Specifying functions in Extended ML

The Extended ML (EML) specification language is ML with minimal extensions for specifying *what* programs are supposed to do, supplementing (or replacing) code — which specifies *how* to do it — with logical assertions. EML is called a *wide-spectrum language*: there is a smooth continuum from programs through to specifications. The ML subset of EML is executable but the rest is not.

EML covers all of ML except references and input/output. But in this part of the course I am going to make a very strong additional restriction:

> I will only consider ML functions that are total: no exceptions and no non-termination

This considerably simplifies the discussion of proof methods. To further simplify matters, I will be avoiding higher-order functions and polymorphism most of the time.

Specifications in software development. In the software development process, specifications are useful:

1. As a means of communication:
   - between client and developer: to set out what the client requires and the developer agrees to supply ("requirements specification")
   - between members of a development team, to mediate dependencies between modules ("design specification"). Issues of information hiding discussed under data abstraction apply, so part of the point of a module specification is to *avoid* communication, in the sense that we specify *exactly* what a client is allowed to take advantage of — no more and no less.
2. As a design aid:
   - Writing a specification forces details to be considered early in the development process, helping developers to understand the problem at the point where mistakes are most costly. Since the emphasis is on "what" instead of "how", useful generalizations tend to naturally suggest themselves. Writing a specification is regarded as worthwhile even if the specification is thrown away after being written.
   - The program can be developed from the specification by a refinement process ("formal program development")
3. Testing, proof, documentation:
   - As a standard for comparison when testing the program or proving that it is correct.
   - As documentation of the design, for use in maintenance.

There are different kinds of specifications:

1. Informal natural language specifications. These are easy to write and read but can be vague and ambiguous.
2. A simple program can act as a specification of a complicated program. For example, ML can be used as a specification language for programs written in Java. Such specifications are relatively easy to write and precise, but there is a problem with over-specification and it is relatively difficult to prove properties. It is not always so clear what the intention is: what aspects of the simple program are to be ignored?

3. Formal specifications. These are precise and can be used for formal proofs of correctness etc., but they can be hard to write and read.

4. Semi-formal specifications, as in UML. These are easier to write than formal specifications and are very helpful in allowing designers to record their ideas at a conceptually high level. But they are not precise enough for use in e.g. proofs of correctness using a mechanized theorem prover.

The functional programming context is particularly well-suited to formal specifications, and that is what we will be considering in this course.

Specifications versus ML function definitions. ML function definitions are required to have a special form: a sequence of equations where the left-hand side has the form `f(pattern)` and the right-hand side uses only the variables mentioned in the pattern. This is what makes programs executable. Suppose that we consider just the problem of *specifying* functions without the restriction that the definitions be executable. A first step towards this might be to use unrestricted equations.

The equations we write are called *axioms*. Often you need more than one axiom to specify a function.

Square root. For example, consider the problem of computing the square root of a real number. There are various algorithms for computing this by successive approximation, e.g. Newton's method. But if we can use unrestricted equations, we can just write:

```
sqrt : real -> real

sqrt(a) * sqrt(a) = a
```

(See below for an improved version.)

Matrix inversion. Suppose that we have defined a datatype of matrices (let's say square matrices of some particular size), matrix multiplication and the identity matrix `I`. Matrix inversion can be specified like this:

```
inv : matrix -> matrix

inv(A) * A = I
A * inv(A) = I
```

(Again, see below for an improved version.)

Square root, revisited. Once we've dropped the restriction to equations of a special form, we can add other logical notation. For example, logical connectives: we can use `andalso`, `orelse`, `not` and logical implication (`implies`).

Here is a better specification of `sqrt`, where it is only required to work for non-negative numbers:

```
a >= 0.0 implies sqrt(a) * sqrt(a) = a
```

Matrix inversion, revisited. Or we can use universal and existential quantifiers. Here is a better specification of matrix inversion, which takes into account that some matrices are not invertible:

```
Exists B => (B*A=I andalso A*B=I)
  implies
    inv(A) * A = I andalso A * inv(A) = I
```

Note that the syntax for `Exists` (similarly `Forall`) is like `fn` in ML. This is natural because these are also variable-binding constructs. And `.` is used already for qualified identifiers.

All the equations so far have been implicitly universally quantified. EML requires that this be explicit, to make it possible to catch more errors in specifications. So the matrix inversion specification becomes:

```
Forall A =>
  (Exists B => (B*A=I andalso A*B=I)
     implies
       inv(A) * A = I andalso A * inv(A) = I)
```

Maximum of a list. To specify the maximum element of a list, we need to say that it belongs to the list and that it is at least as large as any list element.

Assume that we already have a function `member : int * int list -> bool` that checks to see if an integer is in a list. For example,

```
fun member(n,l) = List.exists (fn x => x=n) l
```

Then we can specify:

```
maxelem : int list -> int

Forall l => l<>nil implies member(maxelem l, l)
Forall (a,l) => member(a,l) implies (maxelem l) >= a
```

String searching. As a more complicated example, consider the problem of specifying two functions

```
before : string * string -> string
after : string * string -> string
```

Given two strings s and r, `before`(s, r) is the part of r before the first occurrence of s in r and `after`(s, r) is the part of r after the first occurrence of s in r. So for example:

```
before("i","specification") = "spec"
after("i","specification") = "fication"
```

We can use the library function `String.isPrefix : string -> string -> bool` (where `String.isPrefix` s r is `true` iff the first part of r matches s) to specify these:

```
Forall (s,r) => before(s,r) ^ s ^ after(s,r) = r

Forall (s,r) => Forall (t1,t2) =>
  t1^s^t2=r implies String.isPrefix (before(s,r)) t1
```

The first axiom says that any string r (containing at least one occurrence of s) consists of the part of r before s, then s, then the part of r after s. The second axiom says that `before` and `after` are with respect to the *first* occurrence of s in r: if there is another way of decomposing r as $t1$`^`s`^`$t2$ then `before`(s, r) must be an prefix of $t1$.

But this specification only works if s occurs in r. We need to add a precondition to the first axiom to avoid imposing an impossible restriction when this is not the case. For example, using a function `substring : string * string -> bool`,

```
Forall (s,r) =>
      substring(s,r) implies before(s,r) ^ s ^ after(s,r) = r
```

where `substring` can be specified like this:

```
Forall (s,r) =>
      substring(s,r) iff (Exists (t1,t2) => t1^s^t2=r)
```

The second axiom doesn't need a precondition: if s doesn't occur in r, its precondition doesn't hold so it is vacuous. So if s doesn't occur in r, `before` and `after` are completely unconstrained.

Note that the specifications of `before` and `after` are completely intertwined, with no single axiom or collection of axioms that are devoted to specifying either `before` or `after`. The second axiom constrains `after` even though it isn't mentioned, because of the way that `before` and `after` are related by the first axiom.

Loose specifications. One of the advantages of using arbitrary equations (with or without logical notations) is that it is possible to write specifications that are purposefully vague or *loose*: we can *constrain* a function or value without completely defining it. For example, we haven't specified `maxelem(nil)`. For `sqrt` we didn't say whether we want the positive or negative square root, and we didn't constrain the result of `sqrt` applied to a negative number at all.

Loose specifications aren't imprecise; we specify exactly the properties we require and don't specify properties that we don't need. This leaves decisions to be made later in the design process.

One way to get a loose specification is to refrain from imposing constraints. Another way is to specify the looseness explicitly. For example, we can specify that `sqrt` is to produce a result which is correct to within 1%:

```
Forall a =>
      a >= 0.0 implies
          sqrt(a) * sqrt(a) >= 0.99*a
          andalso
          sqrt(a) * sqrt(a) <= 1.01*a
```

★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★ ★

Summary of EML notation for axioms. Any expression of type `bool` can be an axiom. Apart from ML's built-in logical connectives — `not`, `andalso` and `orelse` for negation ($\neg$), conjunction ($\wedge$) and disjunction ($\vee$) — EML provides `implies` and `iff` for implication ($\Rightarrow$) and equivalence ($\Leftrightarrow$). Logical equality (`==`) and inequality (`=/=`) may be used at any type (explanation to follow), while ML's computational equality (`=`) and inequality (`<>`) are restricted to types that admit equality, as in ML. Universal quantification $\forall x.\varphi$ is written `Forall x =>` φ, where φ is an expression of type `bool`. Existential quantification $\exists x.\varphi$ is written `Exists x =>` φ, and "there exists a unique" $\exists! x.\varphi$ is written `Exists_unique x =>` φ. Quantification over multiple variables as in $\forall x, y.\varphi$ can be written either `Forall x => Forall y =>` φ or `Forall (x,y) =>` φ.