15-150 Fall 2013

Stephen Brookes

Lecture 6
Thursday, 12 September
Sorting lists of integers

1 Outline

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

2 Remarks

In the code below we provide clear and accurate specifications but we don't always use the REQUIRES and ENSURES format. In class we followed the recommended style.

3 Background

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

(antisymmetry) $x \leq y$ & $y \leq x$ implies $x = y$

(transitivity) $x \leq y$ & $y \leq z$ implies $x \leq z$

(totality) $x \leq y$ or $y \leq x$

A list of integers is $<$-*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 it in specifications!

```
(* sorted : int list -> bool *)
fun sorted [ ] = true
 |  sorted [x] = true
 |  sorted (x::y::L) = (compare(x,y) <> GREATER) andalso sorted(y::L);

(* ENSURES sorted L = true iff L is <-sorted. *)
```

Examples:

```
sorted [1,2,3] = true
sorted [3,2,1] = false
```

Make sure you see the relevance of the linear ordering properties here: they are the reason why this `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 `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 `sorted(L) = true` if every item in `L` is $\leq$ all later items in `L`, and `sorted(L) = false` otherwise. Indicate clearly in your proof when you appeal to any of the above properties of linear orderings.

4 Insertion sort

Here is a function that implements *insertion sort*. 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*. A list `A` is a permutation of a list `B` if `A` contains the same items as `B`, possibly in a different order.

```
(* 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 `ins` as a helper, we can implement insertion sort as follows:

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

fun isort [ ] = [ ]
 |  isort (x::L) = ins (x, isort L)
```

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

```
(* isort : int list -> int list *)
fun isort [ ] = [ ]
 |  isort [x] = [x]
 |  isort (x::L) = ins (x, isort L)
```

Exercise: Show that this function is extensionally equivalent to the previous one. (So the singleton clause is irrelevant.)

In the lecture slides and on the blackboard we sketched a proof that `ins` satisfies its specification, and that `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.

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`.

- Inductive step: suppose that A and B are non-empty lists, and that merge 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 $x < y$, $\texttt{merge}(A, B) = x :: \texttt{merge}(A', B)$. 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, $\texttt{merge}(A', B)$ evaluates to a sorted list (say M) that is a permutation of A'@B. We have $\texttt{merge}(A, B) = x::M$. 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, $\texttt{merge}(A, B)$ evaluates to a sorted permutation of A@B.

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

```
(* 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 msort meets this specification, for all integer lists L. Of course, we will make use here of the results (already established) that split and merge satisfy their specifications.

- Base case: When L is empty or a singleton list, 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 `msort` satisfies the spec for lists of length less than n. From above, we know that `split L` evaluates to a pair of lists (`A`, `B`) such that $0 \leq$ `length(A)` $-$ `length(B)` ≤ 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, `msort A` evaluates to a sorted permutation of `A`, and `msort B` evaluates to a sorted permutation of `B`. Then by the spec for `merge`, `merge`(`msort A`, `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:

```
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,
     merge(L,R) evaluates to a permutation of L@R.
```

This spec is not strong enough to help prove that `msort` sorts.

Note also that we can replace `split` by any other function with the same type that satisfies the same specification as we used above, without affecting the correctness of `msort` (defined as above, but using the replacement `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

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

Exercise: check that this alternative split function satisfies the same specification as before. Show why this function is not extensionally equivalent to the original `split`.

7 Work of `msort`

The work (running time) of `msort(L)` depends on the length of `L`. We can derive, from the function definition, a recurrence relation for the work $W_{\texttt{msort}}(n)$ 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`.

Intuitively, `split(L)` has to look at each item in `L`, successively, dealing them out into the left- or right-hand component of the output list. So $W_{\texttt{split}}(n)$ is $\texttt{O}(n)$. We can reach the same conclusion by extracting a recurrence relation from the definition of `split`:

$$\begin{aligned} W_{\texttt{split}}(0) &= c_0 \\ W_{\texttt{split}}(1) &= c_1 \\ W_{\texttt{split}}(n) &= c_2 + W_{\texttt{split}}(n-2) \quad \text{for } n > 1 \end{aligned}$$

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

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

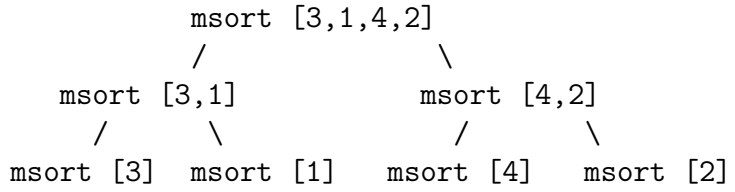
add up to `length(L)`. Hence, the work of `msort` on a list of length n is given inductively by:

$$\begin{aligned} W_{\texttt{msort}}(0) &= k_0 \\ W_{\texttt{msort}}(1) &= k_1 \\ W_{\texttt{msort}}(n) &= k_2 n + 2W_{\texttt{msort}}(n \texttt{ div } 2) \quad \text{for } n > 1 \end{aligned}$$

for some constants k_0, k_1, k_2. Using the table of standard solutions, it follows that $W_{\texttt{msort}}(n)$ is $\texttt{O}(n \ log\, n)$. So the work for `msort` on a list of length n is $\texttt{O}(n \ log\, n)$.

8 Span of `msort`

What about the span? Although we coded this function up using sequential constructs of ML, you might think that the mergesort function is well suited for parallelism, because it makes two *independent* recursive calls on lists of half the length, used to build the two components of a pair. Hence calling `msort(L)` gives rise to a tree-shaped pattern of recursive calls, e.g.



and in general the height of this call tree is $O(\log n)$, where n is the length of the original list. So maybe mergesort has logarithmic span?

Unfortunately, no! First we use `split` to deal the list out into two piles. There is no parallelism here, since we deal the elements out one by one, so we have to wait at least $\texttt{O}(n)$ timesteps in this phase, even if we have as much computational power as we need. This is bad. So the span of `split(L)` is linear in the length of `L`.

For the same reason, the recurrence for the span of `split` is the same as the recurrence for the work of `split`, because the function is inherently sequential:

$$S_{\texttt{split}}(n) \ = \ c + S_{\texttt{split}}(n-2) \text{ for } n > 1$$

for some constant c. Thus, $S_{\texttt{split}}(n)$ is $\texttt{O}(n)$.

Similarly, since `merge` is inherently sequential, the span of `merge` is (like the work) linear in the sum of the lengths of the lists.

However, for `msort` we get, for $n \geq 2$,

$$\begin{aligned} S_{\texttt{msort}}(n) &= S_{\texttt{split}}(n) + max(S_{\texttt{msort}}(n \texttt{ div } 2), S_{\texttt{msort}}(n \texttt{ div } 2)) + S_{\texttt{merge}}(n) \\ &= S_{\texttt{split}}(n) + S_{\texttt{msort}}(n \texttt{ div } 2) + S_{\texttt{merge}}(n) \\ &= cn + S_{\texttt{msort}}(n \texttt{ div } 2) \end{aligned}$$

for some constant c. We use *max* here because the two recursive calls are independent (the component expressions in a pair), and can be calculated in parallel; since the recursive calls are both on lists of approximately half the length, we end up counting just one of them towards the span. We do need the additive terms for the span of split and the span of merge, because of the data dependencies: first the split happens, then the two parallel sorts, then the merge.

Expanding, we see that:

$$\begin{aligned} S_{\texttt{msort}}(n) &= cn + S_{\texttt{msort}}(n \texttt{ div } 2) \\ &= cn + cn/2 + cn/4 + cn/8 + cn/16 + \cdots + cn/2^{log_2\, n} \\ &= cn(1 + 1/2 + 1/2^2 + \cdots + 1/2^{log_2\, n}) \\ &\leq 2cn \end{aligned}$$

The series sum here is always less than 2, and in fact converges to 2 as n tends to infinity. (This might seem to be related to Zeno's paradox, but is based on firm mathematical foundations; technically, $\sum_{n=0}^{\infty} \frac{1}{2^n} = 2$.) So the span of `msort` is therefore $\texttt{O}(n)$.

This is less than ideal. Ignore the constant factors, because a similar example can be chosen no matter what they are. Suppose you want to sort a billion numbers on 64 processors. Note that $\log 10^9$ is about 30, so the total work to do here is 30 billion steps. On 64 processors, this should take less than half a billion timesteps, if you divide the work perfectly among all 64 processors. However, our span estimate says that the length of the longest critical path is still a billion, so you can't actually achieve this division of labor! This problem gets worse as the number of processors gets larger.

The real issue here is that *lists are bad for parallelism.* The list data structure does not admit an efficient enough implementation of split and merge to exploit all the parallelism that might have been available.

In the next lecture we will discuss a more suitable data structure for parallel sorting.