15-150 Fall 2013

Stephen Brookes

Lecture 11
Case study: general sorting

1 Themes

- An abstract formulation of sorting
- Types equipped with comparison functions
- Higher-order functions and polymorphism in action

2 Background

- You should have read the lecture notes on sorting integer lists and trees, types, polymorphism, functions as values, and higher-order functions.
- We use ML syntax for "curried" functions, such as

  ```
  fun f (x:t1) (y:t2) : t' = e,
  ```

  a recursive definition for a function `f` of type `t1 -> (t2 -> t')`. The `->` operator on types associates to the *right*, so we can write this type as `t1 -> t2 -> t'`. You may still need to put in parentheses: `(int -> int) -> int` is not the same type as `int -> int -> int`.

  Contrast this with the "uncurried" syntax

  ```
  fun g (x:t1, y:t2) : t' = e,
  ```

  which defines a function `g` of type `t1 * t2 -> t'`.

  A function `f` of type `t1 -> t2 -> t'` can be applied to an expression `e1` of type `t1`; the application expression `f e1` has type `t2 -> t'`, and evaluates to a function value (if it terminates). We can apply `f e1` to an expression `e2` of type `t2`; `(f e1) e2` has type `t'`. Application associates to the *left*, so we can write either `(f e1) e2` or `f e1 e2`. Again you may need to use parentheses: `(f g) x` is not the same expression as `f(g x)`.

- We will also use patterns in defining such functions, as in:

  ```
  fun f p11 p12 = e1
   |  f p12 p22 = e2
  ```

  These templates also generalize to more than 2 curried arguments, and more than 2 clauses.

3 General sorting

- Sorting collections of data for which there is a "comparison" function. This includes integers (with the usual "less-than" comparison, or with the "greater-than" comparison). It also includes tuples of integers, with lexicographical comparison on components. It also includes strings, ordered as in conventional dictionaries.

- By parameterizing our code design we can be very flexible, and write a single sorting function that can be used for any of these specific purposes.

- We make some general assumptions about the type and properties of a "comparison" function. Standard notions of comparison, such as the standard $<$ on integers, do have these properties. So do "dictionary" orderings for strings, and lexicographic orderings for tuples and lists.

- By defining a comparison as a function that returns a value of type `order`, rather than returning a truth value, we avoid the need to keep distinguishing between "less than" and "less than or equal to". If we want to, we can easily "recover" the implicit less-than relation that corresponds to a comparison function.

- Let `t` be a type whose values represent "data". A comparison function `cmp` of type `t * t -> order` should satisfy:

  (i) For all values `x,y:t`, `cmp(x,y)` evaluates to a value;

  (ii) For all values `x,y:t`,

  ```
  cmp(x,y)=LESS if and only if cmp(y,x)=GREATER,
  cmp(x,y)=EQUAL if and only if cmp(y,x)=EQUAL;
  ```

  (iii) For all values `x,y,z:t`

  ```
  (a) If cmp(x,y)=LESS & cmp(y,z) <> GREATER then cmp(x,z)=LESS
  (b) If cmp(x,y)=GREATER & cmp(y,z) <> LESS then cmp(x,z)=GREATER
  (c) If cmp(x,y)=EQUAL & cmp(y,z)=EQUAL then cmp(x,z)=EQUAL.
  ```

  When these hold we say that `cmp` is a comparison for type `t`.

NOTE: It follows from these properties that for all values `x:t`, `cmp(x,x)=EQUAL`.

4 Comparisons

```
datatype order = LESS | EQUAL | GREATER;

(* val LESS : order     *)
(* val EQUAL : order    *)
(* val GREATER : order  *)

(* compare : int * int -> order *)
fun compare(x:int, y:int):order =
   if x<y then LESS else
   if y<x then GREATER else EQUAL;
```

`compare` is a comparison for type `int`. This is easy to check (but tedious!). `compare` is the "usual" less-than comparison for integers.
Examples:

```
compare(2,3) = LESS
compare(3,2) = GREATER
```

ML actually has the type `order` and the above comparison function as a built-in function named `Int.compare`. We include the definitions here to keep our code self-contained.

ML also has a comparison for strings, named `String.compare`.

```
String.compare : string*string -> order
```

Examples:

```
String.compare("foo", "fool") = LESS
String.compare("foo", "bar") = GREATER
```

There are many additional examples of types and comparisons.
For instance, with data of type `int * int` we can compare with respect to less-than on first components, and we can also compare with respect to less-than of second components:

```
(* leftcompare : (int * int) * (int * int) -> order *)
fun leftcompare((x1, y1), (x2, y2)) = compare(x1, x2);

(* rightcompare : (int * int) * (int * int) -> order *)
fun rightcompare((x1, y1), (x2, y2)) = compare(y1, y2);
```

Examples:

```
leftcompare((1,2000), (2, 1)) = LESS
rightcompare((1,2000), (2,1)) = GREATER
```

Check that `leftcompare` and `rightcompare` satisfy the required properties for a comparison function on type `int * int`.

Now some ways to obtain new comparisons from old.

Reversing an ordering

`flip` is the obvious operation that "reverses" a comparison, or turns it "upside-down".

```
(* flip : ('a * 'a -> order) -> ('a * 'a -> order)  *)
fun flip cmp (x, y) = cmp (y, x);
```

If `cmp` is a comparison for type `t`, so is `flip(cmp)`.

Example: `flip compare` is the "greater-than" comparison for integers:

```
flip compare (x, y) = LESS iff x > y
flip compare (2,3) = compare(3,2) = GREATER
```

Lexicographic ordering for tuples

If we have two comparisons (possibly on different types), we can define a "lexicographic" comparison on pairs of values, by using the first comparison on the first components; if this returns `LESS` or `GREATER` we take that as the result of comparing the two pairs; otherwise (the first components are "equal" according to the first comparison) we use the second comparison on the second components of the pairs.

The following ML function encapsulates this way to build a lexicographic comparison out of two comparisons.

```
(* lex : ('a*'a -> order)*('b*'b -> order) -> (('a*'b)*('a*'b) -> order) *)
fun lex (cmp1, cmp2) ((x1, y1), (x2, y2)) =
    case cmp1(x1, x2) of
        LESS => LESS
      | GREATER => GREATER
      | EQUAL => cmp2(y1,y2);
```

Again it is straightforward to show (and tedious because there are so many little properties to check) that:

> If `cmp1` is a comparison for `t1` and `cmp2` is a comparison for `t2`, then `lex(cmp1, cmp2)` is a comparison for `t1 * t2`.

In particular, `lex(compare, compare)` is a comparison for `int * int`.

Recall that `compare` is integer comparison, i.e.

```
compare(x,y) = LESS iff x<y;
```

`lex(compare,compare)` is the "lexicographic less-than" comparison on pairs of integers:

```
lex(compare,compare)((x,y),(x',y')) = LESS iff x<x' or (x=x' & y<y').
```

`lex` provides a way to build a comparison on pairs using comparisons on component types. For any tuple type there is an analogous version of this construction, for instance for triples.

Exercises

- Let `cmp1` and `cmp2` be comparisons for types `t1` and `t2`. `flip(lex(cmp1, cmp2))` and `lex(flip(cmp1), flip(cmp2))` are both comparisons for `t1 * t2`. Are they the *same* comparison?

- Define a function

  ```
  listlex : ('a * 'a -> order) -> 'a list * 'a list -> order
  ```

  such that when `cmp` is a comparison for type `t`, `listlex(cmp)` is a comparison for type `t list`.
  HINT: use these equations to guide you.

  ```
  listlex cmp ([ ], R) = LESS if R <> [ ]
  listlex cmp (x::L, [ ]) = GREATER
  listlex cmp (x::L, y::R) = cmp(x,y) if cmp(x,y)<>EQUAL
  listlex cmp (x::L, y::R) = listlex cmp (L, R) if cmp(x,y)=EQUAL.
  ```

Less-than, and less-than-or-equal

Given a comparison, we can recover from it a less-than function, and a less-than-or-equal function, both of which return a truth value.

```
(* less : ('a * 'a -> order) -> ('a * 'a -> bool) *)
fun less cmp (x, y) = (cmp(x, y) = LESS);

(* lesseq : ('a * 'a -> order) -> ('a * 'a -> bool) *)
fun lesseq cmp (x, y) = (cmp(x, y) < > GREATER);
```

Obviously we can also go the other way too, but we'll omit the details.

5 Sorted

Given a type `t` and a comparison `cmp` for `t`, we can specify what it means to say that a list of items of type `t` is `cmp`-sorted. As before, this means that each item in the list is "less-than-or-equal" to all items that occur later in the list, *according to the comparison function.* We can again encapsulate this definition as an ML function:

```
(* sorted : ('a * 'a -> order) -> 'a list -> bool *)
fun sorted cmp [ ] = true
 |  sorted cmp [x] = true
 |  sorted cmp (x::y::L) = case cmp(x,y) of
                            GREATER => false
                            | _ => sorted cmp (y::L);
```

When `cmp` is a comparison, we say that a list `L` is `cmp`-sorted if and only if `(sorted cmp L = true)`.

If we let `cmp` be the standard integer comparison function (`compare`, as before), this is the same as saying that an integer list is sorted in the usual sense. This fact is a sanity check that confirms we have made a smooth generalization from the integer setting to a more general setting.

Now that we have shown that there are many examples of types and comparisons to work with, let's revisit the insertion sort function on arbitrary lists.

6 Insertion sorting

Here is insertion sort on general lists, an easy adaptation of the prior code that worked on integer lists. Most functions here have a type that is slightly more complex than before, typically having an additional parameter that represents the comparison. And the spec is more general, in the same manner. The correctness proofs are actually very similar to those we developed earlier. In the proof details, most of which we avoid giving, the basic assumptions about comparison functions are important.

If you have already seen that the function

```
isort : int list -> int list
```

from a prior lecture is expressible using `foldr` or `foldl` (and both ways lead to equivalent code), you will soon learn that when we generalize to more complicated types and comparisons this property may fail.

Insertion

To insert into a `cmp`-sorted list we need to take account of the comparison. So we introduce a function

```
ins : ('a * 'a -> order) -> ('a * 'a list) -> 'a list
```

given by

```
(* REQUIRES: cmp is a comparison and L is a cmp-sorted list *)
(* ENSURES: ins cmp (x,L) = a cmp-sorted permutation of x::L *)

fun ins cmp (x, [ ]) = [x]
 |  ins cmp (x, y::L) =
     case cmp(x, y) of
         GREATER => y::ins cmp (x, L)
        |   _    => x::y::L;
```

Note that `ins cmp (x, y::L) = x::y::L` if `cmp(x, y) = EQUAL`.

We can prove (by induction on `L`) that this function meets its spec:

> **Lemma** For all comparisons `cmp` and all `cmp`-sorted lists `L`, `ins cmp (x, L)` evaluates to a `cmp`-sorted permutation of `x::L`.

(In the details of this proof, you'll need to use the properties that we assumed for a comparison function. We leave the details as an exercise.)

Foldable functions

Given a comparison `cmp` for type `t`, `ins cmp` evaluates to a function of type `t * t list -> t list`. We can therefore "fold" this function along a list of type `t list`, given a "base" value (also a `t list`). Since there are two list folding functions (`foldl` and `foldr`) we can do this in two ways.

Recall the definitions for

```
foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

fun foldl g z [ ]    = z
 |  foldl g z (x::L) = foldl g (g(x,z)) L;

fun foldr g z [ ]    = z
 |  foldr g z (x::L) = g(x, foldr g z L);
```

If we have, as above, `(ins cmp) :  t * t list -> t list`, the most general type of `foldl (ins cmp)` is

```
t list -> t list -> t list
```

because we need to instantiate `'a` as `t` and `'b` as `t list` to make the argument type `('a * 'b -> 'b)` in `foldl`'s type look like the type of `ins cmp`. Then the application `foldl (ins cmp) [ ]` has type `t list -> t list`.

Similarly for `foldr (ins cmp) [ ]`.

Left-handed insertion sort

In the discussion in the previous paragraphs, `t` was an "arbitrary" type. Our type-checking analysis makes sense for any choice of `t`. In fact, we could just as well have argued that if `cmp` was a function of type `('a * 'a -> order)`, the expression `foldl (ins cmp) [ ]` has type `'a list -> 'a list`.

Hence the function definitions that follow below are well-typed, with most general types as indicated in the comments.

The left-handed general insertion sort function is:

```
(* isortl : ('a * 'a -> order) -> 'a list -> 'a list     *)
(* REQUIRES: cmp is a comparison *)
(* ENSURES: isortl cmp L = a cmp-sorted permutation of L *)
fun isortl cmp L = foldl (ins cmp) [ ] L;
```

Examples:

```
isortl compare [3,1,2,1] = [1,1,2,3]
isortl (lex(compare,compare)) [(1,2),(2,1),(1,1),(2,2)]
      = [(1,1),(1,2),(2,1),(2,2)]
```

We can prove that this function satisfies its specification, as usual, using induction. (Look up the proof for the integer list insertion sort function, to see how we've generalized!) Here's what we'd like to prove:

> If `cmp` is a comparison for type `t`, then for all lists `L:t list`,
>
> `isortl cmp L = a cmp-sorted permutation of L.`

If you try to prove this, by induction on `L`, you won't get very far! The reason: `isortl` is *not recursive*, and is instead defined using `foldl`.

Since by definition `isortl cmp L = foldl (ins cmp) [ ] L` you might try to prove

> If `cmp` is a comparison for type `t`, then for all lists `L:t list`,
>
> `foldl (ins cmp) [ ] L = a cmp-sorted permutation of L.`

by induction on `L`. But you would quickly notice that we make a recursive call to `foldl (ins cmp)` on a non-empty list, so you won't be able to appeal to the induction hypothesis. Instead we need to prove something even more general:

> **Theorem** If `cmp` is a comparison for type `t`, then for all lists `L:t list` and all `cmp`-sorted lists `A:t list`,
>
> `foldl (ins cmp) A L = a cmp-sorted permutation of L@A.`

(Letting `A` be the empty list then gives us the desired result.)
Proof of Theorem: By induction on `L`. Suppose `cmp` is a comparison.

- Base case: For `L = [ ]`.
  Show that for all `cmp`-sorted lists `A`,

  `foldl (ins cmp) A [ ] = a cmp-sorted permutation of [ ]@A.`

  (This is very easy!)

- Inductive case: For `L=x::R`. Assume the Induction Hypothesis

```
(IH): For all cmp-sorted lists B,
     foldl (ins cmp) B R = a cmp-sorted permutation of R@B.
```

  Show that for all `cmp`-sorted lists `A`,

```
foldl (ins cmp) A (x::R) = a cmp-sorted permutation of (x::R)@A.
```

  Here is a sketch of the details. Let `A` be a `cmp`-sorted list. Then

```
foldl (ins cmp) A (x::R)
  = foldl (ins cmp) (ins cmp (x, A)) R
                                    (by def of foldl)
  = foldl (ins cmp) B R
          where B is a cmp-sorted permutation of x::A
                                    (by spec for ins cmp)
  = a cmp-sorted permutation of R@B
          where B is a permutation of x::A
                                     (by IH)
  = a cmp-sorted permutation of (x::R)@A
                                     (by properties of permutations)
```

  (Note carefully where we needed to use the assumption that `A` was a `cmp`-sorted list.)

Right-handed insertion sort

Here is the right-handed version:

```
(*  isortr : ('a * 'a -> order) -> 'a list -> 'a list     *)
fun isortr cmp L = foldr (ins cmp) [ ] L;
```

Examples:

```
   isortr compare [3,1,2,1] = [1,1,2,3]
   isortr (lex(compare,compare)) [(1,2),(2,1),(1,1),(2,2)]
         = [(1,1),(1,2),(2,1),(2,2)]
```

Now we'd like to prove that:

> If `cmp` is a comparison for type `t`, then for all lists `L:t list`,
>
> `isortr cmp L = a cmp-sorted permutation of L.`

Again we need to generalize and prove instead:

> **Theorem** For all comparisons `cmp`, all lists `L`, and all `cmp`-sorted lists `A` of the appropriate type,
>
> `foldr (ins cmp) A L = a cmp-sorted permutation of L@A.`

Exercise: do this proof, using induction on `L`.

Again letting `A` be the empty list gives us the desired result.

Left *vs.* right

Although the test examples shown above for `isortl` and `isortr` produce the same results, this isn't always the case! In fact, `isortl` is NOT equivalent to `isortr`!

For example, let `cmpleft` be given by:

```
fun cmpleft((x,y), (x',y')) = compare(x,x');
```

`cmpleft` is a comparison for `int * bool`. But

```
isortl cmpleft [(1,true),(1,false)] = [(1,false),(1,true)]
isortr cmpleft [(1,true),(1,false)] = [(1,true),(1,false)]
```

(There's nothing contradictory here – both of these lists are `cmpleft`-sorted permutations of the original list.) But it follows that

$$\texttt{isortl cmpleft } [(1,\texttt{true}),(1,\texttt{false})] \neq \texttt{isortr cmpleft } [(1,\texttt{true}),(1,\texttt{false})]$$

and hence `isortl cmpleft` $\neq$ `isortr cmpleft`. Further, this tells us that `isortl` $\neq$ `isortr`. (Remember how we defined "equality" for functions!)

The reason for this difference is suggested by the "algebraic" specifications for `foldl` and `foldr`. Note that if `cmp` is a comparison, `ins cmp` is a total function. Let `i = ins cmp`, for brevity. Then the algebraic specs say that for all $n \geq 0$ and all lists $[\texttt{x}_1, \ldots, \texttt{x}_n]$,

$$\texttt{foldl i } [\,]\ [\texttt{x}_1, \ldots, \texttt{x}_n] = \texttt{i}(\texttt{x}_n, \ldots, \texttt{i}(\texttt{x}_1, [\,]) \ldots)$$
$$\texttt{foldr i } [\,]\ [\texttt{x}_1, \ldots, \texttt{x}_n] = \texttt{i}(\texttt{x}_1, \ldots, \texttt{i}(\texttt{x}_n, [\,]) \ldots)$$

The "algebraic" spec for `foldr` implies that `isortr cmp` is "stable" – it preserves the relative list order for `cmp`-equal items. And the algebraic spec for `isortl` implies that `isortl cmp` does NOT do this. (It reverses the relative order of `cmp`-equal items! In fact if all items in `L` are `cmp`-equal, we get `foldl i [ ] L = rev(L)` and `foldr i [ ] L = L`.)

Incidentally, the reason we couldn't tell the left-handed and right-handed versions apart when we used `compare` on `int`, or `lex(compare, compare)` on `int * int`, is because for these cases there is always *exactly one* sorted permutation of a given list. That's not true in general.

7 Stability

A sorting function is stable if it preserves the relative ordering of items for which the comparison result is `EQUAL`. We already saw that `isortr` is stable and `isortl` is not.

We can characterize stability very nicely as an equational property, as follows. First, real the list filtering function:

```
fun filter p [ ] = [ ]
 |  filter p (x::L) = if (p x) then x :: filter p L else filter p L;
```

As before, `filter p L` returns the list of those items in `L` that satisfy `p`.

We can define a predicate for checking if a value is "equal" to a given value, using a given comparison function:

```
(* same : ('a * 'a -> order) -> 'a -> 'a -> bool *)
fun same cmp x y = (cmp(x,y) = EQUAL);
```

So, given a comparison `cmp` for type `t` and a value `v` of type `t`, `same cmp v` evaluates to a function value equal to

```
fn y => (cmp(v,y)=EQUAL)
```

of type `t -> bool`, representing the predicate for checking "`cmp`-equal-to-`v`". We can then say that a function

```
s : ('a*'a -> order) -> 'a list -> 'a list
```

is *stable* iff for all comparisons `cmp`, and all suitably typed `x` and `L`,

```
filter (same cmp x) L = filter (same cmp x) (s cmp L).
```

Incidentally, this example shows the wisdom of designing a function carefully. We deliberately chose to make `same` a "curried" function, and this enabled us to "partially apply" it to `cmp` and `x` to obtain a function that we then used to filter lists.

8 Design matters

We could have turned the insertion function into a local function, as in:

```
fun isortr cmp L =
   let
      fun ins cmp (x, [ ]) = [x]
       |  ins cmp (x, y::L) =
           case cmp(x, y) of
               GREATER => y::ins cmp (x, L)
              |   _    => x::y::L
   in
      foldr (ins cmp) [ ] L
end;
```

In this code (and in the original development) the same comparison function is used in every recursive call to `ins`. We can rewrite the code to use scoping to avoid this redundancy, as:

```
fun isort cmp L =
 let
   fun ins (x, [ ]) = [x]
    |  ins (x, y::R) = case cmp(x, y) of
                           GREATER => y::ins(x, R)
                         |   _     => x::y::R
 in
   foldr ins [ ] L
 end;
```

The idea here is that, for instance, when we call `isort compare` a local function named `ins` is introduced and this function refers to the name `cmp` in its body; this name is bound to `compare`. Every recursive call of `ins` thus uses the same binding.

This `isort` function still satisfies the same spec as before:
If `cmp` is a comparison, then for all lists `L` of appropriate type, `isort cmp L` evaluates to a `cmp`-sorted permutation of L.

9 Remarks

- Mergesort and quicksort on general lists can be obtained in a similar manner. These are good exercises!
- The tree-based sorting code given earlier can also be adapted easily to work in this more general setting.
- We deliberately used curried functions with the comparison as the "first" argument. That enabled us to use partial application to get a sorting function specialized to work with a specific comparison.
- And we deliberately used polymorphic types: we can instantiate by choosing a type for data, and re-use the same ML code over and over again to sort many different kinds of data. One function definition; many re-uses.
- If we prove the correctness of a polymorphically typed function with respect to a general specification ("for all types,...") , we can re-use the same proof for free at any instance: here, for all types of data and all comparison functions. One proof; many conclusions!
- We didn't do any work or span analysis today. If you assume that the comparisons take constant time, it's not difficult to re-do the work and span analysis from earlier classes.
- When dealing with expressions having polymorphic types, we often give specifications of the form "For all types `t`, and all values `v` of type `t`, ...". Avoid the temptation to be sloppy by saying "For all values `v` of type `'a`, ..." and hoping that this statement means the same thing: it doesn't. There are NO values of type `'a`.
  Some polymorphic types do have values: for example, the identity function `fn x:'a => x` is a value of type `'a => 'a`.
  But saying "For all values of type `'a -> 'a`" does not have the same effect as saying "For all types `t`, and all values of type `t -> t`". For example, `fn x:int => x+1` is a value of type `int -> int`, but NOT a value of type `'a -> 'a`. The two statements quantify over a different set of types and values.