15-150 Fall 2017

Stephen Brookes

Lecture 6
Sorting lists of integers
1 Outline

- insertion sort and mergesort
- specifications, correctness, and efficiency
- inductive proof techniques, in action

2 Remarks

In the code below we provide clear and accurate specifications but we don’t always use the REQUIRES and ENSURES format. In lecture today we discussed much of the same code. These notes supplement what was said and done in class, offering a different perspective and sometimes more detail. (The discussion of work in these notes may occur in the next lecture, depending on time.)

3 Background

We introduced the following type and function; each is already pre-defined in SML, but we give explicit definitions since we use them for the first time. (Actually the function compare is called Int.compare in SML. We give it a shorter name just for convenience.)

datatype order = LESS | EQUAL | GREATER

(* Comparison for integers *)

(* 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(x,y)=LESS if x<y *)
(* compare(x,y)=EQUAL if x=y *)
(* compare(x,y)=GREATER if x>y *)
The first line above is a simple form of \textit{datatype} definition. It introduces a user-defined type named \texttt{order}. The type \texttt{order} has just three (syntactic) values, written \texttt{LESS}, \texttt{EQUAL}, and \texttt{GREATER}. You can test expressions of this type for equality (using \texttt{=}) or use them in case-expressions to do different things based on a comparison between two integer values. (The type \texttt{order} is an equality type.)

Later in the semester we will see more sophisticated datatype definitions. The ability to define your own types and have them fit in seamlessly with the rest of the ML type discipline is a very powerful feature of SML.

**Linear ordering**

The $\leq$ operator on integers has some well known properties. In particular, $\leq$ is a linear ordering. This means that for all integers $x$, $y$, $z$:

- $x \leq y \land y \leq x$ implies $x = y$ (antisymmetry)
- $x \leq y \land y \leq z$ implies $x \leq z$ (transitivity)
- $x \leq y \lor y \leq x$ (totality)

**Sorted lists**

A list of integers is \textit{sorted} if each item in the list is $\leq$ all items that occur later in the list. Here is an ML function that checks for this property.

We only use this function in specifications!

\begin{verbatim}
(* sorted : int list -> bool *)
fun sorted [] = true
  | sorted [x] = true
  | sorted (x::y::L) =
      (compare(x,y) <> GREATER) andalso sorted(y::L)

(* REQUIRES L is an integer list *)
(* ENSURES sorted L = true iff L is a sorted list *)
\end{verbatim}

Note the use of $\langle\rangle$, which is ML for “not equal to” and can be used on values of type \texttt{order}. (You can use $\langle\rangle$ and $=$ on values of any equality type, e.g. on integers.) The expression $\text{compare}(x,y) \langle\rangle \text{GREATER}$ evaluates to \texttt{true}.  

3
if the value of \texttt{compare}(x, y) is \texttt{LESS} or \texttt{EQUAL}, and evaluates to \texttt{false} if \( x > y \).

Examples:

\begin{itemize}
  \item \texttt{sorted} [1,2,3] = \texttt{true}
  \item \texttt{sorted} [3,2,1] = \texttt{false}
\end{itemize}

Make sure you see the relevance of the linear ordering properties here: they are the reason why this \texttt{sorted} function behaves as described! And they also explain why it’s not necessary in the third clause of the function to check explicitly that \( x \) is not greater than the elements of \( L \); the term \texttt{sorted}(y::L) checks that \( y \) is less-than-or-equal to the elements of \( L \), and the knowledge that \( x < y \) is enough (because of transitivity).

Exercise: prove by induction on the length of \( L \), that \texttt{sorted}(L) = \texttt{true} if every item in \( L \) is \( \leq \) all later items in \( L \), and \texttt{sorted}(L) = \texttt{false} otherwise. Indicate clearly in your proof when you appeal to any of the above properties of linear orderings.

From now on we will say that an integer list \( L \) is \texttt{sorted} if and only if \texttt{sorted}(L) evaluates to \texttt{true}. We will assume you remember the basic properties described above, and we will often take advantage of “obvious” properties, such as: an integer list of length 1 is sorted, and an integer list of form \([x, y]\) is sorted if and only if \( x \leq y \). Usually these facts are easy to prove from the definition of sortedness, and they rely on the basic properties of linear orderings.
4 Insertion sort

Here is a function that implements insertion sort. This algorithm is often (informally) described as building up its sorted output list by starting with an empty list and successively inserting the items from the input list, at each stage maintaining the correct sorted order among the items inserted so far. Unfortunately this kind of description is too vague to be precise, using fuzzy words like “building up” and “successively”. A better (more precise) description would be:

To insertion sort a list: if the list is empty, do nothing; otherwise, (recursively) insertion sort the tail of the list and then insert the head (into the sorted tail list).

Of course we need a helper function for inserting an item into a list, and we can see from the above that we will only need to do an insertion into an already sorted list. Let’s code this algorithm in SML. We won’t need to use ML functions for extracting heads and tails of lists; instead we’ll use pattern matching.

First we define a helper function for inserting an integer into its proper place in a sorted list. To be clear about what this means, we refer to the familiar notion of permutation. An integer list A is a permutation of a list B if A contains the same items as B, possibly in a different order, and each integer occurs the same number of times in A as in B. For example, [1,2,3,1] is a permutation of [2,1,3,1] and a permutation of 1::(1::2::[3]), but not a permutation of [1,2,3].

(* ins : int * int list -> int list *)
(* REQUIRES L is a sorted list of integers *)
(* ENSURES ins (x, L) = a sorted permutation of x::L *)

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

Examples:

ins (2, [1,3]) = [1,2,3]
ins (2, [3,1]) = [2,3,1]
ins (2, [1,2,3]) = [1,2,2,3]
Using \texttt{ins} as a helper, we can implement insertion sort as follows:

\begin{verbatim}
(* isort : int list -> int list *)
(* REQUIRES L is an integer list *)
(* ENSURES isort(L) = a sorted permutation of L *)

fun isort [ ] = [ ]
  | isort (x::L) = ins (x, isort L)
\end{verbatim}

In the lecture slides and on the blackboard we sketched a proof that \texttt{ins} satisfies its specification, and that \texttt{isort} satisfies its specification. Be sure to study these proofs (and fill in any missing details). They are a very useful exercise in understanding how recursion works.

A variation

Just for interest, here is a slight variation on this theme:

\begin{verbatim}
(* isort2 : int list -> int list *)
fun isort2 [ ] = [ ]
  | isort2 [x] = [x]
  | isort2 (x::L) = ins (x, isort2 L)
\end{verbatim}

We can show easily that this \texttt{isort2} function is extensionally equivalent to \texttt{isort} as defined above. (So the singleton clause is irrelevant.) Indeed, it is very straightforward to prove, by induction on \( n \), that:

For all \( n \geq 0 \) and all integer values \( x_1, \ldots, x_n \),

\[ \text{isort} [x_1, \ldots, x_n] = \text{isort2} [x_1, \ldots, x_n]. \]

- For \( n = 0 \), we have \( \text{isort} [ ] = [ ] = \text{isort2} [ ] \).
- For \( n = 1 \), we have

\[
\begin{align*}
\text{isort} [x_1] &= \text{ins}(x_1, \text{isort} [ ]) \quad \text{by def of isort} \\
&= \text{ins}(x_1, [ ]) \quad \text{by def of isort} \\
&= [x_1] \quad \text{by def of ins} \\
&= \text{isort2} [x_1] \quad \text{by def of isort2}
\end{align*}
\]

- For the inductive step, suppose \( n > 1 \) and the property holds for all lists of length \( n - 1 \), i.e.
For all integer values \( y_1, \ldots, y_{n-1} \),

\[
isort [y_1, \ldots, y_{n-1}] = isort2 [y_1, \ldots, y_{n-1}] .
\]

Let \( L \) be an integer list of length \( n \), of the form \([x_1, \ldots, x_n]\). By the
function definitions we have

\[
isort [x_1, \ldots, x_n] = \begin{cases} 
\text{ins}(x_1, \text{isort} [x_2, \ldots, x_n]) & \text{by def of isort} \\
\text{ins}(x_1, \text{isort2} [x_2, \ldots, x_n]) & \text{by IH} \\
\text{isort2} [x_1, x_2, \ldots, x_n] & \text{by def of isort2}
\end{cases}
\]

**Permutations**

Here are some SML functions to help understand permutations and explore
their properties:

\[
\begin{align*}
\text{mem} &: \text{int} \times \text{int list} \rightarrow \text{bool} \\
\text{del} &: \text{int} \times \text{int list} \rightarrow \text{int list} \\
\text{perm} &: \text{int list} \times \text{int list} \rightarrow \text{int list} \rightarrow \text{bool}
\end{align*}
\]

and their specifications:

\[
\begin{align*}
\text{fun mem}(x:int, [ ]) &= \text{false} \\
\text{| mem}(x, y::L) &= (x=y) \text{ or else mem}(x,L) \\
\text{(* ENSURES mem}(x,L) = \text{true if } x \text{ occurs in } L, \text{ false otherwise *)}
\end{align*}
\]

\[
\begin{align*}
\text{fun del}(x, y::R) &= \text{if } x=y \text{ then } R \text{ else } y::\text{del}(x,R) \\
\text{(* REQUIRES } x \text{ occurs in } L \text{ *)} \\
\text{(* ENSURES del}(x,L) = \text{a list containing all items in } L \text{ except for} \\
\text{the first occurrence of } x \text{ *)}
\end{align*}
\]

\[
\begin{align*}
\text{fun perm}([ ], [ ]) &= \text{true} \\
\text{| perm}(_:_, [ ]) &= \text{false} \\
\text{| perm}([ ], _) &= \text{false} \\
\text{| perm}(x::L, R) &= \text{mem}(x, R) \text{ and also perm}(L, \text{del}(x,R)) \\
\text{(* ENSURES perm}(L,R) = \text{true if } L \text{ is a permutation of } R, \text{ false otherwise *)}
\end{align*}
\]
5 Mergesort

To mergesort a list of integers, if it is empty or a singleton do nothing (it’s already sorted); otherwise split the list into two lists of roughly equal length, mergesort these two lists, then merge these two sorted lists.

Obviously we need helper functions for splitting and merging.

(* split : int list -> int list * int list *)
(* REQUIRES true *)
(* ENSURES split(L) = a pair (A, B) of lists such that *)
(* length(A) and length(B) differ by at most 1 *)
(* and A@B is a permutation of L. *)

fun split [ ] = ([ ], [ ])
| split [x] = ([x], [ ])
| split (x::y::L) = let val (A, B) = split L in (x::A, y::B) end

Example: split [1,2,3,4,5] = ([1,3,5],[2,4]).

We can prove that split meets its specification, by induction on the length of the list being split. In the proof we appeal to some obvious facts about permutations.

• For L of length ≤ 1 the result holds obviously.

• Let L be a list of length n > 1. Then we can express L in the form x::y::R for some integers x and y and a list R of shorter length than n. By induction hypothesis, split R evaluates to a pair of lists (A,B) such that length(A) and length(B) differ by at most 1, and A@B is permutation of R. But then split L = (x::A,y::B), and these two lists have the same length difference as A and B; and (x::A)::(y::B) is a permutation of x::y::(A@B), hence also a permutation of L.

• That completes the proof.

Note that the spec is a bit more precise than the informal English that we used to introduce the algorithm.

The helper function for merging is only going to be used on a pair of sorted lists, and is used to produce another sorted list containing all of the items in both of the input lists. However, it would be terribly inefficient to call the sorted function here; we don’t need to check that the inputs
are sorted lists, provided we prove that the merge function only ever gets
applied to pairs of sorted lists. And we can design the merge function so
that it automatically builds a sorted result, so again there is no need to
verify this property explicitly by calling sorted!

(* merge : int list * int list -> int list *)
(* REQUIRES A and B are sorted lists of integers *)
(* ENSURES merge(A, B) = a sorted permutation of A@B *)

fun merge ([ ], B) = B
| merge (A, [ ]) = A
| merge (x::A, y::B) = case compare(x,y) of
   LESS => x :: merge(A, y::B)
| EQUAL => x::y::merge(A, B)
| GREATER => y :: merge(x::A, B)

Examples:

merge([1,3,5], [2,4]) = [1,2,3,4,5]
merge([5,3,1], [2,4]) = [2,4,5,3,1]

We prove that merge meets its spec by induction on the sum of the lengths of
A and B. This strategy should work, because in each recursive call the length
of at least one of the two lists decreases by 1 (and the other one is either
the same as before or shorter); in all cases the sum of the two list lengths
is smaller. Here are the proof details. You might expect us to use as the
"base case" in this proof the case where the sum of the list lengths is 0, i.e.
when both lists are empty. However, because the of the way the function
is written, it is actually simpler to base our inductive case analysis on the
function definition, as follows.

Note that the spec asserts that merge(A,B) is equal to a sorted perm of
A@B. This is the same as asserting that merge(A,B) evaluates to a sorted
perm of A@B.

• Assume that A and B are sorted lists.
• If A or B is the empty list, then merge(A,B) returns B or A, respectively.
  In both cases the result is sorted, and since [ ]@B = B and A@[ ] = A in
each case the result is equal to (and thus a permutation of) A@B.
In this section, we will prove that the `merge` function satisfies the specification on all pairs of sorted lists whose length sum is smaller than that of A and B. Let \( A = x :: A' \) and \( B = y :: B' \). Just like the function definition (third clause), our proof branches on the result of comparing \( x \) and \( y \).

- If \( x < y \), then \( \text{merge}(A, B) = x :: \text{merge}(A', B) \). By assumption that \( x :: A' \) is sorted, \( x \) is \( \leq \) every item in \( A' \). And by assumption that \( A \) is sorted, so is \( A' \). The length of \( A' \) is one less than the length of \( A \). So by the induction hypothesis, \( \text{merge}(A', B) \) evaluates to a sorted list (say \( M \)) that is a permutation of \( A' @ B \). We have \( \text{merge}(A, B) = x :: M \). Since \( x < y \) and we assumed that \( y :: B' \) is sorted, \( x \) is \( \leq \) every item in \( B \). So \( x \) is \( \leq \) every item in \( M \). Hence \( x :: M \) is a sorted permutation of \( x :: (A' @ B) \).

- The case analysis for when \( x = y \) or \( x > y \) is similar and we leave these cases as an Exercise.

We covered pairs in which one (or both) of the lists is empty in the first case; and the inductive step covers cases where both of the lists are non-empty. Thus we have shown by induction that for all sorted lists \( A, B \), \( \text{merge}(A, B) \) evaluates to a sorted permutation of \( A @ B \).

Now we have the ingredients, we can define a `mergesort` function:

```ml
(* msort : int list -> int list *)
(* REQUIRES true *)
(* ENSURES msort(L) = a sorted permutation of L *)

fun msort [] = []
| msort [x] = [x]
| msort L =
  let
    val (A, B) = split L
  in
    merge(msort A, msort B)
  end
```

Note how closely the function definition resembles the informal algorithm description!
We can now prove by induction on the length of $L$ that $\text{msort}$ meets this specification, for all integer lists $L$. Of course, we will make use here of the results (already established) that $\text{split}$ and $\text{merge}$ satisfy their specifications.

- **Base case:** When $L$ is empty or a singleton list, $\text{msort} \ L = L$, and this is trivially a sorted list and a permutation of $L$.

- **Inductive step:** Assume that $L$ is a list of length $n > 1$ and that $\text{msort}$ satisfies the spec for lists of length less than $n$. From above, we know that $\text{split} \ L$ evaluates to a pair of lists $(A, B)$ such that $0 \leq \text{length}(A) - \text{length}(B) \leq 1$ and $A \@ B$ is a permutation of $L$. Hence, the length of $A$ and length of $B$ are both less than length of $L$. (The maximum possible length for $A$ is $n \div 2$ if $n$ is even, $n \div 2 + 1$ if $n$ is odd, and since $n > 1$ in each case this is less than $n$.) Hence by the induction hypothesis, $\text{msort} \ A$ evaluates to a sorted permutation of $A$, and $\text{msort} \ B$ evaluates to a sorted permutation of $B$. Then by the spec for $\text{merge}$, $\text{merge}(\text{msort} \ A, \text{msort} \ B)$ evaluates to a sorted permutation of $A \@ B$, which must also be a permutation of $L$. We use here some obvious properties of permutations, such as “a permutation of a permutation is a permutation”.

- **That completes the proof.**

Just for interest again, consider the following slight variant:

```haskell
fun msort' [ ] = [ ]
| msort' L =
  let
    val (A, B) = split L
  in
    merge(msort' A, msort' B)
  end
```

If we drop the singleton clause, like this, we get a function that loops on lists of length 1. Hence it also loops on any non-empty list. See where the above proof goes wrong if we try to use it to prove this code correct.
6 The Joy of Specs

The mergesort example shows the benefits of designing helper functions with clear specifications, chosen carefully to make appropriate assumptions about the arguments to which the functions will be applied, and to make strong enough assertions about the results produced by these functions.

To illustrate the potential problems caused by inappropriate helper specs, note that the merge function also satisfies the specification:

For all integer lists L and R,

\[
\text{merge}(L,R) \text{ evaluates to a permutation of } L \oplus R.
\]

This spec is not strong enough to help prove that \texttt{msort} sorts.

Note also that we can replace \texttt{split} by any other function with the same type that satisfies the same specification as we used above, without affecting the correctness of \texttt{msort} (defined as above, but using the replacement \texttt{split} function). The new split function doesn’t need to be extensionally equivalent to the old one; it just needs to satisfy the same specification! For example we could have used

\[
\begin{align*}
\text{fun split } & \text{ [ ] } = (\text{ [ ] }, \text{ [ ] })
| \text{ split } [x] & = (\text{ [ ] }, [x])
| \text{ split } (x::y::L) & = \text{ let val } (A, B) = \text{ split } L \text{ in } (x::A, y::B) \text{ end}
\end{align*}
\]

Exercises

- Check that this split function satisfies the same specification as before.

- Show that this function is not extensionally equivalent to the original \texttt{split}. (Hint: what is \texttt{split [1,2,3]}?)

- For which integer lists \texttt{L} is it true that the two split functions produce the same result?
7 Work of msort

The work (sequential running time) of msort(L) depends on the length of L, but not on the integers that occur in L. We can derive, from the function definition, a recurrence relation for the work \( W_{\text{msort}}(n) \) of msort(L) when L has length \( n \). To get an asymptotic estimate of the work for msort, we must also analyze the work of split and merge.

It seems pretty obvious that split(L) looks at each item in L successively, dealing them out into the left- or right-hand component of the pair of lists being constructed. So \( W_{\text{split}}(n) \) is \( O(n) \). We can reach the same conclusion by extracting a recurrence relation from the definition of split:

\[
W_{\text{split}}(0) = c_0 \\
W_{\text{split}}(1) = c_1 \\
W_{\text{split}}(n) = c_2 + W_{\text{split}}(n - 2) \quad \text{for } n > 1
\]

for some constants \( c_0, c_1, c_2 \). It is easy to show that the solution \( W_{\text{split}}(n) \) to this recurrence relation is \( O(n) \).

Similarly, when A and B are lists of length \( m \) and \( n \), the running time of merge(A, B) is linear in \( m + n \). (The output list has length \( m + n \).)

Apart from the empty and singleton cases, msort(L) first calls split(L), then calls msort recursively twice, each time on a list of length about half of the original list’s length, then calls merge on a pair of lists whose lengths add up to length(L). Hence, the work of msort on a list of length \( n \) is given inductively by:

\[
W_{\text{msort}}(0) = k_0 \\
W_{\text{msort}}(1) = k_1 \\
W_{\text{msort}}(n) = k_2 n + 2W_{\text{msort}}(n \text{ div } 2) \quad \text{for } n > 1
\]

for some constants \( k_0, k_1, k_2 \). Using the table of standard solutions, it follows that \( W_{\text{msort}}(n) \) is \( O(n \log n) \). So the work for msort on a list of length \( n \) is \( O(n \log n) \).
8 Self-test 6

1. When the expression \texttt{ins}(3, \texttt{ins}(2, \texttt{ins}(1, []))) gets evaluated, in what order do the insertions occur?

2. What kind of arguments to \texttt{ins} cause the evaluation of \texttt{ins}(x,L) to take the most steps? These are called \textit{worst-case} arguments for \texttt{ins}.

3. Give a recurrence for the worst-case sequential runtime of \texttt{ins}(x,L) when L is a list of length n. Solve the recurrence and give an asymptotic classification.

4. What kind of list arguments are worst-case for \texttt{isort}(L)?

5. Estimate the worst-case sequential runtime for \texttt{isort}(L) when L is a list of length n.

6. The definition of \texttt{perms} given earlier has 4 clauses. Does it make any difference to the applicative behavior of the function if we change the clause order?

7. Rewrite the \texttt{perms} function to use fewer than 4 clauses.

8. Write an ML function

\[
\texttt{split} : \text{int list} \rightarrow \text{int list} \times \text{int list}
\]

such that for all integer lists L, \texttt{split}(L) evaluates to a pair of lists \((A, B)\) with \(A@B = L\) and \(|\text{length}(A) - \text{length}(B)| \leq 1\).

9. Would the code for \texttt{msort} work correctly (i.e. satisfy the sorting specification) if we replace its \texttt{split} function with the one from the previous question? Explain why, briefly.