A Few Pitfalls in Standard ML

Chris Stone

August 31, 2000

1. **Unary negation**.

   The designers of Standard ML decided that they didn't want to overload "`-`" as both the unary negation operator and the binary substraction operator. They chose "`~`" to represent unary negation, including when you're writing negative constants. Thus -5 is written as `~5`, and -0.0005 is written as `~0.0005` or `~5.0e~4`. Similarly you could write a function

   ```
   fun negateAndDouble x = (~x) * 2
   ```

   and so on.

2. **Keywords**.

   The keywords of the SML language include `as`, `do`, `end`, `if`, `in`, and `val`. Trying to use keywords as variable names can lead to confusing error messages.

3. **`and` vs `andalso`**.

   The short-circuting operations on booleans are called `andalso` and `orelse`. So `e1 andalso e2` is equivalent to `if e1 then e2 else false` and `e1 orelse e2` is equivalent to `if e1 then true else e2`.

   `and` is also a keyword, but it is used for something else: primarily, defining a collection of mutually-recursive functions. For example, we could write the (rather inefficient) code:

   ```
   fun even 0 = true
     | even n = odd (n-1)
   and odd  1 = true
     | odd  n = even(n-1)
   ```

4. **Equality and `real`**

   The polymorphic equality operator `=` and inequality operator `<>` can be used to compare strings, integers, booleans, characters, records and tuples of such values, lists of such values, etc. However, it cannot be used to compare two values of type `real`. There are two justifications given for this:

   - Equality ought to be reflexive (that is, `x = x` ought to always evaluate to `true` no matter what `x` is.) Equality for IEEE floating-point, however, `(0.0/0.0) = (0.0/0.0)` is false; both sides are "Not a Number" which is defined not to be equal to anything including itself.

- Arithmetic for floating-point numbers isn't exact, so you probably don't want to be testing for exact equality anyway. More often the right thing is to test that the difference between the two numbers is sufficiently small.

Whether you agree with these or not, to compare floating-point numbers for equality or inequality you must use the functions `Real.==` and `Real.!=` .

5. **The Value Restriction**

The following restriction is made in Standard ML: an expression can be polymorphic only if it is a "value" (Values here include constants, variables, applications of datatype constructors to values, tuples of values, etc.) So, for example, in the definition

```
val id = (fn x => x)
```

this defines `id` to be a variable of type `'a -> 'a`. (Recall that this type means it can be applied to a value of any type and will return a value of the same type.) Similarly,

```
val lst = nil
```

defines `lst` to be a variable of type `'a list`.

One might expect

```
val lst2 = id lst
```

to therefore define `lst2` to be a variable of type `'a list`. However, the right-hand-side is not a value, and the value restriction says that the result is not allowed to be polymorphic. It turns out that possible non-polymorphic types for `lst2` include `int list`, `bool list`, and infinitely many others. Because in this case there is no single best type for `lst2` the compiler will complain. To get this code to compile you have to add a type annotation to tell the compiler what type you mean: either

```
val lst2 : int list = id lst
```

or

```
val lst2 = id (lst : int list)
```

or

```
val lst2 = (id : int list -> int list) lst
```

etc.

Note, however, that there is no problem with the code

```
let
  val lst2 = id lst
in
  4 :: lst2
end
```

because there is enough context to see that `lst2` is being used as a list of integers. It turns out that when you are dealing with whole programs the value restriction doesn't come up that often, but when you try small examples at the interactive loop it can be annoying. See the Guide to using SML/NJ for more details on the exact error message given by the compiler.

6. **Datatype constructors in patterns**.

   The code

   ```
   fun tail x::xs = xs
   ```

   is not parsed the way you might expect. Recall that the syntactic sugar for multiple curried arguments looks like

   ```
   fun K x y = x
   ```

   The parser will look at the tail code and decide that you are writing a function that takes *three* arguments:

   ```
   fun tail (x) (::) (xs) = xs
   ```

   and the compiler complains (essentially) that it can't pattern-match on `::` by itself.

   Similarly, if you write

   ```
   fun length nil = 0
     | length _::t = 1 + length t
   ```

   you get the same error message (plus other complaints that the clauses are inconsistent in the number of arguments expected.)

   The solution is to write

   ```
   fun tail (x::xs) = xs
   ```

   and

   ```
   fun length nil = 0
     | length (_::t) = 1 + length t
   ```

7. **Generativity.**

   Datatypes (and type made abstract by ascribing a signature to a structure via `:>`) are said to be "generative" because they have the following property: every time the definition is evaluated/compiled, it defines a *new* type different from all other types. For example, the last line of the following code will not typecheck:

   ```
   datatype d = A | B
   val x = A
   datatype d = A | B
   val z = (x=A)
   ```

It does not typecheck because the two datatype definitions create two different types: the type of `x` is the first type, and you can only compare two things of the same type for equality, so the last line's comparison of `x` with `A` from the second datatype definition doesn't typecheck.

When this problem occurs, the error message is likely to say something involving a type `?.d` expected where an expression of type `d` was found, or vice-versa.

The place this tends to come up is when the datatype definition is in a file. The file gets loaded and you do some definitions and run some tests while debugging. Then you make some changes to the file and recompile. Any of the debugging definitions you made involving the datatype still refer to the old datatype, and probably can't be used with the new datatype. The solution is to re-evaluate any definitions you want (or put them into a file that gets recompiled when the datatype gets recompiled).