An anecdote about ML type inference

Andrew Koenig (ark@research.att.com)

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

ML strongly supports functional programming; its programmers tend to avoid operations with side effects. Thus, for example, instead of mutable arrays, ML programmers usually use lists. These lists behave much like those in Lisp, with the added requirement that all elements have the same type.[1] ML lists are not themselves mutable, although the individual elements can be of a (uniform) mutable type. In Lisp terms, ML lists can optionally support `RPLACA` but not `REPLACD`. Although it is possible to define a type in ML that would support `REPLACD` as well, ML programmers usually don't do this unless they need it.

The Lisp functions `CAR` and `CDR` have ML equivalents called `hd` and `tl` and ML represents the Lisp `CONS` function by a right-associative infix operator `::` (pronounced ‘‘cons’’).

While learning ML, I thought it would be a nice exercise to try to write a sort function. Although Standard ML does not define arrays, the implementation I was using (Standard ML of New Jersey) did have them. Nevertheless, I felt it would be a more useful exercise to try to write a purely functional sort program: one that operated on lists, rather than arrays, and had no side effects at all.

Strategy

How does such a function behave? It must evidently take a list as argument and return a sorted list as its result. Moreover, since we don't know the type of the list elements, we had better give the function a second argument, namely a function to use for comparisons.

The desired function, then, should have a type[2] of

```
(’a * ’a -> bool) -> ’a list -> ’a list
```

where `’a` represents any type, as usual, so that

```
sort (op <) [3,1,4,1,5,9]
```

would yield

```
[1,1,3,4,5,9]
```

Such a function will be structured something like this:

```
fun sort less x =
    let (* local function definitions go here *)
     in (* some expression *)
    end
```

so as to make the comparison function `less` and the argument `x` available to the local functions without further formality. To make these local functions easier to write, we can therefore start by temporarily defining an appropriate comparison function for integers:

[1] This restriction is typically not much of an impediment, because any set of specific types can be bundled into a single type called a *datatype*. ML datatypes are somewhat like a semantically safe version of C unions.

[2] Experienced ML programmers will note here that `sort` is actually a curried function, so as to allow partial application. For instance, it is nice to be able to treat `sort (op < : int*int->bool)` as a single function without having to supply a list to sort at the same time.

```
fun less(x: int, y) = x < y
```

or, more directly:

```
val less: int * int -> bool = op <
```

In either case, we can now use less as an integer comparison function while writing the inner functions that will make up our sort; when we're ready to combine them, the name less will then smoothly refer to the one that was passed as an argument to sort rather than the one we just defined.

How does one sort a list? The strategy that presented itself was a recursive merge sort: split the list into two pieces, sort the halves, then merge the sorted halves. That suggests that sort should have two local functions called split and merge. With that in mind, I decided to try writing the inner functions.

Tactics

I did not immediately see a good way to split a list, so I began by writing merge. The merge function takes two arguments, each of which is a sorted list. If either of the arguments is empty, the result is the other. Otherwise, we compare the initial elements of the two lists; the result is the smaller of the two followed by the merge of what's left.

That's easier to express as a program than in prose:

```
fun merge(nil, x) = x
  | merge(x, nil) = x
  | merge(x as h::t, y as h'::t') =
      if less(h, h') then h::merge(t,y) else h'::merge(t',x)
```

A few observations may make it easier to follow this program. It is written as a series of alternatives, or *clausal definitions*. These alternatives are always tested in sequence. In other words, merge first checks if its first argument is nil; if so, it returns the second. Next it checks if the second argument is nil; and returns the first argument in that case.

The third clause is more interesting. Note first that an apostrophe in an identifier is just a letter, so that h' is pronounced ''h prime.'' Next note the *layered patterns* that appear in the formal parameters. Saying x as h::t means ''the formal parameter is x, which incidentally must be of the form h::t.'' Recall that :: is like CONS in lisp, so h is the first element of the list h::t and t is the rest of it. The effect of x as h::t, then, is to allow x to refer to the entire argument and simultaneously to allow h and t to refer to the first element and the rest of the list, respectively. This is guaranteed to be possible because the earlier clauses eliminated the case where either argument was nil.

The merge function as shown above seems to work, and indeed it did not take much experimentation to convince me that I had it right. Next I had to tackle split.

The type of split is straightforward: it takes a list as input and returns two lists:

```
'a list -> 'a list * 'a list
```

Now comes the hard part: how does one go about writing it?

The obvious way to split a list is to take the first half and put it one place and the second half and put it somewhere else. The problem with that is that we don't know how big the list is without traversing it; it is then necessary to traverse it *again* to do the actual splitting.

It would be nice to avoid this two-pass property. After thinking about it for a while, I realized that the problem was like to dealing a deck of cards into two equal piles. If I want to do that by cutting the deck, I must count the cards first, but instead I can just deal cards alternately onto the piles until I run out.

That insight makes the solution much easier. Splitting an empty list gives a pair of empty lists. Splitting a list with only one element gives a one-element list and an empty list. Splitting a list with more than one element lets us take the first two elements and put them on the beginning of the result lists, using split recursively to generate the rest of the result. Again it's easier to write than to describe:

```
fun split(nil) = (nil, nil)
  | split([x]) = ([x], nil)
  | split(x::y::tail) =
      let val (p, q) = split(tail)
       in (x::p, y::q)
      end
```

Again, this function works as written and a little experimentation convinced me that I had it right.[3]

Putting it all together

With `merge` and `split` safely in place, it seemed a simple matter to write `sort` itself. We merely follow the earlier description:

```
fun sort(nil) = nil
  | sort(x) =
      let val (p, q) = split(x)
       in merge(sort(p), sort(q))
      end
```

and indeed, when I typed in this program, the compiler accepted it.

I noticed something very curious, however: I expected the type of `sort` to be

```
int list -> int list
```

because the previous definition of `less` constrains this particular `sort` to work only on lists of integers. Much to my surprise, the compiler reported a type of

```
'a list -> int list
```

In other words, this `sort` function accepts a list of any type at all and returns a list of integers.

That is impossible. The output must be a permutation of the input; how can it possibly have a different type? The reader will surely find my first impulse familiar: I wondered if I had uncovered a bug in the compiler!

After thinking about it some more, I realized that there was another way in which the function could ignore its argument: perhaps it sometimes didn't return at all. Indeed, when I tried it, that is exactly what happened: `sort(nil)` did return `nil`, but sorting any non-empty list would go into an infinite recursion loop.

Once I saw this, it was not hard to figure out why. Suppose, for example, that we are sorting a one-element list. We split it into a one-element list and an empty list and then try recursively to sort the one-element list. That recursion makes no progress, so the computation never terminates.

The fix is simple: add a clause to `sort` to handle a single-element argument. If the argument is more than one element, `split` will certainly reduce it, so that should be sufficient:

[3] ML experts will note that the `split` function can more efficiently be written this way:

```
fun split x =
       let fun loop(p, q, nil) = (p, q)
             | loop(p, q, [a]) = (a::p, q)
             | loop(p, q, a::b::rest) = loop(a::p, b::q, rest)
        in loop(nil, nil, x)
       end
```

This function is faster because it is a tail recursion. It also has the side effect of reversing the list on the way, thus yielding an unstable sort. It is an interesting exercise to restore the stability of the sort by reversing the list *again* during merging. This requires the added finesse that the comparison function must have its sense reversed during odd-numbered recursions, because it is then being used to sort a reversed list. The result can be a substantially faster sort, but is certainly harder to understand.

```
fun sort(nil) = nil
  | sort([a]) = [a]
  | sort(x) =
      let val (p, q) = split(x)
       in merge (sort(p), sort(q))
      end
```

The compiler reports the type of this function as

```
int list -> int list
```

as expected, and again a few experiments show convincingly that it does what it should.

It is now easy to write the general `sort` function as promised earlier. We need merely to make `less` an argument to `sort`, account for that in the recursive calls, and make the `merge` and `split` functions local:

```
fun sort less nil = nil
  | sort less [a] = [a]
  | sort less x =
    let fun merge (nil, x) = x
          | merge (x, nil) = x
          | merge (x as h::t, y as h'::t') =
              if less(h, h') then h::merge(t,y) else h'::merge(t',x)
        fun split(nil) = (nil, nil)
          | split([x]) = ([x], nil)
          | split(x::y::tail) =
              let val (p, q) = split(tail)
               in (x::p, y::q)
              end
        val (p, q) = split(x)
     in merge (sort less p, sort less q)
    end
```

This function does indeed work as expected, and its type is

```
('a * 'a -> bool) -> 'a list -> 'a list
```

If we had written this entire function before testing it, and made the same mistake by leaving out the line saying

```
  | sort less [a] = [a]
```

then the compiler would have reported the type as

```
('a * 'a -> bool) -> 'b list -> 'a list
```

thus showing again that something was wrong.

Reflections

An ML function computes a result from its arguments and whatever other information is available to it at the time it is called. Thus it cannot create a result of a completely new type; the result type must somehow depend on the types that already exist.

That means that if the compiler determines the type of a function to be, say,

```
'a list -> 'b list
```

one should be instantly suspicious. Where could `'b` have come from? It's not the type of something that already exists; such a type would be known explicitly and would not have to be expressed in terms of the type metavariable `'b`. I can think of only three possibilities:

— The function returns an empty list, so that the types of the elements that the list doesn't contain are irrelevant. One example is the function

```
fun f(x) = if null(x) then nil else nil
```

— the function raises an exception, so that the type of the value it doesn't return is irrelevant, or

— the function loops and never terminates at all.

Of course, the function could also do one or another of these things depending on its argument. For example:

```
fun f(nil) = nil
  | f([a]) = raise x
  | f(y) = f(y)
```

returns `nil` if given a `nil` argument, raises exception `x` if given an argument that is only a single element, and otherwise runs forever. The type of that function is indeed

```
'a list -> 'b list
```

In any event, if an ML compiler determines that a function returns a type that is unrelated to its argument types, that should be cause for suspicion.

We constantly hear that the earlier a defect is found and corrected, the less expensive it is. In software, that is a powerful argument for strong typing: a compiler that does aggressive type-checking can often detect errors long before the program is actually run.

This particular case, however, is new to my experience: never before had I seen a compiler prove non-termination of a program I had written! The reason in this case is clear: the ML type-checker infers the most general type that can be ascribed to a particular function. To do that, it must effectively do flow analysis. Of course, it is impossible to prove non-termination for all non-terminating programs; this makes it doubly impressive that it can be done at all.

References

The definition of Standard ML is called, logically enough, *The Definition of Standard ML* by Milner, Tofte, and Harper, (MIT Press 1990, ISBN 0-262-13255-9).

Only a masochist would attempt to learn ML from the Definition. A much more approachable (but less rigorous) source is *ML for the Working Programmer* by Laurence Paulson (Cambridge University Press 1991, ISBN 0-521-39022-2).

There are many books that describe Lisp. A reasonable place to start is *Lisp*, by Winston and Horn (Addison-Wesley 1989, 0-201-08319-1).

Finally, all the examples shown here were run on the Standard ML of New Jersey implementation, version 0.75. The current successor to that implementation is available free of charge by anonymous FTP from `research.att.com`; look in directory `dist/ml`.

About the author

Andrew Koenig is a Distinguished Member of Technical Staff at AT&T Bell Laboratories, where since 1986 he has worked mostly on C++. He joined Bell Labs in 1977 after completing an MS degree in computer science at Columbia University. Aside from C++, his work has included programming language design and implementation, security, automatic software distribution, online transaction processing, and computer chess. He is the author of more than seventy articles and the book 'C Traps and Pitfalls.' He is the Project Editor of the ISO/ANSI C++ committee and a member of five airline frequent flyer clubs.