15-150 Fall 2013
Lab 3

11 September 2013

The goal for the third lab is to make you more comfortable writing functions that operate on lists, and doing asymptotic analysis and proofs.

Take advantage of this opportunity to practice writing functions and proofs with the assistance of the TAs and your classmates. Today in lab we encourage you to collaborate with your classmates and to ask the TAs for help.

Remember to follow the methodology for writing functions—specifications and tests are part of your code!

1 Introduction

1.1 Getting Started

Update your clone of the `git` repository to get the files for this weeks lab as usual by running

```
git pull
```

from the top level directory (probably named `15150`).

2 Append

The "cons" function `::` adds one new element to a list. What if you want to *append* a whole list onto the front of another? Appending a list `l1` to another list `l2` evaluates to a list that contains all of the elements of `l1` in the same order, followed by all of the elements of `l2`, also in the same order. For example,

```
append([1,2,3], [5,13,5]) =>* [1,2,3,5,13,5]
```

A simple implementation of `append` takes elements off of `l1` one at a time, consing them onto the result of appending the rest of the list to `l2`.

Task 2.1 Write the function

```
append : int list * int list -> int list
```

that behaves according to the specification given above.

Task 2.2 Write a recurrence relation for the work of `append`, in terms of the lengths of `l1` and `l2`. What is the O of this recurrence?

Solution 2.2

We take one element off of `l1` each time the function runs until `l1` is nil, doing a constant amount work in each function call.

Thus the recurrence for the work of `append` is:

$$W(0) = k_0$$
$$W(n) = k + W(n-1)$$

where n is the length of list `l1` and k_0, k are constants. From this recurrence we get the work of append is $O(n)$.

For future reference, note that append is built in to the basis library of SML in the form of the `@` operator. The expression `l1 @ l2` appends `l1` to `l2`.

3 List Reversal

Task 3.1 Define an ML function

```
reverse: int list -> int list
```

such that for all integer lists `L`, `reverse(L)` evaluates to the a list that contains all the elements of `L` in reverse order. Do not create any helper functions for this task.

Task 3.2 Write a recurrence relation for the work of `reverse` in terms the the length of `L`. What is the O of this recurrence?

> **Solution 3.2** We will define the recurrence on n where n is the length of the first argument to reverse.
>
> $$W_{reverse}(0) = k_0$$
> $$W_{reverse}(n) = k_1 + W_{reverse}(n-1) + W_{append}n - 1$$
>
> Because $W_{append}(n)$ is $O(n)$, we must perform linear work n times. Thus $W_{merge}(n) \in O(n^2)$.

Task 3.3 Define an ML function

```
reverse': int list -> int list
```

such that for all integer lists `L`, `reverse'(L)` evaluates to the a list that contains all the elements of `L` in reverse order. For this task you should define a helper function that will allow you to solve the problem in $O(n)$ work.

Note for future reference that the SML basis library has a built-in function `rev` to reverse a list.

4 Proving Termination

Consider the function `foo : int list -> int list` given by

```
fun foo (L: int list) : int list =
    case L of
        [ ] => [ ]
     |  (x::R) => x :: foo(rev R)
```

where `rev` is a given function of type `int list -> int list` such that for all integer lists `L`, `rev(L)` evaluates to the reverse of list `L`

Task 4.1 Prove the following theorem by induction on the length of `L`

Theorem: For all values `L : int list`, `foo(L)` terminates.

You may use the following lemmas as facts in your proof, but be sure to cite them.

Lemma 1: If `L : int list` and `L value`, then `L = [ ]` if `length(L) = 0` and `L = x::R` for some `x : int, R : int list` if `length(L) > 0`

Lemma 2: For all values `L : int list`, `rev L` terminates and `length(rev L) = length(L)`.

You may assume `length(L)` is defined by

```
fun length([ ]) = 0
  | length(x::R) = 1 + length(R)
```

Theorem:

Proof: By __________________ (method) on ___________________ (variable/type)

Base Case (   ):

NTS (Need to Show):

Inductive Step (   ):

IH (Inductive Hypothesis):

NTS (Need to Show):

$\boxed{\textbf{Solution 4.1}}$ We will prove the theorem by induction on the length of `L`

Base Case (`length(L) = 0`):

To Show: `foo(L)` terminates for all lists of length 0

By Lemma 1, since `length(L) = 0`, `L = [ ]`.

```
foo([ ])
⇒ * case [ ] of [ ] => [ ] | (x::R) => x ::  foo(rev R)
⇒ * [ ]
```

Since `foo (L)` evaluated to a value, this case is proven.

Inductive Step (`length(L) > 0`):

Inductive Hypothesis: Assume `foo(L')` terminates for all lists L' of length $n >= 0$

To Show: `foo(L)` terminates for all lists of length $n + 1$

Let `L` be a list of length $n + 1$.

As $n \geq 0$, $n + 1 \geq 1$. Then by Lemma 1, since `length(L) > 0`, `L = x::R` for some `x : int` and `R : int list`.

```
      foo(x::R)
 ⇒ *  case x::R of [ ] => [ ] | x::R => x ::  foo(rev R)
 ⇒ *  x ::  foo(rev R)
 ⇒ *  x ::  foo(R')
```

for some `R'` where `length(R')` = `length(R)` by Lemma 2

Note `length(x::R) = 1 + length(R)`. We also have `length(x::R) = n + 1` so `n + 1` = `1 + length(R)`. Thus subtracting 1 from both sides, we get `length(R) = n`.

This means by the IH, taking `L' = R'`, `foo(R')` terminates and

```
      x ::  foo(R')
 ⇒ *  x::R''          for some R''
```

Since we showed `foo(L)` evaluates to a value, this case is proven.

Have the TAs check your work in the previous section before proceeding.

5 Fibbing

As we saw in lecture, sometimes it is possible to speed up a computation by "accumulating" some extra information as we go. The Fibonacci sequence is defined by the following recurrence:

```
fib(0) = 1
fib(1) = 1
fib(n) = fib(n-2) + fib(n-1)    for n>1
```

And we saw in class that the obvious recursive ML function based on this recurrence has exponential runtime. We can compute Fibonacci numbers more efficiently if we calculate two consecutive numbers instead of just one at a time. If the pair $(\mathtt{x}, \mathtt{y})$ contains the $(n-1)^{th}$ and n^{th} Fibonacci numbers, we can easily see that the "next" such pair is $(\mathtt{y}, \mathtt{x}+\mathtt{y})$.

Task 5.1 Write an ML function

```
fibber : int -> int * int
```

such that for all $\mathtt{n} \geq 0$, $\mathtt{fibber}(\mathtt{n})$ returns the pair of integers equal to $(\mathtt{fib}(\mathtt{n}), \mathtt{fib}(\mathtt{n}+1))$. Your function should have *linear* running time, i.e. `fibber(n)` should take time proportional to `n`. [Do *not* use `fib`!]

Task 5.2 Derive a recurrence relation for $W_{\mathtt{fibber}}(n)$, the work (runtime) of $\mathtt{fibber}(n)$.

Solution 5.2

$$W_{fibber}(0) = k_0$$
$$W_{fibber}(1) = k_1$$
$$W_{fibber}(n) = k_2 + W_{fibber}(n-1)$$

Task 5.3 Find a closed form for $W_{\mathtt{fibber}}(n)$ and use it to confirm that your function does have linear runtime.

Solution 5.3 We notice that at each level of the recursion tree we perform a constant amount of work. When computing `fibber(n)` we have a total of n calls to `fibber`, giving us the following closed form.

$$\begin{aligned} W_{fibber}(n) &= \sum_{i=0}^{n} k' \\ &= k' \sum_{i=0}^{n} 1 \\ &= k'(n) \end{aligned}$$

As the total work is bounded by $k'(n)$ for some constant k' we have that $W_{fibber} \in O(n)$.

6 Merge

Task 6.1 Write a function

```
merge : int list * int list -> int list
```

that merges two sorted lists into one sorted list. You should assume that your input lists are sorted in increasing order, and the list you return should also be in increasing order.

Task 6.2 Write a recurrence relation for the work of `merge`, in terms of the lengths of `l1` and `l2`. What is the O of this recurrence?

Solution 6.2 We will define the recurrence on n where $n = |l_1| + |l_2|$ where $|x|$ is the length of the list.

$$W_{merge}(0) = k_0$$
$$W_{merge}(n) = k_1 + W_{merge}(n-1)$$

Similar to `fibber`, there are n levels in the work tree and a constant amount of work is done per level. Thus, the closed form of $W_{merge}(n) = nk'$ and thus $W_{merge}(n) \in O(n)$

7 More Functions on Lists

Task 7.1 Define an ML function

```
evens : int list -> int list
```

such that for all integer lists `L`, `evens(L)` returns the sublist of `L` consisting of the integers in `L` that are *even*, in the same order as they occur in `L`. For example:

```
evens [1,2,3,4] = [2,4]
evens [1,3,5,7] = [ ]
```

Solution 7.1 See solution in `lab03-sol.sml`.

Task 7.2 Write a function

```
bitAnd : int list * int list -> int list
```

such that if given two int lists that contain only 1s and 0s, `bitAnd` returns the bitwise 'and' of the two lists as if they were interpreted as bitstrings (that is, interpreted as if 1 means *true* and 0 means *false*).

Solution 7.2 See solution in `lab03-sol.sml`.

Task 7.3 Define an ML function

```
interleave : int list * int list -> int list
```

such that for all integer lists `A` and `B`, `interleave(A, B)` returns a list built by alternating the items in `A` and `B`, until reaching the end of one of the lists, after which we take the remaining items from the other list. For example,

```
interleave([2],[4]) = [2,4]
interleave([2,3],[4,5]) = [2,4,3,5]
interleave([2,3],[4,5,6,7,8,9]) = [2,4,3,5,6,7,8,9]
interleave([2,3],[ ]) = [2,3]
```

Solution 7.3 See solution in `lab03-sol.sml`.