Lecture Notes on Formal Program Development

Stefan Kahrs

Abstract

This document was originally produced as lecture notes for the MSc and PG course "Formal Program Development" early in 1997. After some initial general considerations on this subject the paper focusses on the way one can use Extended ML (EML) for formal program development, which features EML contains and why, and which pitfalls one has to avoid when formally developing ML programs. Usage, features, and pitfalls are all presented through examples.

1 Introduction

As the name suggests, the course "formal program development" is about the *formal development of programs*, here in the context of Standard ML (short: SML) programs and the specification language Extended ML (short: EML). It is assumed that the reader has a fair knowledge of the features of SML, for instance as attainable by participation in the Applicative Programming course.

The word "formal" in Computer Science or Mathematics has been used in various meanings. In order to avoid any prior misunderstandings I will briefly discuss these:

- "Formal" sometimes refers to a particular approach to foundations of mathematics, traditionally associated with David Hilbert. In certain respects, formalism overlaps with intuitionism, but both schools oppose the platonist idea of an ideal mathematical universe, in which men are still real men, women real women, little furry creatures from Alpha Centauri real little furry creatures from Alpha Centauri, and — most importantly — propositions are either true or false.
- Another possible meaning of "formal" is "completely formalised", that is: ready to be swallowed by your computer (program). The computer programs in question are typically proof checkers such as LEGO or PVS. Proof checkers address the problem that sufficiently large proofs tend to share one property with sufficiently large programs: they both almost always contain errors. Unfortunately, there is a dilemma: proof checkers are sufficiently large programs themselves.

- Finally, "formal" may just mean "not informal", using a formal language as opposed to English or Gaelic. This use of the word "formal" is common in the world of program specification, the catch-phrase one often finds is "formal methods". Thus the formalism applies to the language(s) specifications are written in, it does not necessarily (though possibly and recommendably) stretch to proofs (that a specification is satisfied) or the semantics of the specification language itself.

As you might have guessed we are concerned here with the last of the three possible meanings. EML is an extension of the SML programming language with certain specification features. It actually has a formal semantics [KST94, KST95, KST97] but we will stick to informal descriptions of its features, largely because the formal semantics is rather complicated and very few of these complications have genuine repercussions on EML's ordinary use. We will gloss over most of the difficult questions the formal semantics of EML has to answer — this course is about how to use the language, not how to define a specification language (semantics) of your own.

Regardless of how useful the EML language itself actually is, why is it that one would want to use a *formal language* at all? The *formality* of the language resolves ambiguities — it is easier to see what a specification actually specifies and what not, it helps to avoid hidden assumptions and to structure large specifications. Using a *language* as opposed to, say, a graphical notation (quite common in Software Engineering) has the advantage that the sentences of a formal language are more manœuvrable, and that specification manipulation becomes an activity similar to ordinary program manipulation.

Considering these other two words in the title, EML is indeed a language tailored for the task of *program* development. Other specification languages try to capture the whole of mathematics or at least most of it[1], EML — more specifically, its semantics — is not well-suited for such a task. EML is no attempt to set up an all-embracing mathematical formalism, its domain is the world of programs only.

The *development* of SML programs is to some extent already supported within SML, in particular through its module sublanguage. EML extends these capabilities by supporting incomplete programs and especially by extending the expressiveness of module interfaces. Syntactically, this is extremely simple, though rather powerful when used in connection with the modularisation features.

While this document is largely focussing on the features of EML and their relationship to the ideas of formal program development, I should also recommend [San89]. That report shows EML's features at work, without singling them out. Considering that it predates the formal definition of EML by several years, it should not come as a big surprise that the syntax has changed since then and that the formal semantics had to answer more questions than anticipated at the time. Still, the fundamental ideas remain unchanged.

[1] Of course, Gödel's incompleteness results and their construction mean that there is a limit to what a specific formalism can achieve.

2 Specification

The purpose of a specification is to express certain properties the corresponding program should have. There can be several reasons for writing a specification:

1. We may give a specification before a program is written, i.e. we set up the task of obtaining a program that meets the specification. This could be done by writing the program ourselves, by having somebody else to write it, or even (if we are very lucky) by some automatic tool that transforms our specification into a program.

2. We also may give it afterwards, as a guide to how to use and maintain the program, i.e. as a piece of documentation. We hardly ever want to read the source code (never mind the compiled machine code) of a program to understand its meaning.

3. Finally, we can think of it as an interface of a program module: we can replace the module by something else, provided it also satisfies the specification — the overall program should be unaffected by such a change. The ability to perform such manipulations is not only useful from a program maintenance point of view (plain speak: when we have to fix a bug), but it is also vital when such a module is machine-dependent or OS-dependent, as for example the code generator of a compiler.

What kind of specification language do we want, which properties of programs would we want to express? It is fair to say that there is no perfect answer to that question.

However, we almost always want to be able to specify its input/output behaviour, i.e. to express which outputs it produces when given certain inputs, or even that it produces any output at all. As we want to specify this in some finite way, we typically end up writing universally quantified equations, not dissimilar at all to writing a functional program in the first place. To allow greater flexibility and brevity we can go a little further and support all the quantifiers and connectives of first-order logic. This is indeed a (very crude) summary of all the features of EML's specification logic.

Are there other properties we might be interested in specifying? This is quite possible: we might demand a certain efficiency in terms of time and space consumption, we might ask (in a language with concurrency primitives) for fairness, the absence of deadlocks, etc. SML does not have concurrency primitives, and its language definition [MTH90, MTHM97] leaves the efficiency of SML programs unspecified.

The last point is important. SML's formal semantics tells us what the result of a computation will be, but it does not tell us anything about the time and space complexity this computation requires, neither in absolute nor relative terms. Although we may have a pretty good idea of how efficient a particular SML program is and although we may also be able to confirm that idea experimentally, we cannot really rely on the degree of efficiency of that program when we move it between different compilers. An SML compiler has the liberty to replace inefficient programs by efficient ones and vice versa[2], as long as their overall behaviour stays the same. This has a consequence for the design of a suitable specification language. Only compiler-invariant properties should be specifiable.

2.1 Incomplete Declarations

Traditionally, algebraic specification languages (see [EM85, Wir90]) have two parts: a signature giving the types of operations and constants, and a collection of axioms specifying their properties. The signature sets up the class of semantic domains we might consider and the axioms narrow it down as far as we like[3]. There are two major ways to interpret such algebraic specifications: either a *loose semantics*, in which every model satisfying all axioms has to be considered, or an *initial semantics* which favours the initial model. I shall not go into this in any detail, but it should be said that for several reasons an initial semantics is unsuitable for EML; this has something to do with its logic, its module system, and the built-in computational features.

One can mimick the algebraic specification style in EML, although the language is a bit more flexible and the distinction between axioms and signatures is less clear-cut. There are several ways to describe a signature (in the sense of Algebraic Specification) in EML, but all of them include one feature: unspecified code. Strictly speaking, it is a collection of four features, but the central idea is the same: we make a declaration but omit some part and write a `?` instead. For example:

```
val someint : int = ?
```

This is a declaration of an integer constant `someint` about which we do not know anything at the moment. One can write declarations like this in EML "programs" everywhere one can write ordinary declarations. Actually, this *is* an ordinary declaration, it just happens that the expression on the right-hand side of the declaration is extra-ordinary: the symbol `?` is itself an expression.

What is the meaning of a declaration like this? It does not have a specific operational meaning, for example if we try to evaluate `someint+1` we will not succeed. However, it has a collection of meanings, which in this case is the set of all environments in which the identifier `someint` is bound to some integer value. By writing further declarations (most notably *axioms* which we treat later) we can restrict this set of environments. For example:

```
fun factorial n = if n=0 then 1 else n*factorial(n-1)
val somefact = factorial someint
```

[2]Sometimes, elaborate program "optimisations" backfire and have the opposite effect.

[3]Provided, of course, that what we like is expressible in the specification formalism. For example, the Löwenheim-Skolem theorem limits what we can achieve with first-order logic.

These two declaration are complete, they do not leave us any space to be filled in later. Surprisingly, they also amount to an implicit restriction on our earlier value binding: `someint` cannot be negative. The reason is a bit subtle. Suppose `someint` were bound to a negative number. In the next step we successfully process the declaration of `factorial` and bind this identifier to a corresponding closure. Finally, we try to process the declaration of `somefact`, but this fails because the evaluation of its right-hand side fails to terminate for negative values of `someint`. As a consequence, the declaration of `somefact` fails to terminate and consequently the sequence of all three declarations. Therefore — and this is the crucial point — there is no environment we *obtain as the result* of processing this sequence of declarations in which `someint` is bound to a negative value.

The type of the expression `?` in general is $\forall\alpha.\alpha$ and for a particular occurrence it is the most general type we assume during type inference[4]. The general idea of incomplete programs is to defer their completion to a later stage in which the `?` is replaced by concrete text. Of course, this text has to be syntactically correct. Moreover we also require that it has the same type as the particular `?` occurrence — without that requirement we might make our program ill-typed. Practically this requirement forces us most of the time to write a type assertion restricting the type of `?` to something more concrete; if we omitted the type assertion from our example the only permitted replacements of `?` have type $\forall\alpha.\alpha$ and none of those terminates.

There is an alternative way of incompletely declaring values in EML — we can use (EML/SML) signatures and declare an incomplete structure.

```
signature SOMEINT =
    sig
        val someint : int
    end;
structure S:SOMEINT = ?
```

Here we have left the body of the structure `S` unspecified. The effect is essentially the same as in our previous example, except that (i) we have now a couple of additional identifiers in scope, (ii) we have to write `S.someint` instead, and (iii) we can only write this declaration at places where we are allowed to write top level declarations. The reason for the third restriction is that SML signatures are only allowed at top level. The declaration of the structure `S` alone is a bit more flexible, as it can be written anywhere structure declarations are possible.

This second use of `?` is restricted to the right-hand side of structure declarations; moreover, the signature constraint is then mandatory. This feature has been designed to accompany SML'90. For SML'97 with its slightly enriched syntax for structure expressions it would be more appropriate to allow an additional form of structure expression: ? :>*sigexp*. This would already cover the

[4] This does not make sense in the type system for Damas's toy language [DM82] which you may recall from the Applicative Programming course. However, there is a technical difference between EML/SML's type system and Damas's — roughly speaking, the SML type system is closer to the inference algorithm **W**, it does not allow generalisation of type variables at proper subexpressions and restricts instantiation to occurrences of identifiers.

third form of incomplete declarations in EML: `?` which is an empty functor body with mandatory output signature. We learn about functors a little later. Henceforth I will stick to EML as defined w.r.t. SML'90, tacitly omitting incompatible features as much as possible, with the occasional remark pointing at notable differences.

By permitting incomplete structures we implicitly also permit incomplete types, for instance:

```
signature TYPE =
    sig
        type sometype
    end;
structure S:TYPE = ?
```

In this example, `S.sometype` is some type we do not know yet. Because the type-checking phase is separate in EML (and SML) incomplete types are somewhat less flexible than incomplete values. In particular, we cannot restrict the concrete type of `S.sometype` by writing further declarations, we have to treat `S.sometype` for type-checking purposes as an opaque type. This is similar to the treatment of unspecified types in functor bodies, e.g. if `TYPE` were the input signature of some functor.

For technical reasons, related to EML's module semantics, the above way of providing an unspecified type is effectively useless. EML's module semantics allows us to use structures only through their signatures; as a consequence, it is impossible to have any values of type `S.sometype`. We shall come to that later. There is a more concise way of declaring unspecified types in EML which avoids this problem — we can use incomplete type declarations.

```
type sometype = ?
```

This is the fourth (and last) form of incomplete declarations. We can replace the right-hand side of a *type*-declaration (not a *datatype*-declaration!) by `?` and thus specify a yet unknown type. If our type declaration has any parameters then the eventual replacement of `?` can make use of the corresponding type variables, just as the eventual replacement of `?` within an expression can use all the declared/bound identifiers at that occurrence.

There is another form of incomplete type declarations which uses the keyword *eqtype* instead of *type* — in this case we assume that the specified type is an equality type, i.e. that we can compare values of this type with the predefined equality operation `=`. When we later replace `?` by a concrete type expression we have to make sure that this results indeed in an equality type. This is completely analogous to the `S.sometype` example if we replaced its *type* specification by an *eqtype* specification.

2.2 Axiom Declarations

We have already seen that it is possible to restrict the meaning of incomplete programs by adding declarations that may fail to terminate. This alone is a surprisingly powerful feature. For example, if we wanted our previous declarations to satisfy some property `P` we could just add the following declaration:

```
fun loop() = loop()
val test = if P then () else loop()
```

It is nice that we have this capability, but it is rather cryptic and I certainly would not recommend using it for this purpose. EML has a primitive for imposing a property on a program. For example, if we wanted to specify that our `someint` value is even we could write:

```
axiom someint mod 2 = 0
```

Technically, this is a declaration. Syntactically, it consists of the keyword *axiom* followed by an expression[5] of type `bool`. The semantics of an axiom declaration is either the singleton set containing the empty environment (if the axiom holds) or the empty set of environments (otherwise).

This gives us just what we want when we sequentially compose declarations. What is the meaning of dec_1 ; dec_2 in an environment E? We simply take the meaning of dec_1 in E; for each environment E' in the resulting set we take the meaning of dec_2 in $E + E'$ and for each E'' in this set we take $E' + E''$; this gives us a set of set of environments over which we then form the union.

Now suppose dec_2 is an axiom declaration. If for a particular choice of E' in the semantics of dec_1 the axiom holds then there is exactly one E'' in the semantics of dec_2 (the empty environment) and $E' + E''$ is equal to E'. Thus, if the axiom holds for the choice E' then E' is part of the overall result. On the other hand if it does not hold then the semantics of dec_2 is the empty set, it does not contain any E'' and so no $E'+E''$ can be formed, leaving E' out of the overall result. This is not quite the full story of the meaning of sequential composition of declarations, but it is for the language features we have encountered so far.

Syntactically an axiom declaration is considered a structure declaration, so we cannot use it within *let*-expressions and thus not within a function declaration.

2.3 EML Logic

The *axiom* feature in itself will not get us very far, without any means of quantification we can only make assertions but never state universal properties. For example, we might want to state of our `factorial` example that all numbers from 1 to n divide $n!$. This is possible in EML:

```
axiom forall n => forall m =>
        (m<=n andalso 1<=m implies (factorial n mod m = 0))
```

We can use this feature to claim certain properties of our already defined operations, or we can narrow down the possible choices for our yet-incomplete

[5]It's actually a bit more complicated than that, but for our purposes it is good enough.

programs. Technically, there is not really a difference between the two uses — in both cases it could happen that our axiom is inconsistent with what has been written before.

We can observe quite a few of the features and peculiarities of the logic of EML in this example already.

1. Logical expressions are just ordinary expressions of type `bool`.

2. The propositional connectives are the usual boolean connectives of SML, with the addition of *implies*. Similarly to SML's *andalso* and *orelse* the semantics of *implies* is directed. The second part of the implication is only evaluated once the first has been evaluated to `true`.

3. Universal quantification is expressed through a keyword *forall*. Syntactically, the text after *forall* is a so-called *match* which is the same thing that comes after the *fn* symbol to express a functional abstraction.

4. Quantifiers bind tighter than propositional connectives, following the tradition[6] of logic textbooks. Moreover, *andalso* binds tighter than *implies*. The parentheses around the equation are actually unnecessary and only included for better readability.

5. The type of a universally quantified variable is inferred from the context.

Some of these decisions in the language design of EML have important consequences. Using `bool` as the type for logical expressions means that we can mix the logic with our own functions that operate on boolean values — this is nice; we even can use logical connectives in ordinary expressions, e.g. to define predicates. On the other hand, we lose the Curry-Howard isomorphism (propositions as types, terms as proofs[Tho91]) — this is not nice; however, EML/SML allow us to write non-terminating functions which means that we lose the correspondence between proofs and terms anyway.

It was crucial in our example that *implies* is directed, i.e. that it evaluates the second part only if the first part evaluates to `true`. Why? Type inference tells us that the type of both `m` and `n` is `int`. So both variables range over all integer values. These include 0 and all negative numbers.

Suppose *implies* were an ordinary function that evaluates first both of its arguments. Consider the case that both `m` and `n` are 0. The premise of *implies* evaluates to `false` but the evaluation of the conclusion raises an exception. This packet is propagated and the result of the body for the forall. A similar though slightly different situation appears when `n` is negative. Again the premise is `false` but this time the evaluation of the conclusion does not terminate, because our version of `factorial` does not terminate when applied to negative numbers. This behaviour is again propagated, i.e. the evaluation of the strict implication does not terminate.

[6] I think that was a bad move in terms of language design on part of the logicians. Anyway, let's be grateful that it's not as bad as the language of mathematical analysis.

2.3.1 Quantifiers

So we suddenly face the problem of what a quantifier means if the evaluation of the expression in its body does not terminate for certain values in the range of the quantifier. A universal quantification in EML is `true` iff for all values over which we quantify the body of the *forall* evaluates to `true` — and not `false` and not a packet and it should terminate. Therefore in this particular case the entire axiom would not evaluate to `true`, meaning that our specification is not met by `factorial` and we end up with an empty set of environments.

One might now ask the question: what if `m` and `n` themselves are non-terminating expressions? This is excluded, because quantification does not range over *expressions* but over *values* of the corresponding type. So for `int` we get exactly what we would expect. For other types this problem is a lot less straightforward than we would hope and we come to that a little later.

It is very convenient that we can exploit type inference to find out the type of our quantified variables. However, we may not want to do that and explicitly specify the types of the quantified variables, just to make sure that the quantification ranges over the right type, or just as a matter of documentation. In other words, we may want to modify our example to:

```
axiom forall n:int => forall m:int =>
        (m<=n andalso 1<=m implies (factorial n mod m = 0))
```

This is already possible. Recall that the text after the keyword *forall* is a *match*, so in particular it can be of the form *pat*`=>`*exp*. A variable is a pattern, but so is a pattern with a type assertion like `n:int`. We could even write horrible things like *forall* `0=>true`. The meaning of this is the same as the meaning of *forall* `x:int => x=0` *orelse* (*raise* `Match`), which in turn is undefined.

Why is that? So far, we have only learned the conditions that make a universal quantification `true`. Naively, one might expect that the quantification is `false` in all other cases. Consider the following example though.

```
axiom forall x:int => loop()
axiom forall x:int => not(loop())
```

Both `loop()` and `not(loop())` exhibit the same behaviour — they fail to terminate. So both axioms would necessarily have the same semantics. For EML this was chosen to be "undefined". Roughly speaking, the reason for giving[7] it this semantics is that "undefined" is a fixpoint of `not`, as are non-termination and packets. The above axioms have the same effect as writing:

```
axiom loop():bool
```

This is an axiom that is never satisfied (as it does not evaluate to `true`) which semantically is treated the same as

[7] To be precise: the formal semantics of EML does not give the "undefined" meaning explicitly, it is left implicit by simply not defining it at all. This is analogous to non-terminating expressions in SML, which are exactly those for which the semantics fails to specify an evaluation result.

axiom `false`

which is never satisfied either.

We have not said anything yet about the situation in which the body of the quantification is `false` for some values and (possibly) fails to terminate etc. for others. If the body is `false` for at least one value over which the quantification ranges then the universal quantification is `false`, regardless what the evaluation of the body for other values would produce.

Existential quantification is defined dually to universal quantification in EML. Syntactically, it consists of the keyword *exists* followed by a *match*. Semantically, an existential quantification is `true` if it is `true` for some value in the quantification range, and `false` if it is `false` for all values in the quantification range.

I have been (deliberately) rather sloppy by using phrases like "evaluates to `true`" or "is `true`" in connection with the processing of expressions to establish their truth values. The EML semantics has coined a word to express that activity: "verificate". The reason for this assault on the English language was that the activity in question combines and intertwines ordinary verification of formulae as we know it from logic with ordinary evaluation[8] as expressed in the operational semantics.

2.3.2 Scoping and Type Abstraction

So far I have pretended that the question "what are the values of type τ" always has an easy answer. This is not quite true. For the precise definition there are several possible choices. We shall not go into too much detail, because some of the problems of determining the exact meaning only arise when we delve into the dark corners of the language.

First, the interpretation of polymorphic types and function types is not straightforward, but we delay their consideration for the moment. There are a couple of other problems. The following example has an obvious meaning.

```
datatype t = A | B
fun f (_ : t) = A
axiom exists y:t => f y <> y
```

Clearly, the axiom holds — under any reasonable semantics. But what happens if we modify the example a little bit:

```
datatype t = A | B
exception B
fun f (_ : t) = A
axiom exists y:t => f y <> y
```

[8]EML also has a notion of evaluation which coincides with SML's operational semantics and for which specification constructs such as quantifiers have no meaning at all; the purpose of having a separate notion of evaluation is to avoid impredicativity in the formal definition.

All we did was inserting a declaration and hiding the constructor `B`. Does the axiom still hold? This is tantamount to the question: "what does `y` range over"? If it only ranges over the "visible values" then the axiom is `false`, if it ranges over all values introduced by the original type definition then it remains `true`.

EML makes the latter choice, it views quantification semantically and scoping as an auxiliary syntactic construct. The rationale for this decision goes back to the origins of EML which lie in algebraic specifications rather than type theory. In any case, the example also indicates that it is not all that easy to make values inaccessible in the first place; it is certainly not recommended specification style to write specifications that depend on this particular aspect of the meaning of quantifiers.

Another problem arises when types are not defined directly but indirectly, using some method of type abstraction. Here is an example showing the difficulty.

```
signature S = sig eqtype u;  val B:u  end;
structure T:S = struct datatype u = B|C end;
axiom forall (x,y):T.u => x=y
structure T':S = T
```

To check the axiom we have to settle the question: has the type `T.u` two values or only one? The *implementing type* of `T.u` certainly has two values, but only one of them is accessible. This is not a scoping problem, the implementing type is internally regarded as incompatible with the implemented one, e.g. `T.u` and `T'.u` are treated as incompatible types in EML. For EML we have considered both choices of answering the question, they correspond to two different views of what a structure declaration is doing semantically.

The official EML semantics takes the view that semantically the implementing and implemented type are the same, and hence quantification can in a certain sense see through type abstraction barriers. In order to avoid this concept violating the general abstraction principle — which we will discuss at greater depth in the section on modularisation — EML employs a simple trick: at each abstraction barrier the implementing type is implicitly replaced by another one matching the same interface. This means that the axiom actually is satisfiable, simply by picking an implementation in which `T.u` has only one value. The effect of writing the axiom as an *axiom declaration* is to enforce this choice.

2.3.3 Polymorphism

Similar to its (and SML's) type system, EML does not support explicit quantification over types — syntactic quantification is restricted to values. However, the implicit quantification in its type system can also be used to express quantification over types in specifications.

We have already mentioned that the type of quantified variables is inferred, it does not have to be given explicitly. Type inference does not always find a unique type, it may be left with a choice at the end:

```
fun length [] = 0
  | length (x::xs) = 1+length xs
fun perms [] = [[]]
  | perms (xs:'a list) = ? : 'a list list
axiom forall xs => length (perms xs) = factorial (length xs)
axiom forall ys =>
    map length (perms ys) = map (fn _ => length ys)(perm ys)
```

Here we have an example of a partially defined polymorphic function `perms`. There are two axioms restricting the possible choices for `?`, but they do not fully determine the behaviour of `perms`. When we specify a universal property of a polymorphic function we would (almost always) like to formulate this property in a polymorphic manner, we do not want to be forced to provide a type instance to state the property.

The context of the first axiom restricts the range of the quantified variable `xs` to `'a list` (similarly for the second axiom and `ys`), but not any more than that. To determine the meaning of the axiom we have to settle the following question: What are the values of the quantification range?

If we consider the type `'a list` on its own, it has only one value — the empty list. We do not have values of type `'a` so we cannot form any non-empty lists over them. If that were the meaning of the above quantification then the axiom would not have restricted the `?` any further — simply because $0! = 1$ and 1 is the length of `[[]]`. This is not what we would intuitively expect to happen. Rather we would think that the axiom should hold for *all instances* of type `'a list`. This is indeed the meaning of this axiom in EML — unresolved type variables become universally quantified. Another way of looking at it is that the type of `xs` is actually $\forall' a.'$`a list` and not just `'a list`. Properly polymorphic values are not first-class citizens in SML, because they cannot be passed around as function parameters[9], therefore it would not be wise to quantify over them. Instead we should think of having separate quantifications for values and types, where the latter is notationally suppressed.

What about existential quantification? Consider the following example:

```
axiom exists x => forall y => perms [x,y] = perms [y,x]
```

This is not a property we would expect to hold for a permutation function on lists. But if we instantiate the variable `x` with `()` then the following situation arises: (i) the type of `x` is `unit`, (ii) so is the type of `y`, (iii) if our earlier axioms for `perm` are satisfied then both `perms [x,y]` and `perms [y,x]` are lists of length 2 (since $2! = 2$) containing only lists of length 2 of `unit` type. But that means that they must be equal! So, in a certain sense there is indeed an `x` that makes this axiom true.

On the other hand, if we choose the type instance `int` then it is impossible[10]

[9] If we pass them around as function parameters they lose their polymorphic status; this is different for *functor* parameters though.

[10] In order to prove this claim one can use general properties of polymorphic functions, see [Wad89].

to find such an `x`. Considering that the type quantification and instantiation is suppressed syntactically this may leave us with a rather uncomfortable feeling.

EML choses the following solution. Regardless of whether a variable is universally or existentially quantified, its free type variables are always implicitly universally quantified. Thus we have to think of the existential quantifier of the last example as being preceded by an invisible $\forall$`'a` : TYPE where we understand TYPE to be the "type" of all types. To satisfy a polymorphic existential quantifier we have to find witnesses for all its type instances. On the other hand, if the truth value of a quantification depends on the chosen type instances then it is left undefined in EML.

The example shows a couple of other peculiarities. First, I lied in the last paragraph. The type of `x` is not `'a`, it is `''a` — this is simply a consequence of using the good old equality operation within the axiom: the result type of `perm` is τ `list list` for some type τ which is the type of `x` as well. We used equality on the type τ `list list` and the type inference algorithm figures out that this is only possible if τ admits equality as well. Typically, one would like to avoid the restriction to equality types in such situations — after all we want to make a claim for *all* type instances. EML has a primitive concept for this purpose which we ignore for the moment. From a technical point of view, it is quite often sufficient to state a property of a generally polymorphic function for its equality type instances only [or even for a particular type only]; this is again a consequence of general properties of polymorphism [Wad89], but it is not considered good specification style to rely on these principles too heavily.

What is the type of the universally quantified variable `y`? Apparently, it is also `''a`. However, it must be the same `''a` as `x` has as its type, which means that `''a` is not a free type variable at the inner quantifier and thus is not implicitly quantified there. We cannot quantify over `''a` at this point because it is free in the context — this is completely analogous to type abstraction.

As explained, implicit quantification over types is always universal and there is no syntactically explicit quantification for types. So we cannot express existential quantification over types at all, can we? Somewhat surprisingly, we can.

```
axiom let eqtype t = ?
      in exists x:t => forall y => perms [x,y] = perms [y,x]
      end
```

The `?` by its very nature acts like an existentially quantified variable. For types we can only use it in the context of a type declaration, but type declarations can be made local to expressions. Consequently, we are indeed able to express existential quantification of types.

At this point I should perhaps admit my lack of imagination — I find it hard to contemplate any non-pathological use of this feature. What the heck, pathological examples are fun!

2.3.4 Equality

Suppose we wanted to specify that the first permutation in the list of permutations produced by `perms` is always the original list. So we may want to write something like this:

axiom forall `xs => hd (perms xs) = xs`

As we have already seen, this implicitly imposes the equality-type constraint on `xs`. We cannot directly *deduce* from it the corresponding behaviour on non-equality types, although we can *induce* it by appealing to the polymorphism of `perms`. Induction is a rather heavy proof tool and whenever we can we would like to use the more operational deduction instead.

EML has a primitive for specificational equality, written `==`.

axiom forall `xs => hd (perms xs) == xs`

The meaning of `==` is more or less the same as `=`, but there are some differences:

- The computational equality `=` demands equality types for its arguments, the specificational equality `==` makes no such requirements, except that the types have to be the same.
- Since `t=u` is just a function call, it is strict in both arguments; actually there is only one argument which is a pair. One the other hand, specificational equality is non-strict and mirrors definitional equality in the following sense: `t==u` not only holds if the evaluation of `t` and `u` produces the same values, but also if they raise the same exceptions or if both fail to terminate.
- On the other hand, `t==u` demands that neither `t` nor `u` make any use of "specification constructs" such as quantifiers; they have to be SML-evaluatable, otherwise a special exception is raised. This does not concern `t=u`.

The first point raises the question what `==` "does" when confronted with a non-equality type. Specificational equality is a specification construct, i.e. ordinary *evaluation* of `t==u` is undefined regardless of the type involved. The *verification* of `t==u` requires that both expressions are indistinguishable by program contexts. For functions, this normally means that they are pointwise equal — the proviso "normally" has to be there because of the presence of SML-style exceptions. To be precise, the mentioned program contexts that can be used for distinguishing expressions can refer to hidden constructors and breach type-abstraction barriers — this is completely analogous to the way quantification range is defined.

2.3.5 Abnormal behaviour

The evaluation of an SML/EML expression does not always return a value, it may fail to terminate, it may produce an exception. Bearing in mind the semantics of specificational equality, we are already able to express the absence of such abnormal behaviour:

> *axiom forall* `x=>` *exists* `y=> perms x == y`

This axiom is claiming that `perms` is a total function, simply because `y` only ranges over values. Fortunately, we do not have to write such axioms very often, because the totality requirement is typically implicit in ordinary axioms. For example, our first axiom for `perms` which stated that the length of the result of a `perms` call is the `factorial` of the length of the argument is of exactly that nature.

EML provides a shorthand to express the absence of abnormal behaviour: *exp proper* is an expression which is `true` iff the evaluation of *exp* results in a value.

> *axiom forall* `x => perms x` *proper*

means exactly the same as the previous axiom. EML also has primitives to specify termination and exception-raising behaviour:

> *axiom forall* `n => (n>=0` *implies* `factorial n` *terminates*`)`
> *axiom forall* `(m,n) => ((m mod n` *raises* `Div)` *implies* `n=0)`

The EML syntax makes the above expressions embarrassingly[11] close to their meaning in the English language.

Similarly to `==`, both *proper* and *terminates* specify behaviour of *evaluation*, not of *verification*. They are themselves specification constructs and thus cannot be used within ordinary evaluations. This has the usual fundamental reason of preventing the diagonalisation of a halting-problem-decider.

2.3.6 Higher types

EML has function types. But what is a function? What are quantifiers over function types exactly ranging over? Given two functions `f` and `g`, under what circumstances are `f` and `g` equal? These questions are addressed (indirectly) in the EML semantics [KST94] and (directly) in its gentle introduction [KST95, KST97].

However, in a certain sense the proper answer to these questions is "Mu!", i.e. you should not ask these questions in the first place. In other words, you should avoid writing specifications that rely too heavily on the particular way these questions are answered by the EML semantics.

When we validate a universally quantified formula we do this w.r.t. a particular model. More specifically, our model has to provide an interpretation for function spaces. For interpreting `A->B` there are various choices: we could take all functions from `A` to `B`, all continuous functions, all computable functions, or all ML definable functions. One of the problems is that there is no fully developed recursion theory on higher types; for first-order functions we know that

[11] Embarassingly, because we are following the tradition of COBOL here.

the ML-definable functions are exactly the partial recursive ones, but for higher types we do not have any evidence to support a generalised Church-Turing thesis.

SML defines in its formal semantics what the values of the operational semantics are, there is no direct association between values and types. Values can be described as free terms over some first-order signature. This includes the function values most of which are given as so-called "closures". A closure is essentially a pair consisting of the syntax of the function and an environment determining the meaning of the free variables.

EML is very similar in its value domain (for verification), and we shall not discuss the subtle differences here. Since we have (and need) a typed quantification, we have to somehow split the domain of values into subdomains associated with each type. The idea is simple: a value of type τ is a value that arises from the successful evaluation of an expression of type τ; and an expression of type τ is an expression that passed the SML type-check with type τ in a particular environment. The environment in question gives access to all constructors and the implementation of types.

Thus the domain of function types is given by a more general principle that applies to all types. It is similar with specificational equality: two expressions are considered equal (of type τ) if they are indistinguishable by any program context. The program contexts in question have to be well-typed and have a hole of type τ — they are formed in the same environment we use for the evaluation of expressions that determine the set of values.

While I am generally trying to avoid going into the subtleties of questions such as "what are the values of a function type?", one subtlety is worth mentioning. The quantification ranges over "SML values", not EML ones. This means that (i) we indeed require that an *evaluation* of the representing expression succeeds, not just a verification and (ii) that the resulting value has no references to any specification constructs, such as universal quantifiers hidden in a closure. In particular, when we quantify over the type `int->bool` we quantify over ordinary SML functions of that type, not over EML predicates.

I could give examples the successful verification of which depends on these choices, but I am not going to — I do not want to tempt you into following this road.

3 Modularisation

One of the most fundamental ideas underlining EML is to incorporate specifications in the module system. This means (i) to localise specifications to modules and (ii) to restrict the use of a module to the features and properties guaranteed by its specification.

Traditionally we use modules for programming-in-the-large. Often when a program has grown too large to be thought of as a unit we split it into more or less coherent pieces. Ideally, the programming language should support this in some way, i.e. we would like to use the various pieces without being forced to merge the source code. Even better would be some way of compiling these pieces separately, as this speeds up program maintenance, installation etc., not to mention that the capability of separate compilation is evidence that our splitting-up was not arbitrary, but along certain logical lines.

We may do this for our own peace of mind, but it is even more significant within a team working on a common project. There are typical pitfalls that frequently occur in such a setting which — if possible — our module system should address. A typical problem in many programming languages is the complete lack of scoping: a variable is visible anywhere. Especially if you work in a team this leaves you with the problem of which variable names to use and which to avoid. The project manager may end up dividing the name space of variables forcing each member of his team only to use variable names starting with a specific sequence of characters[12].

In other words, we would like a discipline in the language that restricts the class of globally accessible identifiers to some fixed (for each project) well-defined set, and at the same time supports the use of local names in the various modules. Whether two names in different scopes are equal or not should have no significance.

Concerning EML/SML, this specific problem is taken care of by various means, for instance *local* declarations, but also imposing a signature on a structure has such an effect. Some readers may be offended by this claim, but I maintain that any decent programming language provides at least some support in that direction.

Of course, this is not enough. Scoping only helps us to keep out of each other's way, it does not help us to interact. At some point, the various pieces we have split our program into have to communicate and use each other. Especially in the presence of separate compilation, such interactions between different program units are a potential source of errors. We need a discipline to guide the communication between structures, a notion of *secure interface*. In SML/EML interfaces are called *signatures*.

So far we have only used signatures to curtail structures. They have another use in SML, as the interface of a parameterised module, a so-called *functor*.

[12]This is not a joke, this happens in real life.

3.1 Structures with Interface

Before we come to functors, we have a more thorough look at what SML/EML signatures offer w.r.t. structures. A structure declaration has usually the following form:

structure `A:S` = *struct* `....` *end*

where `A` is the structure identifier we introduce, `S` is a signature expression (most often a signature identifier) and the four dots are the implementing structure.

The signature expression `S` specifies the contents of `A`. The most important elementary specifications possible in a signature are various forms of value and type specifications. We specify a value always together with its type, or rather: its type scheme, because structure components can be polymorphic. However, the type of a specified value may itself be (partially) specified rather than concrete. When we specify types, we actually specify type functions, i.e. n-ary functions from types to types.

There are various forms of such specifications, giving away various chunks of information about the concrete objects associated with the specifications. The reason to provide such a variety is two-fold: (i) if the signature is the interface of a single structure then — as a matter of good software design — we give away as little information as possible, allowing us more freedom in the actual implementation; (ii) if the signature is the interface of a functor (we come to that later) then restricting the interface as much as possible increases the applicability and thus re-usability of the functor.

For example, here are several possibilities for specifying types:

type `t1`
eqtype `t2`
datatype `t3 = C1` *of* `t2 | C2` *of* `int`
datatype `t4 = E` *of* `t3 | F` *of* `t1`

If we find these specifications in a signature then we know hardly anything about `t1` — its just a type constructor of arity 0, i.e. any type. We know a little bit more about `t2`: it must be an equality type. This has two consequences: the (any) concrete type associated with `t2` must admit equality, and we can exploit this fact, i.e. the predefined functions `=` and `<>` are available of type `t2*t2->bool`.

We know a lot about `t3`: it must be a *datatype* with these two constructors of the given types; moreover, these are *the only* constructors of type `t3`; finally, the constructors themselves are also specified by this declaration and they are specified *as constructors* which means that consequently we can use them for pattern matching. Of course, `t3` ultimately depends on `t2`, similarly as `t4` depends on both `t1` and `t2`. The difference between these two specifications is that `t3` is inferred[13] to be an equality type, while `t4` is not. In the concrete

[13]Inferring the equality attribute is a surprisingly delicate issue which has changed in the SML'97 revision [MTHM97]. See also [GGM93] for a more thorough look at this problem.

structure we associate with this specification we may have an equality type realising `t1` — and thus making `t4` an equality type as well, but the signature does not give access to that information.

The extra information one could possibly reveal for a value is its constructor status. For constructors of datatypes, giving away their status is only possible in the already mentioned form, we cannot reveal a constructor without revealing the others. One pragmatic reason behind this restriction is to enable the compiler to warn the user that a particular *match* does not cover all the values of its type. The same restriction does not apply to exception constructors, there are specifications of (single) exception constructors. The consequences of providing that extra bit of information are completely analogous to the *eqtype* specification: customers of the signature can exploit the information, suppliers have to provide a concrete structure meeting all these criteria.

Going back to the structure declaration `structure A:S = ...`, the components of the structure `A` are determined by its signature `S`. In particular, we have only access to those identifiers specified in `S`. Other identifiers declared in the implementing structure remain hidden. In fact, the curtailment of structures goes a bit further than that:

```
signature IL = sig val il:int list end;
structure A:IL = struct val il = [] end
```

In the example, we have implemented a specified `int list` by a polymorphic `'a list`. This is fine, we can specialise a polymorphic value to the required type; but whenever we do so we forget its polymorphic origin. This means that `A.il` can only be used as an integer list, not as a list of any other type.

We can slightly modify the example, abstracting away the component type.

```
signature IL' =
    sig type a;
        val il: a list
    end;
structure A':IL' =
    struct type a=int;
           val il=[]
    end
```

Again, if we use `A'.il` we cannot exploit its polymorphic origin. But do we know that it is an `int list`? We should not, it is not information available in the interface of `A'`. Indeed, we do not have access to this information in EML, EML's structure-signature matching is *opaque*. In SML though, this property shines through (all together: boo! hiss!), its matching is *transparent*. SML'97 provides both forms of matching, where opaque matching is expressed using `:>` instead of `:`. The reasons for the somewhat counter-intuitive meaning of structure/signature matching in SML'90 are largely pragmatic; in connection with functors, opaque matching easily leads to the need for sharing constraints (we learn about them later), a rather unpopular feature amongst ML programmers.

Transparent matching is unsuitable for separate compilation, almost by definition. Moscow ML only supports a small sublanguage of SML's module system, and it does indeed use the module system for separate compilation — it should come as no surprise that it only has opaque matching. The idea of opaque matching and its use for separate compilation goes at least back 20 years to the imperative language Modula-2.

Considering that we can only access structure components that are specified in the corresponding signature it makes perfect sense to allow incomplete structure declarations. An incomplete structure declaration provides the signature of a structure without giving an implementation. It should be possible to compile an incomplete structure and other structures depending on it — only the linker should require a completion.

```
structure B:IL = ?
```

By the opaqueness principle all we can see of a structure is what the interface gives us access to. Thus we can see exactly the same thing of `A` and `B`. Whenever we can use `A` we could use `B` instead and vice versa.

In a certain sense this is unsatisfactory. Surely, there are different integer lists, and this difference should not be messed up just by packaging them up as structures?

The problem we face is that the type of a value is a helpful but generally insufficient piece of information when we use a module. We may need to know further properties of a value. Suppose we wanted to implement integer sets by sorted integer lists without repetitions, based on some given partial order. Concerning the types, a partial order on integers is just a function of type `int*int->bool`. So we might write something like this:

```
signature POint = sig val le : int*int->bool end;
structure A:POint = ?
abstype intset = Emb of int list
with
        val empty = Emb []
        local
            fun adds(x,[]) = [x]
            |   adds(x,s as y::z) =
                if A.le(x,y) then
                    if A.le(y,x) then s
                    else x::s
                else y::adds(x,z)
        in  fun addset (x,Emb xs) = Emb(adds(x,xs))
        end
        fun memberset(x,Emb []) = false
        |   memberset(x,Emb(y::z)) =
                if A.le(x,y) then A.le(y,x)
                else memberset(x,Emb z)
end
```

This is perhaps not the most elegant way to attack the problem (some of its other deficiencies we shall discuss later), but it looks perfectly reasonable. However, it only works properly if the function `A.le` has certain properties. For example, if `A.le` is given by the constant `false` function then `adds` adds elements at the end of the list and `memberset` always returns `false`.

Still, we do not have to know what `A.le` is *exactly* in order to guarantee the correctness[14] of the integer set implementation. Any partial order[15] would do nicely.

What we need is a specification device for "exporting" those properties of a structure that are required further on; we have to make these properties known just as we make (some of) the types of structure components known. Such a device should maintain the general opaqueness principle. The general principle behind this feature is analogous to the way we treated the *eqtype* information: it provides information to the customer and demands it from the supplier.

EML provides such a feature: axioms in signatures. Additional to the exported values and types we export some of their properties. For our example of partial orders it could look like this:

```
signature POint =
    sig
        val le: int*int -> bool
        axiom forall x => le(x,x)
        axiom forall (x,y,z) =>
            (le(x,y) andalso le(y,z) implies le(x,z))
        axiom forall (x,y) =>
            (le(x,y) andalso le(y,x) implies x=y)
    end
```

If we use this version of `POint` instead of the previous one then our implementation of integer sets behaves in the desired fashion: the only replacements we allow for `?` have to satisfy the three axioms.

We may not only use signatures with axioms to specify incomplete structures, we can also use them as proof obligations for structures where some (or all) of its components are fully specified. For instance, we could write:

```
structure Divides:POint =
    struct
        fun le(x,y) = exists z:int=>x*z=y
    end
```

...with the idea to use divisibility as the partial order. Of course, we have to ensure that our implementation indeed satisfies the specification, in this case:

[14] This is an informal notion of correctness as we had no formal requirement that our abstract type really did implement finite sets. As a friend once told me: "I never write incorrect programs. When they display 'segmentation fault' on the screen then they are clearly supposed to do so."

[15] If `A.le` were just a preorder we also get sets, but we could not in general represent *all* finite integer sets.

that it provides a partial order. By using a quantifier in the body of a function we were pushing our luck a little bit.

There is nothing wrong with the idea, but the devil is in the detail. Unfortunately and rather surprisingly, the example does not quite work, because SML's integers have limited precision. In particular, for numbers `x` close to the maximum value the evaluation of `x*z` gives an overflow exception for any `z` different from 0, 1 or -1. Recalling the semantics of `exists` this means that `le(m-1,m)` is undefined for sufficiently large `m`. As a consequence, the antisymmetry axiom of our new `POint` signature is not satisfied: we choose `x=m-1` and `y=m` and observe that the body of the universal quantifier (of the antisymmetry law) is undefined for that choice, leaving the universal quantifier as a whole undefined, which in turn is not good enough to establish satisfaction of the axiom.

The purpose of this erroneous example is to highlight a general problem we have here: what is the meaning of a structure declaration in which the signature check goes wrong? The problem is in fact more general than that, because the implementing structure might not be uniquely determined:

```
structure Somepo:POint =
    struct
        fun le(x:int,y:int) = ?:bool
    end
```

Here there are many ways to complete the structure `Somepo`, but most will not satisfy the axioms of `POint`.

EML structure/signature matching makes the following demand: any structure in the semantics of the structure expression in question must satisfy the axioms of the signature! If that is the case then the meaning of the matching is the set of all structures matching the signature (and not just the ones arising from the verification of the structure expression itself). Otherwise, the structure/signature matching has no meaning, making it behave like a non-terminating expression.

Notice that the last principle differs crucially from giving it the empty set of structures as its meaning. As a consequence, structure/signature matching is the place in EML where actual verification (in the ordinary sense) is required.

In other words, neither of our two `POint` examples goes through, they are both judged to be meaningless, or rather: both fail to be judged to have a meaning. We can easily repair our divisibility example though:

```
structure Divides:POint =
    struct
        fun le(x,y) = exists z:int=>(x*z=y handle _ => false)
    end
```

The motivation for this modification is simple: if the evaluation of `x*z` overflows then the absolute value of this expression (on unbounded integers) is larger than any representable one and hence larger than `y`. This time `le` is a total function and it is easy to implement.

As mentioned earlier, the meaning of this structure declaration in the EML verification semantics is to bind the structure identifier `Divides` to *an arbitrary* structure satisfying the axioms of `POint`. The reason for this superficially weird choice is the opaqueness principle: you get what you specify and nothing more.

Another remark on the failed examples: the very existence of structure declarations without a meaning has an impact on the meaning of sequential composition of declarations. We have not only to cope with sets of environments, but also have to incorporate the possibility that we sometimes encounter a meaningless structure declaration. What makes things worse is that the failure to have a meaning in general depends on the choice we have for earlier declarations. Here is an example:

```
val some'le: int*int->bool = ?
structure Dep:POint = struct val le=some'le end
```

The question whether the second declaration successfully *verificates* solely depends on whether we choose in the first declaration a partial order. EML's verification semantics propagates verification failure: a composition of two declarations verificates if all the choices we can make for the first declaration lead to a successful verification of the second. "Successful" here just means: has any meaning, including the empty set of structures. Therefore, the example fails, because for some choices we have for `some'le` the following structure declaration fails to verificate.

While EML provides *axiom* specifications as an additional feature to export information about values, it does not provide any specific features (that is: additional to SML) to export information about types and structures. SML already provides rather powerful features in this respect. A very important one is the notion of *sharing*. A *sharing* constraint is a particular kind of specification whose only purpose is to specify certain types or structures as being equal. We shall see the importance of this feature (which has undergone major surgery in the SML'97 revision) in connection with functors in the next section.

One can explain the need for such a feature with an analogy concerning first-order terms. A signature is like an open term, the unspecified entities are its variables. A structure (without incomplete declarations) is like a ground term, it has no unspecified components, no variables. A structure matches a signature if there is a *realisation* mapping the specified items of the signature to concrete items in the structure; similarly, a ground term t matches an open term p iff there is a substitution σ such that $\sigma(p) = t$. A signature is more general than another one if any structure matching the latter also matches the former. The same principle on first-order terms gives rise to the subsumption preorder on open terms, alternatively given as: $p \leq q \iff \exists\sigma.\,\sigma(p) = q$. Along the same lines we can pre-order signatures. On first-order terms we have that $F(x,y)$ subsumes $F(z,z)$ which in turn subsumes $F(H(C),H(C))$, but neither subsumption is reversible. The subsumption $F(x,y) < F(z,z)$ on first order terms corresponds to a *sharing* constraint in SML signatures.

3.2 Functors

Recall the example of integer sets as sorted lists without repetitions we encountered earlier. In the example, we defined sets as ordered lists w.r.t. an externally given partial order. It was a bit clumsy and awkward in several respects, most notably:

- The incomplete *structure* `A` was rather more like a parameter than anything incomplete, since any such `A` gives rise to integer sets of some description.
- We defined sets monomorphically for integers, although nothing in the example really uses the type — replacing the three occurrences of `int` by any other monomorphic type would have worked just as well.
- But the polymorphic version would have eluded us: if we replace the three occurrences of `int` by `'a` then the example becomes effectively useless: the component `A.le` would *need to* be polymorphic and any such function would not depend on its arguments.

We could remedy the third imperfection by declaring an unknown type and using this unknown type consistently in place of `int`. However, we still would only have one set type available at any one time, and the first imperfection would be even worse, since we now have two parameter-like beasts on the loose, scattered around in the source code.

What we need is a notion of parameterised structure; a structure parameterised by another structure. Many languages with a module system do not have a notion of parameterised module, and most of those who do have a less elaborate module system than SML. For some reason, a parameterised module is called a *functor* in SML[16], although they have very little in common with functors in category theory and zilch with functors in Prolog. A functor parameter is a structure matching the input interface of a functor — an input interface is simply a signature.

A signature can be seen as the type of a structure, something we may call a *kind*. A functor is (roughly, there is a hitch) a function mapping structures of one kind to structures of another.

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

This signature encapsulates what we perceived to be the parameters in the `set` examples: we want a parameter type without saying what it is plus a partial order working on *that particular type* (and not a polymorphic partial order).

[16] Moscow ML does not support functors, but they are part of the official definition. The same goes for nested structures as in the `INSERT` example.

For the moment we shall ignore axioms since they do not have any particular significance concerning the points we need to discuss.

```
signature INSERT =
    sig
        structure A:PO;
        val insert: A.t * A.t list -> A.t list
    end;
signature SET =
    sig
        structure A:PO;
        type set
        val empty: set
        val addset: A.t * set -> set
        val memberset: A.t * set -> bool
    end
```

Given any partial order, we can create an insertion function on lists over that type. In order to express what an insertion function is we need to mention its type, making the above definition of the signature `INSERT` a bit awkward. Of course, we have not actually specified partial orders and insertion here, because the signatures do not contain any axioms. Specifying `SET` was quite similar; notice that there is no connection between the specified type `set` and the component type of the partial order other than provided by `addset` and `memberset`.

The language designers of SML/EML could have avoided the slight awkwardness of both `INSERT` and `SET` by providing a feature of parameterised signatures, but they chose not to do so. One can express parameterisation of signatures differently, though it has to be said that this is one of the less elegant features of the language. Some of the modifications of signatures in the language revision SML'97 move the language a bit closer to such a feature of parameterised signatures.

A functor mapping a structure of kind `PO` to a structure of kind `INSERT` could be written as follows:

```
functor Insert(S:PO):INSERT =
    struct
        structure A=S;
        fun insert(x,[]) = [x]
        |   insert(x,y::ys) =
                if A.le(x,y) then x::y::ys
                else y::insert(x,ys)
    end;
```

There is no type inference for functors in EML; SML allows us to drop the output signature though and infers it, making everything as transparent as possible. In EML output signatures are compulsory and employ opaque matching only. There is no fundamental semantical reason for this syntactic requirement in EML, it merely enforces a certain programming discipline.

> Remark: a full-blown generalisation of type inference from functions to functors is impossible, the type inference problem for functors is undecidable. This is related to both the undecidability of second-order unification [Gol81] — which we are hit by when we guess what type function a type constructor is bound to, and the undecidability of semi-unification [KTU93] — which comes our way when we are guessing the type scheme of a value. End of remark.

In fact, in the example SML would have inferred a different output signature from the one we wrote down. It is intriguing to see why, as it explains the need for certain features of SML's module language. Generalising the opaqueness principle from structures to functors, we should be allowed to replace the functor body by another one if it still fits the old interface. For instance:

```
functor Insert'(S:PO):INSERT =
    struct
        structure A:PO=
            struct
                datatype t = C of t;
                fun le p = true
            end;
            ....
    end;
```

The functor `Insert'` differs from `Insert` only in the structure `A`. Since both functors have the same interface, we can use one wherever we can use the other (in EML; in SML'97 this is true if we use `:>` instead of `:` before the output signature). In particular, this means that they produce the same types.

By comparing the two examples it becomes clear that the functor interface does not tell us what the type in the returned structure is. To retain type soundness the type-system makes a worst case assumption (`Insert'` comes mighty close to the worst case) which is that the type `A.t` in the output signature is freshly created and incompatible with any other type. At this point it should be said that functors are not quite like functions, because if a functor creates a fresh type then it does so *on each call.* Functors are not extensional[17].

The creation of a fresh type is not what we wanted. The functors `Insert` and `Insert'` create fresh types, because their interface fails to express any dependency between parameter and output. EML/SML provide a feature for this: *sharing* constraints.

```
functor Insert(S:PO):
    sig include INSERT; sharing A=S end = ....
```

Here, the sharing constraint states that the structure `A` in the output is the same as the structure `S` from the input. The functor `Insert'` would not meet this stricter structure interface.

[17] It is possible to come up with a module system with extensional functors for transparent matching, but there are complications. For instance, without restricting other parts of the language we would lose the decidability of type inference.

Unfortunately we have met here a feature of SML that has changed in the SML'97 revision of the language [MTHM97]. The described revised functor interface was fine in SML'90 and EML, but it is rejected by SML'97. SML'97 only has type sharing, although a restricted form of structure sharing is still possible as a derived form. The reason the above functor interface is rejected in SML'97 is that the implicitly shared type `S.t` comes from outside the signature in which it is shared.

We can still express a corresponding interface in SML'97, but it requires the use of a newly added feature. The SML'97 interface of our example is as follows:

```
functor Insert(S:PO) :> INSERT where type A.t=S.t = ...
```

In some sense, the new syntax is more appealing, as it is closer to viewing `A.t` as a parameter of `INSERT` rather than a fixed component, but the incompatibility of versions is nevertheless deplorable.

There is a general trick how one can avoid most of the time the needed *sharing* constraints between input and output of a *functor*. The trick is to use *implicit* rather than *explicit* sharing. For example:

```
functor Insert2(S:PO) :
    sig val insert:S.t*S.t list -> S.t list end =
    ....
```

The functor `Insert2` has a different kind than `Insert`, because its result does not contain a copy of the input structure `S`. Still, the generated `insert` function does indeed operate on the desired lists: the structure identifier `S` is visible when we analyse the result signature. We could not have bound this output signature to a global signature identifier, because we can only define it w.r.t. `S.t`: this is a situation in which parameterised signatures would be useful.

Analogous to incomplete structures are incomplete functors in EML: we simply write a `?` for the functor body. Another analogy to structures is verification: since signatures can contain axioms, writing a functor may leave us with some proof obligations. The principle should be obvious: a functor is correct if it maps any structure matching the input structure to a structure matching the output structure, where "matching" includes the verification of the axioms in question. Considering that the functor body itself may contain incomplete declarations, this is not quite the full story: we have to verify that *any* possible functor result matches the output signature.

In practice, people tend to apply a functor exactly once, making it effectively a device to work with incomplete structures. The reason is simply that the more a functor contains the more specific is its application, and it is not a very attractive prospect to write a functor that contains very little. For the latter you need to compose lots of functors to get anywhere, the absence of higher-order functors (which means that functors themselves can be structure components, functor parameters, etc.) from the language definition[18] does not help either. Still, with many little functors there is a better chance of reusing code.

[18] SML of New Jersey supports them though.

3.3 Verification

What happens to the properties of structures when they are passed through signature interfaces? Do they remain the same? Can we lose information? Can we gain information? Can properties change from `true` to `false`, or vice versa?

The general principles answering these questions (to a certain extent, anyway) are:

- Transparent matching requires the satisfaction of the axioms, but otherwise it leaves the structure unchanged.
- Opaque matching also requires satisfaction of axioms, but the structure is afterwards replaced by something arbitrary matching the signature.
- We can lose information.
- We can gain information, though only in a very limited manner.
- Properties never change, unless we generally view everything upto observational equivalence, most notably the meaning of quantifiers and specificational equality.

We have to elaborate a little bit on these points.

First, EML seems not to support any transparent matching — so the first point does not apply, does it? It does. EML has implicit transparent matching at two places: (i) for matching a structure against the argument signature of a functor, and (ii) for the matching of substructures. Let us forget about (ii) (it is similar anyway): when we call our *functor* `Insert` and apply it to a partial order, then we do not want this particular partial order to be replaced by an arbitrary one when we deal with the body of the functor.

The second point is quite simply a direct consequence of our general opaqueness principle: any opaque structure binding provides us with an opportunity to replace the implementing structure by something completely different, as long as it still matches the interface. This clearly means that we lose information at opaque structure bindings, or rather that we lose more information than we already do through transparent matching, like which components of a structure are available.

In a weak sense, we gain information through both kinds of matching: by specialising polymorphism, certain formulae which were undefined *before* may have a proper meaning *after* the structure binding. Take the `perms` example and the axiom:

axiom forall `(x,y) => perms [x,y] == perms [y,x]`

This axiom is not satisfied for a polymorphic permutation function `perms`. But if the signature interface chooses to specialise the argument type of `perms` to `unit list` then the axiom is true, under both forms of matching.

However, we do not gain information in the strong way we might think we do. Since quantification and specificational equality are unaffected by scoping changes, transparent matching has no impact on them either. But the same is true for opaque matching, since both logical features look beyond abstraction barriers; one can say that they operate on the underlying algebra. The nice consequence of this is that properties we have once established as true can be passed through abstraction barriers without any need to re-verify them.

On the downside, this principle of EML deprives us of "true" abstract datatypes and is hence rather counter-intuitive. Example:

```
signature SET2 =
    sig
        include SET;
        val union: set*set->set
        axiom forall (x,y) => union(x,y)==union(y,x)
    end;
```

This looks like a perfectly reasonable requirement for sets. Moreover, we may think that we can implement these sets by our familiar sorted lists, e.g. in the following way:

```
functor SET2(X:PO): sig include SET2 sharing A=X end =
struct
    structure A=X;
    type set = X.t list;
    val empty = []
    fun addset(x,[]) = [x]
     | addset(x,s as y::z) =
                if A.le(x,y) then
                    if A.le(y,x) then s
                    else x::s
                else y::addset(x,z)
    fun memberset(x,[]) = false
     | memberset(x,y::z) =
                if A.le(x,y) then A.le(y,x)
                else memberset(x,z)
    fun union([],x) = x
     | union(x::xs,z) = addset(x,union(xs,z))
end;
```

Alas, there is a catch: the functor is not correct according to the official EML semantics. The problem is that `union` is not really commutative. It is commutative on all the sets we can form through the signature (sorted lists without repetitions), but it is not on the underlying implementation (all lists, sorted or not, repetitions or not) and the quantifier ranges over exactly that.

This is not what one would normally expect. An alternative semantics of EML could treat this functor as correct, simply by preventing quantifiers and specificational equality from piercing abstraction barriers. Such an alternative semantics would indeed make signature matching change the meaning of formulae, it is a major change to the meaning of logical connectives. On the plus side, we would be closer to "real" abstract data types; on the minus side, proofs become significantly more difficult as almost any formula has to be re-checked once it passes through an opaque interface.

There is a semantic reason why the EML semantics was defined that way: it is viewing a structure/signature matching as a "reduct" operation on algebras. A semantics dealing properly with observational equivalence would need to be based on a more sophisticated operation on algebras, also involving a reachability constraint.

A pragmatic reason to design the EML logic this way was that EML is supposed to support the development of *SML programs*. At least SML'90 does not offer opaque matching and therefore not the security required to justify an observational reading of EML logic. In particular, an SML program could use the `union` function created by the `SET2` functor on non-sorted lists, exhibiting its non-commutative behaviour on the full domain:

```
structure S:PO =
    struct
        type t = bool;
        fun le(x,y) = not x orelse y
    end;
structure SS = SET2(S);
val s1 = SS.empty
val s2 = [true,false]
val a = SS.memberset(false,SS.union(s1,s2))
val b = SS.memberset(false,SS.union(s2,s1))
```

The above sequence of declarations is illegal in EML, because we used the list `s2` as a `set`, violating opaqueness. However, it is legal in SML'90 and we find to our disgust that `a` is bound to `false` and `b` to `true`.

After the 1997 revision of the SML definition the above example is significantly less compelling. We have to change the syntax of the `SET2` *functor* anyway, because the *sharing* constraint in its output signature is now illegal, for the already mentioned reason that it shares a specified type with a type in the basis. When we turn the *sharing* constraint into a *where*-clause then the example also goes through in SML'97. However, if we also change the `:` sign before the output signature to `:>` then the matching is opaque and the example is rejected.

4 Refinement

So far we have seen many language features describing what we can write in an EML program, or perhaps we should say "specigram". An EML program is a static entity though, once it is there it is there, no need for any development.

So how do we go about developing a program? Where do we start, where do we stop, what are the steps we can take in between?

There is never a perfect answer to that, everybody has their own style when it comes to writing and developing a program. Therefore, all EML provides is a framework in which certain steps are guaranteed to work, certain others involve proof obligations.

4.1 Adding something new

We can always add a new declaration to whatever we already have, but we have to make sure of a few details.

Clearly, we should not hide anything, or at least not in such a way that we change any bindings. Furthermore, we have to make sure that our new declaration is sound w.r.t. the rest of the program, which means we have to check that its own axioms are satisfied. We should also make sure that the declaration does not raise an exception:

```
val x = let exception a
        in 1
        end handle _ => 2
```

In the example, `x` is bound to 1. If we insert the declaration `val z=raise a` after the exception declaration then — although `z` is never used anywhere — the binding of `x` changes to 2. For similar though slightly less compelling reasons we have to make sure that our declaration terminates: if it does not then the termination behaviour of the surrounding block of declarations is affected and EML provides the metapredicates *terminates* and *proper* to observe such changes.

However, we will not need to re-check the rest of our program, because the EML logic is stable under such extensions.

While it is alright to add a declaration, it is much more problematic to extend an existing one. In particular, extending a *datatype* by another constructor affects the meaning of the associated quantifiers; it also affects the meaning of functions operating on such a type: functions that used to be indistinguishable may suddenly be observably different. This problem does not extend to exception constructors, since the type `exn` is always open to the addition of new constructors anyway.

Adding an axiom specification to a signature induces proof obligations: any structure that supposedly matches the signature in question has to be checked against the new axiom. This also applies if the signature is (contained in the) output signature of a functor, but no such check is needed if only the input signature is affected. Notice that the customers of that structure do not need a re-check: they are supposed to work happily with anything the old interface provided.

Somewhat surprisingly, it is always possible to add an *axiom* declaration. If the axiom is inconsistent with the rest of the program then the meaning of the new program is just the empty set of environments: but it does have a meaning!

The dual to adding a declaration is removing it. As long as the rest of the program does not use the declared identifiers we may think that it is always safe to remove a declaration, but this is not quite the case.

- In order to remove a value declaration we have to make sure that its verification returns a value, i.e. that it terminates without raising an exception.
- In order to remove an *axiom* declaration, we have to check that the axiom expression is always satisfied.

These conditions sound a bit strange, almost paradoxical. We would perhaps expect something like that when we introduce a declaration, but not when we remove it. First, the reason for requiring non-exception raising behaviour is rather easy to explain, because it is dual to our earlier example:

```
val x = let exception a;  val z=raise a
        in 1
        end handle _ => 2
```

In the example, `x` is bound to 2, but if we remove the (unused) declaration of `z` then `x` is bound to 1.

The reason for requiring termination of value declarations is twofold: on the hand we have problems dual to the introduction of non-termination, i.e. the metapredicates are affected; more importantly though, our requirement is similar to the requirement than an axiom is satisfied before it can be removed. Axiom declarations restrict the choices we may have for preceding declarations. The presence of such a restriction may be needed to ensure the verification of another structure declaration. Example:

```
fun pred (x:int) = ? : bool;
val x:int = ?
axiom pred x;
signature P = sig val y: int; axiom pred y end;
structure S:P = struct val y = x end
```

This sequence of declarations verificates quite happily. The axiom makes sure that whatever choice we make for `pred` and `x`, `pred x` evaluates to `true`. This is needed for the verification of the final structure declaration. If we remove the axiom then we have a wider choice for `pred` and `x` — too wide for the verification of `S`. Notice that the axiom is already *sometimes* satisfied, meaning for some of the choices we have earlier. However, it is not *always* satisfied and this is the property we need to safely remove an *axiom* declaration.

4.2 Replacing question marks

The most obvious development step is the replacement of a `?` by concrete code. Under which conditions is this correct? Of course, it has to pass the type-check, but apart from that?

If the incomplete declaration in question is either a structure or functor declaration then we have to check the axioms of the accompanying signature. That is all. The reason why this is sufficient is that we have opaque structures only, which keeps proof obligations strictly local. There is a general trick to ensure that this (almost) always works. If we start with

> *signature* `S = ` *sig* ... *axiom* `P` *end*;
> *structure* `A:S = ?`

then we can push the proof obligation inside our structure as follows:

> *signature* `S = ` *sig* ... *axiom* `P` *end*;
> *structure* `A:S = ` *struct* .... *axiom* `P` *end*;

By simply repeating the axiom from the signature in the structure we make sure that our development step is correct. In particular the body of the new structure can contain incomplete declarations itself. Effectively we have delayed the proof to the point when we want to discard the axiom in the structure body.

The above claim was made with a proviso. There are actually two provisos: (i) if we viewed everything up to observational equivalence (the official EML semantics does not) then this trick is too simplistic: module interfaces affect the meaning of formulae modulo observational equivalence; (ii) in the (fortunately rare) case that the implemented operations are *more polymorphic* than their specifications and if the axiom `P` uses implicit typing, then copying `P` literally into the body is not fully satisfactory: it could be that `P` is satisfied for the type instance the signature demands but not in the general case.

Replacing a `?` in a type declaration is always fine. Often we would like to replace such an incomplete *type* declaration by a fresh *datatype* declaration. This is possible, though one should keep in mind that such a development is made up of several more elementary steps including the addition of a declaration: whenever we introduce a datatype we also introduce its constructors.

If we replace a `?` value expression by a more concrete expression, we have to make sure that it does not raise an exception, and also that its evaluation terminates. The absence of exceptions is needed because of nasty examples like the one in the previous section; moreover, the semantics of EML *assumes* that the `?` replacement does not raise an exception. This does not apply for `?` protected by functional abstractions, which can be arbitrarily nasty.

Notice that replacing a value `?` does not involve checking any axioms. We do not need that, because we simply picked one of the choices for `?` all of which would make the rest of the program pass the verification. In particular, it trivially passes the verification if a following *axiom* (declaration) is *not* satisfied. This may seem paradoxical again, but is perfectly in line with the observation made earlier that *discarding* an axiom forces us to prove it while we can *add* it at our pleasure.

Of course, we would like to avoid *undiscardable* axioms, because they prevent us from completing a program development.

4.3 Divide et impera

When we replace a question mark we may choose something rather concrete as its replacement, i.e. we may make a *coding step*. Or we may not. In order to make any progress it would be fine to choose something that is more concrete, without going all the way towards a running implementation. In other words: we may choose a top-down strategy of program development and make a *decomposition step*.

Most typically, this applies for incomplete structure and functor declarations. Given an incomplete structure declaration, we can attempt to decompose it into a functor and another structure, both for the time being incomplete as well. We can proceed analogously for incomplete functor bodies.

By doing such a decomposition, we have implicitly introduced several proof obligations. The general picture is the following: We start with

```
structure A:S = ?
```

and decompose it into a functor and another structure:

```
structure B:S' = ? ;
functor G(X:S'):S = ? ;
structure A:S = G(B)
```

Notice that we also need to define a new signature `S'` to make sense of it all. Is such a decomposition always correct? It is indeed, but then we have not made much progress. The reason the decomposition is always correct is that we used signatures that exactly fitted the requirements.

It remains correct if we allow sharing constraints in the output signature of `G`. Adding sharing constraints is always correct w.r.t. to EML's standard semantics; the story is somewhat different if specifications can be satisfied up to observational equivalence. More sharing, more observers, more things to quantify over, less observational equivalence.

Instead of using signatures that exactly fitted the requirements, we could have used ones that only entail them. For example, we could use a different signature `S''` as input for `G` but then we would introduce a proof obligation: any structure matching `S'` has also to match `S''`. A similar modification is also possible concerning the connection of the output signature of `G` and `S`. Moreover, we could also allow sharing between the input and output of `G`.

There are two typical problems that one faces when adjusting the input and output interface of a functor. These are underspecification and overspecification. We have overspecified the functor if it is impossible to produce the required output from the available input. We need to know more about the input. For example, if the input just gives us two abstract types `a` and `b` and the required output contains a function of type `a->b` that always terminates then we cannot deliver. There is no functor meeting this requirement. As a whole, the functor is overspecified because its input is underspecified; similarly, it could be overspecified, because its output is as well.

During a program refinement, one typically goes through a series of over- and underspecifications: getting a specification right is just as difficult as getting the program right. The specification part of the language is more expressive than the programming part, allowing specifications to be more concise than programs. The added expressiveness is a two-edged sword though — it is not only easier to get it right, it is also easier to get it wrong.

5 A Case Study

Let us see at an example of how one can go on to develop an EML program. Suppose we wanted a little program that implements ML pattern matching.

The first problem we face is: how do we specify it, what actually *is* ML pattern matching? We can (and are likely to) be very conservative in our first attempt. We start with terms and patterns, so we are given:

```
signature Term =
    sig
        type term;
        type pattern;
    end;
```

and we want a structure defining substitutions, substitution application and matching a term against a pattern. Something like this:

```
signature Match =
    sig
        structure T: Term;
        type substitution;
        val apply: substitution -> T.pattern -> T.term
        val match: T.term * T.pattern -> substitution option
        axiom forall s as (t,pat) =>
            (case match s of
                SOME r => apply r pat == t
                | NONE => not (exists r' => apply r' pat == t))
    end

functor M(X:Term) : sig include Match sharing X=T end = ?
```

Can this possibly work? Without any information how patterns and terms are connected we have little chance of creating a meaningful substitution application. No chance at all, actually. It is an awful thought, but perhaps we get away with a meaningless notion of substitution application? After all, we have not asked very much about `apply`. One possible solution to the specified problem is the following:

```
functor M(X:Term) : sig include Match sharing X=T end =
    struct
        structure T=X;
        type substitution = unit
        fun apply p = apply p
        fun match p = NONE
    end;
```

It indeed matches our specification. Of course, this functor is nowhere near implementing ML pattern matching, our specification was too general, we *underspecified* our problem. An obvious modification is to require that `apply` is total, in other words we add the following *axiom* specification to the signature `Match`:

```
axiom forall (p,s) => apply s p proper
```

With this additional claim we have outlawed the earlier attempt for `M`, since it did not provide a total `apply` function. Now we face another problem: there is no (fully-coded) `M` that satisfies the specification, we have *overspecified* our problem.

Still, there was nothing wrong with the additional axiom for `Match`, we just need to refine the signature `Term` and provide some connection between patterns and terms.

```
signature Term =
    sig
        type term;
        type pattern;
        type symbol;
        val tform: symbol * term list -> term;
        val pform: symbol * pattern list -> pattern
    end;
```

That looks a bit more like it, but is it all we need? Clearly not. With the extended version of `Term` we are able to *specify* the homomorphic property of `apply`, which should be included in the requirements of `Match`.

```
axiom forall (c,ps,s) =>
        apply s (T.pform(c,ps)) == T.tform(c,map(apply s)ps)
```

On the other hand, the extension is of no use whatsoever when it comes to *implementing* `apply` or `match`. For the implementation of `apply` we need the converse of `pform`, we need to be able to decompose patterns. Not every pattern is a constructor pattern though, we also want variables. Thus our next attempt at signature `Term` is:

```
datatype ('a,'b) Either = Left of 'a | Right of 'b
signature Term =
    sig
        type term;
        type pattern;
        type symbol;
        type variable;
        val tform: symbol * term list -> term;
        val pform: symbol * pattern list -> pattern;
        val pembed: variable -> pattern
```

```
    val pdestruct: pattern ->
            (variable,symbol * pattern list) Either
    axiom forall t => tform t proper
    axiom forall cps => pdestruct(pform cps) == Right cps
    axiom forall v => pdestruct (pembed v) == Left v
    axiom forall p => case pdestruct p of
        Left v => pembed v == p
        | Right cps => pform cps == p
end;
```

The axioms provide a connection between composition and decomposition of patterns. They also mean that all specified operations are total — for the pattern operations this is implicit from their axioms: the constructors `Left` and `Right` are total, hence the right-hand sides of both equations are always defined and thus the left-hand sides have to be defined as well, which — for any particular choice of arguments — requires the properness of `pembed` and `pform`. Similarly, `pdestruct` is total, because *case* is strict in its discriminating expression.

Do we have enough information around to implement our functor `M`? Perhaps we have, perhaps we have not. We have been looking for additional structure necessary to implement substitution application. We have not looked at matching yet. Perhaps we should make a decomposition step then, settling substitution application now and leaving matching for later.

```
signature Substitute =
sig
    structure T: Term
    type substitution
    val apply: substitution -> T.pattern -> T.term
    axiom forall (s,p) => apply s p proper
    axiom forall (c,ps,s) =>
        apply s (T.pform(c,ps)) == T.tform(c,map(apply s)ps)
end;
functor Sub(X:Term):sig include Substitute sharing X=T end = ?
functor Match(X:Substitute):
    sig include Match
        sharing T=X.T
        sharing type substitution = X.substitution
    end = ?
functor M(X:Term):
    sig include Match
        sharing X=T
    end = Match(Sub(X))
```

Can we implement this lot then? Our attention has focussed so far on substitution application, so we are in with a shout for `Sub`:

```
functor Sub(X:Term):sig include Substitute sharing X=T end =
    struct
        structure T=X;
        type substitution = variable -> T.term
        fun apply s p =
            case T.pdestruct p of
                Left v => s v
                | Right (c,ps) => T.tform(c,map(apply s) ps)
    end
```

Looks cool! Is it correct? No! Why not? We required that `apply` is a total function and the above is not, simply because the type `substitution` contains partial functions as well. Can we restrict this type to total functions only? No, EML does not provide such a feature.

What can we do? We could regard the totality requirement for `apply` as an overspecification, but it is not easy to refine. Instead, we could use association lists instead of functions:

```
fun assoc [] x = NONE
 | assoc ((v,a)::xs) x = if v=x then SOME a else assoc xs x
fun finmap [] = true
 | finmap ((v,a)::xs) = finmap xs andalso
        case assoc xs v of NONE => true
         |  _ => false
```

However, this also means that the domain of the function we want to model is an equality type. This means that we have to refine the signature `Term` again and make `variable` an equality type. After this modification we can write:

```
functor Sub(X:Term):sig include Substitute sharing X=T end =
    struct
        structure T=X;
        type substitution = (variable,T.term) list
        fun apply s p =
            case T.pdestruct p of
                Left v => (case assoc s v of
                    NONE => ?
                    | SOME a => a)
                | Right (c,ps) => T.tform(c,map(apply s) ps)
    end
```

That was not entirely successful either, note the remaining `?`. If the pattern contains a variable that is not accounted for in our substitution then we face the problem what to do with it. In the context of ML pattern matching this situation should never arise, but here we are stuck again, because we do not have this information available here.

One way to proceed is to incorporate in our specification that this situation cannot arise. For example, we introduce something like a substitution domain, relativising the totality of `apply` w.r.t. such a domain and demanding that `match` generates a substitution with a particular domain.

In the context of a writing an SML implementation that way would be quite appropriate, but for our purposes it is a bit over the top. We could simply go for ordinary pattern matching of first-order terms and only need that variables can be embedded into terms as well. In other words, we make another modification of the signature `Terms` and include a function `tembed:variable -> term`. This allows us to replace the question mark in `Sub` by `T.tembed v`.

By putting all of this together we obtain a candidate for an implementation of substitution application:

```
signature Term =
   sig
      type term
      type pattern
      type symbol
      eqtype variable
      val tform: symbol * term list -> term
      val pform: symbol * pattern list -> pattern
      val pembed: variable -> pattern
      val tembed: variable -> term
      val pdestruct: pattern ->
             (variable,symbol * pattern list) Either
      axiom forall t => tform t proper
      axiom forall v => tembed v proper
      axiom forall cps => pdestruct(pform cps) == Right cps
      axiom forall v => pdestruct (pembed v) == Left v
      axiom forall p => case pdestruct p of
          Left v => pembed v == p
          | Right cps => pform cps == p
   end;
functor Sub(X:Term):sig include Substitute sharing X=T end =
   struct
       structure T=X;
       type substitution = (T.variable,T.term) list
       fun apply s p =
           case T.pdestruct p of
               Left v => (case assoc s v of
                   NONE => T.tembed v
                   | SOME a => a)
               | Right (c,ps) => T.tform(c,map(apply s) ps)
   end
```

Now we have coded the *functor* `Sub`, or have we? There is indeed a problem, it concerns the termination behaviour of `equal`. Let us try to find an input for `Sub` that breaks its specification, in other words: a pathological structure implementing terms.

```
structure PathologicalCounterexample: Term =
    struct
        type term=unit and symbol=unit and variable=unit;
        datatype pattern = V | C of unit -> pattern list
        fun tform _ = () and pform(_,xs) = C(fn () => xs)
        fun tembed _ = () and pembed _ = V
        fun pdestruct V = Left ()
         | pdestruct (C f) = Right ((),f())
        val littlebastard =
           let fun recu () = [C recu]
           in C recu
           end
    end;
```

This weird structure indeed matches our signature `Term`. The offending problem concerning the correctness of our `Sub` implementation is caused by the presence of such strange patterns as the `littlebastard`. If we `apply` a substitution to `littlebastard` then our implementation will not terminate: each decomposition of `littlebastard` gives us a new copy of the little fellow.

Do we care about such pathological monsters? We should do. What is the alternative? Where is the borderline between pathological nonsense and sensible examples? We know where this borderline is: it is given by our signatures. Anything matching the signature is a sensible example, anything else is not of our concern.

In other words: patterns are still underspecified. What is missing is an induction principle for patterns. There are several ways of creating one. The simplest is to *specify* `pattern` as a *datatype*, i.e. to reveal its implementation in the signature. For our purposes this is the very last resort. An alternative would be to add an induction principle through a functional — in the same way `foldr` works for lists; however, this does not really solve the core of the problem, because we would still need to make assertions concerning the termination of the functional. Instead, we can try to specify the induction principle more directly:

```
datatype nat = Zero | Succ of nat
infix >>
fun Zero >> _ = false
 | Succ _ >> Zero = true
 | Succ x >> Succ y = x >> y
signature NatInduction =
    sig
       type any
       val measure: any -> nat
       axiom forall x => measure x proper
    end;
```

Why have I not used `int` instead of `nat`? Two reasons: (i) there are negative integers, messing up the induction principle with the additional requirement that the `measure` should return positive values. More importantly, (ii) there is a maximum value of type `int`, making the type in a strict sense ineligible as an induction vehicle for any type with (conceptually) infinitely many values.

A second question one may ask: why have we not stated a more general induction principle, after all we can quantify over functions? We can quantify over functions, but we cannot quantify over predicates. EML logic remains first-order, a general induction principle requires second-order logic.

Now we can express that the type `pattern` has an induction principle, and that the measure of a pattern decreases when we decompose it.

```
signature Term =
    sig
        type term  and pattern
        type symbol
        eqtype variable
        val tform: symbol * term list -> term
        val pform: symbol * pattern list -> pattern
        val pembed: variable -> pattern
        val tembed: variable -> term
        val pdestruct: pattern ->
                (variable,symbol * pattern list) Either
        axiom forall t => tform t proper
        axiom forall v => tembed v proper
        axiom forall cps => pdestruct(pform cps) == Right cps
        axiom forall v => pdestruct (pembed v) == Left v
        axiom forall p => case pdestruct p of
            Left v => pembed v == p
            | Right cps => pform cps == p;
        structure Ind: sig include NatInduction
                          sharing type any=pattern
                         end
        axiom forall (s,xs,ys,p) =>
            Ind.measure(pform(s,xs @ [p] @ ys)) >>
                Ind.measure p
    end;
```

Our pathological counter-example is now ruled out, because there is no corresponding `measure` function that satisfies the last axiom of our new version of `Term`. We do not have to change anything in our implementation of `Sub`. Since the measure decreases each time we decompose a pattern, we know that our `apply` function terminates.

To complete the implementation of `M` there remains the task to code `Match`. On the basis of the given data this is impossible, because we do not have any method to decompose a term. We also would need to be able to compare symbols, i.e. the symbol we get from decomposing a pattern with the one we get from decomposing a term. So we either have to modify `Term` yet again, or we need a new signature `Term2` for this particular purpose.

If we think about it a little, it becomes clear that we need essentially the same structure for terms and patterns. Additional to that we need to be able to compare symbols in order to compare the head of a pattern with the head of a term. Writing everything twice is something we would definitely want to avoid. Instead, we extract the common structure:

```
signature SimpleTerm =
    sig
      type term;
      eqtype symbol;
      eqtype variable;
      val form: symbol * term list -> term;
      val embed: variable -> term
      val destruct: term ->
              (variable,symbol * term list) Either
      axiom forall cts => destruct(form cts) == Right cts
      axiom forall v => destruct (embed v) == Left v
      axiom forall t => destruct t proper
      structure Ind: sig include NatInduction
                       sharing type any=term
                     end
      axiom forall (s,xs,ys,p) =>
          Ind.measure(form(s,xs @ [p] @ ys)) >>
              Ind.measure p
      axiom forall t => case destruct t of
          Left v => embed v == t
          | Right s => form s == t
    end;
```

This is the structure common to both patterns and terms. To form a new version of `Term` we just take two copies of it, i.e. two structures both matching this signature. However, we should not forget to share the types of symbols and variables, because we want the *same* variables and the *same* symbols in both patterns and terms.

```
signature Term2 =
    sig
      structure P: SimpleTerm and T: SimpleTerm;
      sharing type P.symbol=T.symbol
      sharing type P.variable=T.variable
    end;
functor Adjust(X:Term2):
    sig include Term;
      sharing type term=X.T.term
      sharing type pattern=X.P.term
      sharing type symbol=X.T.symbol
```

```
        sharing type variable=X.T.variable
    end = ?
```

Writing the implementation for `Adjust` should be obvious, it is essentially a bunch of renamings, although some values are not exported. The purpose of writing `Adjust` is that we can continue using our other functor `Sub`. The overall picture is now as follows. First, we need a new input signature for the `Match` functor:

```
signature MatchInput =
    sig
        structure X: Substitute;
        structure Y: Term2;
        sharing type Y.P.term=X.T.pattern;
        sharing type Y.T.term=X.T.term;
        sharing type Y.T.symbol=X.T.symbol;
        sharing type Y.T.variable=X.T.variable
    end;
functor Match(M: MatchInput):
    sig include Match
        sharing T=M.X.T
        sharing type substitution = M.X.substitution
    end = ?
```

The signature `Substitute` does not contain enough information about our terms, which is the reason why we need to keep a copy of the `Term2` structure we start with — the `Sub` functor forgets too much information, so we can only partly use its output. Considering the mountain of sharing equations we have to produce to persuade EML that we still work with the same sort of terms, this is not the likely way we would proceed in practice. It is easier to modify the interface and implementation of `Sub` accordingly. For our purposes though, it is somewhat enlightening to push some of the language features to their ugly limit.

Of course, we still have to code `Match`. Based on whatever implementation we provide for `Match` we can implement our overall goal, the functor `M` as follows:

```
functor M(Z:Term2):
    sig include Match
         sharing type Z.T.term=T.term
         sharing type Z.P.term=T.pattern
         sharing type Z.T.symbol=T.symbol
         sharing type Z.T.variable=T.variable
    end =
    let structure A:sig include MatchInput; sharing Y=Z  end =
                struct structure X=Sub(Adjust(Z));
                       structure Y=Z
                end
```

```
in Match(A)
end
```

As the example shows, EML/SML support *let*-expressions on the structure level. We needed this feature here, because every structure in EML (not SML) is required to be equipped with an interface. In particular, we could not have applied the functor `Match` to the anonymous structure `A` is bound to. There is a conceptual reason for this restriction, but I somehow fail to remember what it was exactly.

After this fight with sharing constraints (it should be clear by now why they are not very popular amongst programmers), we are still left with the task of coding `Match`.

```
functor Match(M: MatchInput):
    sig include Match
        sharing T=M.X.T
        sharing type substitution = M.X.substitution
    end =
    struct
        open M.X;
        fun match(t,p) =
            case M.X.pdestruct p  of
                Left v => ?
            |   Right (s,ps) =>
                (case M.Y.T.destruct t  of
                    Left v' => NONE
                |   Right (s',ts) =>
                    if s=s'  then matchlist(ts,ps)
                    else NONE)
        and matchlist([],[]) = ?
         | matchlist([],p::ps) = NONE
         | matchlist(t::ts,[]) = NONE
         | matchlist(t::ts,p::ps) =
              glue(match(t,p),matchlist(ts,ps))
        and glue x = ? : substitution option
    end;
```

As we can see, there are still three question marks left. We still have to solve the problems of (i) which substitution matches a term against a variable pattern, (ii) which substitution matches an empty list of terms against an empty list of patterns, and (iii) how can we glue two substitutions together into one. We can see how far we are yet away from our goal: we have absolutely no way yet to form a substitution and without the type assertion in the body of `glue` type inference would suggest type `'a option` as result type throughout.

We are hampered by the security of the module system again, because not having access to the internal structures of substitutions prevents us from solving the problems right away. The `Substitute` signature is program-internal and we can put as much information into it as we desire. As a matter of good programming style we would like to put as little into it as possible. But we have to address the mentioned problems.

For addressing these problems it is insufficient just to provide functions of the required types, even if we implement them in the right way. The problem is that their implementation is protected behind a signature interface and we can only rely on what the signature promises. We could proceed along these lines, but we would need rather elaborate axioms to secure the meaning we need. Instead, the signature `Substitute` could export a more general substituiton-building operation and leave the task to form it correctly to its customer.

```
signature Substitute =
sig
    structure T: Term
    type substitution
    val apply: substitution -> T.pattern -> T.term
    val sform: (T.variable*T.term) list -> substitution
    axiom forall (s,p) => apply s p proper
    axiom forall (c,ps,s) =>
        apply s (T.pform(c,ps)) == T.tform(c,map(apply s)ps)
    axiom forall (v,xs) =>
        (finmap xs implies
            apply(sform xs)(T.pembed v) ==
            (case assoc xs v of
                NONE => T.tembed v
                | SOME t => t))
end;
```

This specification only states the behaviour of `sform` in case its argument is a finite map, i.e. if it does not associate a variable more than once. Otherwise, it could do anything. This design decision makes it rather easy to implement `sform` in the needed modification of `Sub`:

```
functor Sub(X:Term):sig include Substitute sharing X=T end =
    struct
        structure T=X;
        type substitution = (T.variable*T.term) list
        fun apply s p =
            case T.pdestruct p of
                Left v => (case assoc s v of
                    NONE => T.tembed v
                    | SOME a => a)
                | Right (c,ps) => T.tform(c,map(apply s) ps)
        fun sform xs = xs
    end
```

Clearly, this implementation also satisfies a stricter axiom in which we drop the `finmap` requirement. By having restricted the safety guarantee of `sform` we have hidden an implementation detail of `sform`. Any customer of the signature `Substitute` can only reliably use the function `sform` in connection with proper finite maps. It would not be of much help anyway to reveal the missing detail as we shall see in a moment.

First, here is a little auxiliary function for working with values of type `'a option`.

```
infixr >>=
fun NONE >>= f = NONE
 | (SOME x) >>= f = f x
```

This is a very useful little function when it comes to composing functions that return optional results. So useful that the language Haskell opted for predefining it and supporting it with a special syntax.

```
functor Match(M: MatchInput):
    sig include Match
        sharing T=M.X.T
        sharing type substitution = M.X.substitution
    end =
    struct
        open M.X;
        fun match'(t,p) =
             case M.Y.P.destruct p of
                  Left v => SOME [(v,t)]
             |   Right (s,ps) =>
                  (case M.Y.T.destruct t of
                       Left v' => NONE
                  |   Right (s',ts) =>
                       if s=s' then matchlist(ts,ps)
                       else NONE)
        and matchlist([],[]) = SOME []
         | matchlist([],p::ps) = NONE
         | matchlist(t::ts,[]) = NONE
         | matchlist(t::ts,p::ps) =
               match'(t,p) >>= (fn xs =>
               matchlist(ts,ps) >>= (fn ys => SOME (xs@ys)))
        fun match x = match' x >>= (fn xs =>
             if finmap xs then SOME (sform xs)
                          else NONE)
    end;
```

Is this correct? Almost, but not quite. Our implementation correctly treats the "`SOME`" case of the original specification of `match`, but there remains a problem with the "`NONE`" case.

We simply do not know if the pattern we started with was linear or not, i.e. whether it contained repeated variables. The totality requirement for `pform` means that there *are* non-linear patterns and our specification of `Substitute` explains that substitution application is defined on non-linear patterns as well. On the other hand, `match` always returns `NONE` when given a non-linear pattern. This is not always correct; in particular, we could well match the same variable more than once against the same term.

Superficially, this looks like a self-inflicted injury: we exported `sform` as a partial map and now we suffer from its partiality. Looking a bit closer we may detect though that a total `sform` would not have helped us here: if the association list associates *different* terms with the same variable then the matching result should be `NONE`.

There is a choice: we can leave `Match` unchanged but make the appropriate modifications to rule out non-linear patterns; or, we modify `Match` to cope with the offending case. The required modification of `match` is not too difficult to find:

```
fun squeeze [] = SOME []
 |  squeeze ((v,t)::xs) =
        squeeze xs >>= (fn ys => case assoc xs v of
            NONE => SOME ((v,t)::ys)
            | SOME u => if equal(t,u) then SOME ys else NONE)
fun match x = match' x >>=
              (fn xs => squeeze xs >>= (SOME o sform))
```

Unfortunately, we have cheated a little bit: we do not have a function `equal` operating on terms. An easy way out would be to require `term` to be an equality type and then using the predefined = function. This is possible, but not very nice for a number of reasons. In particular, we already have enough information available to write an equality function ourselves:

```
functor Equal(X:SimpleTerm):
sig
   val equal:X.term*X.term->bool
   axiom forall (t,u) =>(t==u)=equal(t,u)
end =
struct
   fun equal(t,u) = case X.destruct t of
             Left v =>
             (case X.destruct u of
                 Left v' => v=v' | _ => false)
             | Right(s,ts) =>
               (case X.destruct u of Left _ => false |
                   Right(s',us)=> s=s' andalso eql(ts,us))
   and eql([],[]) = true
    |  eql(x::xs,y::ys) = equal(x,y) andalso eql(xs,ys)
    |  eql _ = false
end
```

This is obviously correct, or is it? Could we possibly have missed anything?

Possibly yes, but we did not. Our specification of `SimpleTerm` — more specifically, the axiom with the *case* expression — ensured that `destruct` is injective, hence it reflects equality. Without this our functor `Equal` would not be correct. Another property we have to secure is the termination of `equal`, simply because `==` is total. Here we can use again (now for the third time) that we have an induction principle for terms.

Where we do we stand now? We have finished our development. The final versions of `Sub` and `M` we have already seen, the final version of `Match` is the following.

```
functor Match(M: MatchInput):
    sig include Match
        sharing T=M.X.T
        sharing type substitution = M.X.substitution
    end =
    struct
        open M.X;
        fun match'(t,p) =
            case M.Y.P.destruct p of
                Left v => SOME [(v,t)]
            |   Right (s,ps) =>
                (case M.Y.T.destruct t of
                    Left v' => NONE
                |   Right (s',ts) =>
                    if s=s' then matchlist(ts,ps)
                    else NONE)
        and matchlist([],[]) = SOME []
         | matchlist([],p::ps) = NONE
         | matchlist(t::ts,[]) = NONE
         | matchlist(t::ts,p::ps) =
              match'(t,p) >>= (fn xs =>
              matchlist(ts,ps) >>= (fn ys => SOME (xs@ys)));
        structure E = Equal(M.Y.T);
        fun squeeze [] = SOME []
         | squeeze ((v,t)::xs) =
                squeeze xs >>= (fn ys => case assoc ys v of
                    NONE => SOME ((v,t)::ys)
                    | SOME u => if E.equal(t,u) then SOME ys
                                else NONE)
        fun match x = match' x >>=
              (fn xs => squeeze xs >>= (SOME o sform))
    end;
```

6 Conclusion

Is there a moral after this, just as at the end of each episode of our average American TV sitcom?

Our case study makes formal program development appear a more complicated task than programming, even than programming with correctness proofs. To a degree, it quite simply is more complicated: we have not only developed (and proved correct) a program that solves our immediate problem of implementing substitution and matching, we have developed a collection of modules for this problem that are individually replacable by anything else matching the same interface.

One could say: we have got rid of hidden assumptions. Our program works, because every module has *separately* been made sure to work, it does not just work by accident. If it works by accident then it also may fail to work by accident if one of our hidden assumptions are violated. Our modularised solution gives us added flexibility when it comes to maintain and modify the program — we only have to modify and check the modules whose interfaces we touch.

The added flexibility comes with a price of additional proof obligations. But it also gives us more insight into our problem: we seriously have to address the questions "what do we really have to assume to solve this problem" and "what can we really guarantee about our solution if we only know that much about these other modules?"

Apart from the modularity issues, our development revealed another class of assumptions. We typically assume that the various functions we call terminate on all inputs and we rely on induction principles without ever actively noticing it. By solely relying on the logical assertions in our module interfaces we are consistently forced to consider the worst case, and the worst case gives us ample opportunities for termination failure. Isolating an abstract type from its usage through module interfaces often forces us to include an induction (or co-induction) principle for reasoning about this type. EML is expressive enough to deal with this problem but it does not provide much support for the user.

References

[DM82] Luis Manuel Martins Damas and Robin Milner. Principal type schemes for functional programs. In *9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specifications I*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[GGM93] Carl A. Gunter, Elsa L. Gunter, and David B. MacQueen. Computing ML equality kinds using abstract interpretation. *Information and Computation*, 107(2):303–323, December 1993.

[Gol81] W. D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[KST94] Stefan Kahrs, Don Sannella, and Andrzej Tarlecki. The definition of Extended ML. Technical Report ECS-LFCS-94-300, University of Edinburgh, 1994.

[KST95] Stefan Kahrs, Don Sannella, and Andrzej Tarlecki. A gentle introduction into the definition of Extended ML. Technical Report ECS-LFCS-95-322, University of Edinburgh, 1995.

[KST97] Stefan Kahrs, Don Sannella, and Andrzej Tarlecki. The semantics of Extended ML: A gentle introduction. *Theoretical Computer Science*, 173(2):445–484, 1997.

[KTU93] A.J. Kfoury, J. Tiuryn, and P. Urcyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 102(1):83–101, January 1993.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[San89] Donald Sannella. Formal program development in Extended ML for the working programmer. Technical Report ECS-LFCS-89-102, University of Edinburgh, 1989.

[Tho91] Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991.

[Wad89] Philip Wadler. Theorems for free. In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.

[Wir90] Martin Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 13, pages 675–788. Elsevier Science Publishers, 1990.