\}$. However, the resulting set of environments does not contain for example $\{\texttt{S} \mapsto S_t, \texttt{T} \mapsto T_f\}$ 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 a single environment for each binding of `S` considered (as before), this information is filtered out by the interface provided in the binding as described earlier. 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.

The sets of environments above arise through interaction between the verification semantics for modules and for the core. At the point where a declaration is passed from the module semantics to the core semantics, a question mark interpretation (which is required as a component of the model used to interpret core phrases) is chosen arbitrarily. Verification may succeed or fail for this choice; one possible reason for failure is that an axiom contained in the declaration may not verificate to `true` (see Section 4.3.1). This does not necessarily mean that the declaration is incorrect. It means only that the particular choice of question mark interpretation is unsuccessful and will not contribute to the result of the verification semantics of the declaration. Only those environments resulting from a successful verification of the declaration for some choice of the interpretation of question marks are included in the result of the verification of the declaration at the structure level. This is captured by the 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 verification semantics for the structure-level declaration enclosed in `struct ... end` tries to verificate its enclosed sequence of 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.

The constraints imposed by consecutive axioms accumulate by gradually restricting the set of environments constructed by the verification semantics. For example, the verification semantics for the following structure expression yields $\{E_6\}$:

```
struct
    val a: int = ?
    axiom a>5 andalso a<8
    val b = a+2
    axiom a mod 2 = 0
end
```

The order of such axioms does not matter, and they may be arbitrarily intermingled as above with core declarations (provided that identifiers used in axioms remain in scope). The situation is different when substructure declarations are present. Consider the following structure expression:

```
struct
    val a: int = ?
    axiom a>5 andalso a<8
    val b = a+2
    structure A:sig val c:int; axiom c mod 3 = 2 end =
        struct val c:int = b end
    axiom a mod 2 = 0
end
```

The declaration of the substructure `A` is required to verificate in both E_6 and E_7. Since its verification fails for E_7, the verification of the overall structure expression fails. In contrast, changing the order of the final axiom (which filters out E_7) and the substructure declaration gives the following structure expression which verificates successfully, since the substructure `A` verificates in E_6:

```
struct
    val a: int = ?
    axiom a>5 andalso a<8
    val b = a+2
    axiom a mod 2 = 0
    structure A:sig val c:int; axiom c mod 3 = 2 end =
        struct val c:int = b end
end
```

In the same way as EML quantification is based on expressible values (see Sections 2.4 and 4.3.1), question mark interpretations QI map question marks to expressions, not to values, with the difference that expressions here are allowed to contain specification constructs. 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 = ?:int in g c end
```

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

The treatment of question marks in type bindings is somewhat different. The static semantics guarantees that whatever replacement a question mark interpretation provides for a question mark type (such that certain attributes are preserved), 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 by interpreting the types derived during static analysis using one or both of the type interpretations recorded in the state.

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 SIGA =
    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 `SIGA`. When a structure is matched against `SIGA`, the type instantiation arising from the match is applied to the axiom in `SIGA`. The semantic object associated with axioms in signatures consists mainly of the syntax of the axiom itself — see below for details — and this is not affected by the application of the type instantiation. But the syntax of the axiom is accompanied by its trace, and this *is* affected. The result is that the existential quantifier in the above axiom will range over the realisation of `t` given by the type instantiation.

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, where their interpretation may influence the verification of axioms. For example, a structure matching the following signature

```
signature SIGB =
    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`, a value `b` has to be found such 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 [Far92]).

Axioms in signatures are stored in the form of so-called *generalised axioms*. The two most important forms of generalised axiom arise in the signatures `SIGA` and `SIGB` above. There are no hidden components in `SIGA`, so the resulting generalised axiom has the form $(B, \gamma, axdesc)$ where $axdesc$ is the syntax of the axiom as it appears above, γ is the trace produced for this phrase by the static semantics, and B is a basis for the interpretation of global identifiers in the axiom (in this case, just the identifier `true`). The purpose of the basis is exactly the same as that of the environment in a closure. The judgement form for satisfaction of a generalised axiom is $E \vdash A \Rightarrow \{\}$, which is read: in the environment E, the generalised axiom A holds. The environment E comes from the structure that is matched against the signature containing the axiom. For a generalised axiom of the form $(B, \gamma, axdesc)$, this judgement amounts to the statement that $axdesc$ verificates to `true` in the environment $B + E$, using a trace obtained from γ as explained above. Since the signature `SIGB` has hidden components, the resulting generalised axiom has a form that we can write as $\exists\Sigma.A$, where A is a "normal" generalised axiom (as in the previous example) for the visible part of the signature and Σ is the hidden part. For this to be satisfied, there must exist a structure expression $strexp$ that matches Σ (and satisfies its axioms) such that A is satisfied in an appropriate extension of the environment obtained from $strexp$.

The above presentation has focussed on the verification of structure expressions and structure declarations. This extends to the verification of functor declarations in the obvious way.

5 Final remarks

We have tried in this paper to provide a readable exposition of the definition 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.

The genesis of EML was the decision to design a framework for the specification and development of programs in SML, an existing real programming language, building on foundations in the theory of algebraic specification. EML was designed to be a *minimal* extension of SML, which led to a wide-spectrum formalism integrated with SML rather than a "pure" specification formalism added as a separate layer above it. The "property-oriented" algebraic style of specification was adopted rather than the use of explicit pre/post-conditions or model-oriented specifications. (But note that pre/post-conditions can be expressed easily in EML, and model-oriented specifications are subsumed once behavioural equivalence is taken into account.) An attempt was made to include as much of SML as possible, with the main omissions being imperative features and input/output. The only modification to the underlying programming language was the use of a stricter type discipline for module interfaces ("opaque" signatures), to enable separate development of the components of a modular system. A natural consequence of the decision to conform fully with SML was to re-use SML's existing formal definition, given in an operational style, and build the formal definition of EML on this basis. An obvious alternative would have been to provide a denotational or algebraic-style semantics for EML. In spite of the perceived advantages of this approach for reasoning about programs, it was rejected as it would have meant starting from scratch with no practical way of ensuring compatibility with SML. Adopting SML's semantics as our starting point almost forced a number of other decisions, including the modelling of functions as closures rather than as mathematical functions and quantification over expressible values only. It seemed convenient to reuse SML's constructs (`andalso`, `not`, etc.), with their usual semantics, in the language of EML axioms. This led to the decision to use boolean expressions, extended by the addition of a few specification constructs (`forall`, `terminates`, `==`, etc.) as axioms. A more conventional alternative that would make the logic closer to standard two-valued logic would have been to build a separate layer of logical expressions — this would refer to SML expressions but not vice versa — with a new set of logical connectives to combine them. We chose to make quantification over type variables implicit, as in SML, in the expectation that this would seem more natural to an SML programmer, although a simpler and more conventional choice would have been to require explicit type quantification.

The enterprise of engineering a sizable completely formal definition of a realistic, practically useful formalism is an inherently complex task. 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 [MTH90]. As a result of the study of the definition by a larger body of users, this was then followed by a number of subsequent changes included in [MT91]. And even now, some inaccuracies, weak points and minor mistakes in the definition are still being discovered [Kah93]. Nevertheless, as a whole, the SML definition 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. We expect that the definition of EML will undergo a similar process of revision. In fact, as various footnotes above indicate, a second version of [KST94b] is already on the way.

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

One issue that is not treated in [KST94b] is the role of behavioural equivalence in the methodology for formal development in EML as described in [ST89]. Following ideas concerning the use of axioms to specify encapsulated abstractions (see e.g. [Rei81], [GM82], [ST87]), in order to obtain correct results it is not actually necessary for the axioms in an EML signature to be satisfied "literally": it is enough if they are satisfied "up to behavioural equivalence", meaning that there is no way to detect failure to satisfy the axioms by performing computations that yield observable results (i.e. results of base types such as `bool`). This relaxation is required to adequately deal with certain examples of refinement involving choice of data representation.

Further study is needed before we will be able to change the present definition to permit axioms in signatures to be satisfied up to behavioural equivalence. Unexpectedly, the approach used in [ST89], via a definition of behavioural equivalence between models, will not achieve the desired effect here because of our use of models incorporating a rather concrete representation of types and values. It should be possible to take a different approach, which would involve a comparatively slight modification to the semantics of quantification and logical equality. It is first necessary to show that there is a satisfactory relationship between what this would yield and the behavioural equivalence relation used for the foundations of formal development, following [BHW94]; a first step in this direction is taken in [HS95].

The next major step in work on EML 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, and in fact we expect that work in this area (which is already underway) may force us to re-think some of the details in the design of the language of EML axioms. Furthermore, checking the formal soundness of these rules with respect to the semantics given in [KST94b] 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 definition 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 applications in which correctness is crucial. 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, Luis Dominguez and anonymous referees for helpful comments on a draft of this paper. Thanks also to Thorsten Altenkirch, Mike Fourman, Martin Hofmann, Robin Milner, and others for helpful discussions concerning the language of EML axioms. We owe special thanks to Robin Milner, Mads Tofte and Robert Harper for their work on the definition of SML, without which the research described here would not have been possible.

References

[AM93] A. Appel and D. MacQueen. Standard ML of New Jersey, version 0.93. AT&T Bell Laboratories (1993).

[Ast86] E. Astesiano, C. Bendix Nielsen, N. Botta, A. Fantechi, A. Giovini, P. Inverardi, E. Karlsen, F. Mazzanti, J. Storbank Pedersen, G. Reggio and E. Zucca. The draft formal definition of Ada. Deliverable 7 of the CEC-MAP project (1986).

[Bac78] 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).

[Bau85] F. 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).

[BHW94] M. Bidoit, R. Hennicker and M. Wirsing. Characterizing behavioural semantics and abstractor semantics. *Proc. 5th European Symposium on Programming*, Edinburgh. Springer LNCS 788, 105–119 (1994). Extended version to appear in *Science of Computer Programming.*

[BKLOS91] M. Bidoit, H.-J. Kreowski, P. Lescanne, F. Orejas and D. Sannella (eds.) *Algebraic System Specification and Development: A Survey and Annotated Bibliography.* Springer LNCS 501 (1991).

[BG81] 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).

[DM82] L. Damas and R. Milner. Principle type schemes for functional programs. *Proc. 9th Annual ACM Symp. on Principles of Programming Languages*, 207–212 (1982).

[Far92] J. Farrés-Casals. Verification in ASL and Related Specification Languages. Ph.D. thesis; report CST-92-92, Univ. of Edinburgh (1992).

[GLT89] J.-Y. Girard, Y. Lafont and P. Taylor. *Proofs and Types.* Cambridge University Press (1989).

[GM82] 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).

[GH93] J. Guttag and J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer (1993).

[Har89] R. Harper. Introduction to Standard ML (revised edition). Report ECS-LFCS-86-14, Univ. of Edinburgh (1989).

[HL94] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. *Proc. 21st Annual ACM Symp. on Principles of Programming Languages*, 123–137 (1994).

[HS95] M. Hofmann and D. Sannella. On behavioural abstraction and behavioural satisfaction in higher-order logic. *Proc. 6th Intl. Joint Conf. on Theory and Practice of Software Development*, Aarhus. Springer LNCS 915, 247–261 (1995).

[Kah88] G. Kahn. Natural semantics. In: *Programming of Future Generation Computers* (K. Fuchi and M. Nivat, eds.), 237–258. North-Holland (1988).

[Kah93] S. Kahrs. Mistakes and ambiguities in the definition of Standard ML. Report ECS-LFCS-93-257, Univ. of Edinburgh (1993). With addenda dated 14th February 1995 in `http://www.dcs.ed.ac.uk/~smk/SML/errors-new.ps.Z`.

[KST94a] S. Kahrs, D. Sannella and A. Tarlecki. The semantics of Extended ML: a gentle introduction. *Proc. Intl. Workshop on Semantics of Specification Languages*, Utrecht, 1993. Springer Workshops in Computing, 186–215 (1994).

[KST94b] S. Kahrs, D. Sannella and A. Tarlecki. The definition of Extended ML, version 1. Report ECS-LFCS-94-300, Univ. of Edinburgh (1994). Subsequent versions of this definition, when they appear, will be available by WWW from `http://www.dcs.ed.ac.uk/~dts/eml`.

[Kaz92a] E. Kazmierczak. Modularizing the specification of a small database system in Extended ML. *Formal Aspects of Computer Science* 4(1):100-142 (1992).

[Kaz92b] E. Kazmierczak. Model theory for Extended ML. Draft report, Univ. of Edinburgh (1992).

[Kri90] B. Krieg-Brückner. PROgram development by SPECification and TRAnsformation. *Technique et Science Informatiques* (1990).

[Lan64] P. Landin. The mechanical evaluation of expressions. *Computer Journal* 6:308–320 (1964).

[Ler94] X. Leroy. Manifest types, modules, and separate compilation. *Proc. 21st Annual ACM Symp. on Principles of Programming Languages*, 109–122 (1994).

[LHKO87] D. Luckham, F. von Henke, B. Krieg-Brückner and O. Owe. *Anna, a Language for Annotating Ada Programs: Reference Manual.* Springer LNCS 260 (1987).

[MacQ86] D. MacQueen. Modules for Standard ML. In: Report ECS-LFCS-86-2, Univ. of Edinburgh (1986).

[MG94] S. Maharaj and E. Gunter. Studying the ML module system in HOL. *Higher Order Logic Theorem Proving and its Applications.* Springer LNCS 859, 346–361 (1994).

[MT91] R. Milner and M. Tofte. *Commentary on Standard ML.* MIT Press (1991).

[MTH90] R. Milner, M. Tofte and R. Harper. *The Definition of Standard ML.* MIT Press (1990).

[Mit90] J. Mitchell. Type systems for programming languages. In *Handbook of Theoretical Computer Science, Vol. B* (J. van Leeuwen, ed.), 365–458. North Holland (1990).

[Pau91] L. Paulson. *ML for the Working Programmer.* Cambridge Univ. Press (1991).

[PS93] A. Pitts and I. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? *Proc. 18th Intl. Symp. on Mathematical Foundations of Computer Science*, Gdańsk. Springer LNCS 711, 122–141 (1993).

[Plo81] G. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Aarhus University (1981).

[Rei81] H. Reichel. Behavioural equivalence: a unifying concept for initial and final specification methods. *Proc. 3rd Hungarian Computer Science Conference*, 27–39 (1981).

[San91] 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).

[San93] D. Sannella. Static and logical correctness conditions in formal development of modular programs. Draft report, Univ. of Edinburgh (1993).

[SdS95] D. Sannella and F. da Silva. Case studies in Extended ML. Draft report, Univ. of Edinburgh (1995).

[SST92] D. Sannella, S. Sokołowski and A. Tarlecki. Toward formal development of programs from algebraic specifications: parameterisation revisited. *Acta Informatica* 29:689–736 (1992).

[ST86] 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).

[ST87] D. Sannella and A. Tarlecki. On observational equivalence and algebraic specification. *Journal of Computer and System Sciences* 34:150–178 (1987).

[ST88] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: implementations revisited. *Acta Informatica* 25:233–281 (1988).

[ST89] D. Sannella and A. Tarlecki. Toward formal development of ML programs: foundations and methodology. *Proc. 3rd Intl. Joint Conf. on Theory and Practice of Software Development*, Barcelona. Springer LNCS 352, 375–389 (1989).

[ST91] 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).

[ST92] 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).

[Sch87] O. Schoett. Data Abstraction and the Correctness of Modular Programming. Ph.D. thesis; report CST-42-87, Univ. of Edinburgh (1987).

[Tar92] A. Tarlecki. Modules for a model-oriented specification language: a proposal for MetaSoft. *Proc. 4th European Symposium on Programming*, Rennes. Springer LNCS 582, 451–472 (1992).

[Tof88] M. Tofte. Operational Semantics and Polymorphic Type Inference. Ph.D. thesis; report CST-52-88, Univ. of Edinburgh (1988).

[VG94] M. VanInwegen and E. Gunter. HOL-ML. *Higher Order Logic Theorem Proving and its Applications*. Springer LNCS 780 (1994).

[WRZ93] J. Wing, E. Rollins and A. Zaremski. Thoughts on a Larch/ML and a new application for LP. *Proc. 1st Intl. Workshop on Larch*, Dedham. Springer Workshops in Computing, 297–312 (1993).