15–150: Principles of Functional Programming

Sorting Integer Lists

Michael Erdmann*

Fall 2024

1 Background

Let's define what we mean for a list of integers to be sorted, by reference to datatypes and comparison functions in SML.

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

(This type is predefined in SML.)

```
(* Comparison for integers *)

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

(This function is predefined in SML as `Int.compare`.)

In this document, we will say that a list of integers is *sorted* if each item in the list is no greater than all items that occur later in the list. Here is an SML function that checks for this property.

```
(* sorted : int list -> bool
   REQUIRES: true
   ENSURES:  sorted(L) evaluates to true if L is sorted and to false otherwise.
*)
fun sorted [ ] = true
  | sorted [x] = true
  | sorted (x::y::L) = (compare(x,y) <> GREATER) andalso sorted(y::L)
```

Note: Technically we should say that a list is *<-sorted* since we have placed an ordering on the integers using the < comparator. Unless otherwise specified, we will mean "<-sorted" when we say or write "sorted". Most generally, we simply expect a `compare` function.

*Adapted from a document by Stephen Brookes.

2 Insertion Sort

As a reminder, in the last lecture we looked at the following code implementing *insertion sort*:

(We assume the reader knows what it means for one list to be a permutation of another.)

First, we define a helper function for inserting an integer into its proper place in a sorted list, then we write the main function.

```
(* ins : int * int list -> int list
   REQUIRES: L is sorted
   ENSURES:  ins(x, L) evaluates to 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)

(* isort : int list -> int list
   REQUIRES: true
   ENSURES: isort(L) evaluates to a sorted permutation of L.
*)
fun isort [ ] = [ ]
  | isort (x::L) = ins (x, isort L)
```

Work of insertion sort: Recall that evaluation of `ins(x,L)` requires $O(n)$ work, with n the length of `L`, and evaluation of `isort(L)` requires $O(n^2)$ work.

3 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. We will use helper functions for splitting and merging:

3.1 split

```
(* split : int list -> int list * int list
   REQUIRES: true
   ENSURES:  split(L) evaluates to a pair of lists (A, B) 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 prove that `split` meets its specification as follows:

Theorem 1 *For any value* `L : int list`*,* `split(L)` *returns a pair of lists* `(A,B)`*, differing in length by at most 1, such that* `A@B` *is a permutation of* `L`*.*

Proof: By a variant of the standard structural induction template for lists, on the variable `L`.

The informal intuition is: Instead of peeling off one element at a time, we will peel off two elements. We then need two base cases, one to support the induction for lists of even length and another to support the induction for lists of odd length.

BASE CASE 0: `L = [ ]`.

We need to show that `split([])` returns a pair of lists whose lengths differ by at most 1, which when appended together produce a permutation of `[]`.

Showing: Observe that `split([ ]) ==> ([ ], [ ])`
and that `[ ] @ [ ] ==> [ ]`.

That establishes this base case.

BASE CASE 1: `L = [x]` for some `x : int`.

We need to show that `split([x])` returns a pair of lists whose lengths differ by at most 1, which when appended together produce a permutation of `[x]`.

Showing: Observe that `split([x]) ==> ([x], [ ])`.
and that `[x] @ [ ] ==> [x]`.

That establishes this base case.

INDUCTIVE CASE: `L = x::y::L'`.

Induction Hypothesis: `split(L')` returns a pair of lists `(A',B')`, differing in length by at most 1, such that `A' @ B'` is a permutation of `L'`.

Need to show: `split(L)` returns a pair of lists `(A,B)`, differing in length by at most 1, such that `A@B` is a permutation of `L`.

Showing: (with some variable renaming in the code for clarity)

```
    split(L)
=   split(x::y::L')
==> let val (A', B') = split L' in (x::A', y::B') end
```

Write `A = x::A'` and `B = y::B'`.

The lists `A` and `B` differ in length by at most 1 since they are, respectively, one longer than the lists `A'` and `B'`, which differ in length by at most 1, by the induction hypothesis.

Again by the induction hypothesis, `A'@B'` is permutation of `L'`. The list `(x::A')@(y::B')` is a permutation of `x::y::(A'@B')`, and is therefore a permutation of `x::y::L'`, that is, of `L`. □

3.2 merge

The helper function for merging is only going to be called on a pair of sorted lists, producing another sorted list containing all of the items in both of the input lists.

```
(* merge : int list * int list -> int list
   REQUIRES:  A and B are sorted lists.
   ENSURES:   merge(A,B) evaluates to 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))
```

We prove that `merge` meets its specification as follows:

Theorem 2 *For all sorted lists* `A` *and* `B`*,* `merge(A,B)` *returns a sorted permutation of* `A@B`*.*

(Implicitly, we assume that `A` are `B` values of type `int list`.)

Proof: By strong induction on the product of the lengths of `A` and `B`, which we will write as $|\texttt{A}||\texttt{B}|$.

BASE CASE: $|\texttt{A}||\texttt{B}| = 0$.

Then at least one of $|\texttt{A}|$ and $|\texttt{B}|$ is 0. We need to show that `merge(A,B)` returns a sorted permutation of `A@B`.

- If $|\texttt{A}| = 0$, then `A` must be the empty list so:

  ```
  merge(A,B) = merge([ ],B) ==> B
  ```

  which is a sorted permutation of `A@B` since

  ```
  A@B = [ ]@B ==> B
  ```

  and since `B` is sorted.

- The case $|\texttt{B}| = 0$ is similar.

INDUCTIVE CASE: $|\texttt{A}||\texttt{B}| > 0$.

Induction Hypothesis: For all `A'` and `B'` such that $0 \leq |\texttt{A'}||\texttt{B'}| < |\texttt{A}||\texttt{B}|$, `merge(A',B')` returns a sorted permutation of `A'@B'`.

Need to show: `merge(A,B)` returns a sorted permutation of `A@B`.

Showing: Since $|\texttt{A}||\texttt{B}| > 0$, neither `A` nor `B` is the empty list. Write `A` = `x::A'` and `B` = `y::B'`. The third clause of `merge` is relevant.

- Suppose `compare(x,y) ==> LESS`.

  Since `A` is sorted, `A'` is also sorted. The length of `A'` is one less than the length of `A`, so $|\texttt{A'}||\texttt{B}| < |\texttt{A}||\texttt{B}|$. Thus (with some variable renaming in the code for clarity):

```
    merge(A,B)
==> x :: merge(A', y::B')
==> x :: L, with L a sorted permutation of A'@(y::B'), by IH.
```

Since `A` is sorted, `x` $\leq$ `a` for every element `a` in `A'`, and since `B` is sorted, `y` $\leq$ `b` for every element `b` in `B'`. Thus by transitivity (don't forget that `x` $<$ `y`), we see that `x` $\leq$ `z` for every element `z` in `L`. We see therefore that `x::L` is a sorted permutation of `A@B`, as desired.

- The cases for which `compare(x,y)` evaluates to either `EQUAL` or `GREATER` are similar.

□

3.3 msort

Given these ingredients, we may now define a mergesort function:

```
(* msort : int list -> int list
   REQUIRES: true
   ENSURES:  msort(L) evaluates to 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
```

We prove that `msort` meets its specification as follows:

Theorem 3 *For any value* `L : int list`*,* `msort(L)` *returns a sorted permutation of* `L`*.*

Proof: By a variant of strong induction on m, the length of `L`. (Comment: We could try a variant of structural induction on `L` that is similar to strong induction. However, it is more convenient here to work with the size of `L`, in order to avoid looking into the internals of the code for `split`.)

BASE CASE 0: $m = 0$.
In this case, `L = [ ]`. We need to show `msort [ ]` is a sorted permutation of `[ ]`.
Showing: `msort [ ] ==> [ ]`, which is a sorted permutation of `[ ]`.

BASE CASE 1: $m = 1$.
In this case, `L = [x]`, for some integer `x`. We need to show that `msort [x]` is a sorted permutation of `[x]`.
Showing: `msort [x] ==> [x]`, which is a sorted permutation of `[x]`.

INDUCTIVE CASE: `L` has length $m > 1$.

Induction Hypothesis: `msort(L')` returns a sorted permutation of `L'` for every list `L'` of length less than m.

Need to show: `msort(L)` returns a sorted permutation of `L`.

Showing:

Theorem 1 tells us that `A` and `B` each have length no greater than $\frac{m+1}{2}$. That is strictly less than m, since $m > 1$. The induction hypothesis therefore tells us that `msort(A)` is a sorted permutation of `A` and `msort(B)` is a sorted permutation of `B`. By Theorem 2, `merge(msort A, msort B)` is therefore a sorted permutation of `(msort A) @ (msort B)`, which is itself a permutation of `A @ B`, as desired. □

4 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 $O(n)$. 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. This recurrence is very similar to recurrences we saw in Lecture 6, except that the recursive call is on size $n-2$ not $n-1$. That merely halves the total number of calls, but from a big-O perspective, the running time is linear in n. Indeed, one can prove by strong induction on n that $W_{\texttt{split}}$ is $O(n)$.

Similarly, when `A` and `B` are lists of length m and n, respectively, the worse-case running time of `merge(A,B)` is linear in $m+n$. (The merge peels off at least one element from each list at each step and may need to touch all elements.)

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:

$$\begin{aligned}
&W_{\texttt{msort}}(0) = k_0 \\
&W_{\texttt{msort}}(1) = k_1 \\
&W_{\texttt{msort}}(n) = k_2 + k_3 * n + 2 * W_{\texttt{msort}}(n \texttt{ div } 2) \quad \text{for } n > 1 \quad (^*)
\end{aligned}$$

for some constants k_0, k_1, k_2, and k_3. Using the table of standard solutions from Lecture 6, it follows that $W_{\texttt{msort}}(n)$ is $O(n \log n)$. So the work for `msort` on a list of length n is $O(n \log n)$.

(*) More precisely, we should write $W_{\texttt{msort}}(n - n \texttt{ div } 2) + W_{\texttt{msort}}(n \texttt{ div } 2)$, but for simplicity we think of that as $2 * W_{\texttt{msort}}(n \texttt{ div } 2)$.

For those of you interested in more of a justification, here is an instance of the "Tree Method" discussed in the last lecture notes:

We can view the work via a computation tree, as follows. This is for a perfectly balanced tree, which represents a worst case scenario. The constant k is some constant that is larger than any of the k_i above.

```
                    k*n
        k*n/2                   k*n/2
   k*n/4      k*n/4      k*n/4      k*n/4
    ...        ...        ...        ...
  k    k     k    k     k    k     k    k
```

Summing up the work at each level gives $k * n$. There are roughly $\log_2 n$ levels. So the overall work is $O(n \log n)$.

Alternatively, one can prove that the complexity estimate is correct by using induction to establish a more general form of the recurrence.

5 Span of `msort`

What about the span? Although we coded this function up using sequential constructs of SML, 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.,

```
              msort [3,1,4,2]
              /              \
        msort [3,1]       msort [4,2]
         /      \           /       \
    msort [3]   msort [1]  msort [4]   msort [2]
```

and in general the height/depth 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 $O(n)$ time steps 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:

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

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

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 $n \geq 2$,

$$\begin{aligned} S_{\texttt{msort}}(n) &= k + S_{\texttt{split}}(n) + \max(S_{\texttt{msort}}(n \texttt{ div } 2), S_{\texttt{msort}}(n \texttt{ div } 2)) + S_{\texttt{merge}}(n) \\ &= k + S_{\texttt{split}}(n) + S_{\texttt{msort}}(n \texttt{ div } 2) + S_{\texttt{merge}}(n) \\ &\leq cn + S_{\texttt{msort}}(n \texttt{ div } 2) \qquad \text{(for sufficiently large n)} \end{aligned}$$

for some constants k and 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 toward the span. Again, to be precise, we should write $\max(S_{\texttt{msort}}(n - n \texttt{ div } 2), S_{\texttt{msort}}(n \texttt{ div } 2))$, which would become $S_{\texttt{msort}}(\lceil \frac{n+1}{2} \rceil)$, but we regard that simply as $S_{\texttt{msort}}(n \texttt{ div } 2)$, for ease of presentation.

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:

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

The series sum here is always less than 2, and in fact converges to 2 as n tends to infinity. The span of `msort` is therefore $O(n)$. **Summary:** The $O(n)$ span is better than the $O(n \log n)$ work.

Although the span is better than the work, it is still not as good as one should expect. 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 $\log_2 10^9$ is about 30, so the total work to do here is roughly 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 could be available.

One can do much better with trees, by using the natural parallelism of trees for splitting and merging. We will discuss sorting on trees in the next lecture.