15-150 Summer 2016
Lab 6

02 June 2016

1 Exam Review

This lab tries to be as comprehensive as possible in covering the topics you have learned in class so far. You should answer the questions on a sheet of paper.
You are encouraged to start working on problems you're most unsure about first.

1.1 Disclaimer!

The difficulty and length of this lab does *not* reflect the actual difficulty, length, or topic focus of the exam. The format of the questions however, will likely be similar to questions on the exam.

1.2 Exam Information

Time: Monday, Jun 6, 1:30-2:50 PM.
Location: Gates 4215

2 Short Answer (Values, Types, etc)

For each of the following expressions, state the most general type *and* syntactic value of the expression. If the expression is not well-typed or does not reduce, explain briefly why or why not.

(a) `1/2`

(b) `[]::[]`

(c) `"abcd"+"f"`

(d) `SOME NONE`

(e) `(fn a => a, fn b => b)`

(f) `(fn a => fn b => b a) 6`

(g) `(fn a => 1::a)`

(h) `(fn a => 1::a) [1]`

(i) `let fun f (x::L) = x in f "abc" end`

(j) `(op o)`

(k) `fun f f x = x f f`

For each of the following expressions, state the most general type. If the expression is not well-typed, briefly explain why or why not.

(l) `map filter`

(m) `filter map`

Solution 2.0

(a) Not well-typed, since `/` is type `Real -> Real`
(b) Type: `'a list list`. Value: `[[]]`
(c) Not well-typed, since `+` is `Int -> Int` or `Real -> Real`
(d) Type: `'a option option`. Value: `SOME NONE`
(e) Type: `('a -> 'a) * ('b -> 'b)`. Value: `(fn a => a, fn b => b)`
(f) Type: `(int -> 'a) -> 'a`. Value: `fn b => b 6`
(g) Type: `int list -> int list`. Value: `fn a => 1::a`

(h) Type: `int list`. Value: `[1, 1]`

(i) Not well-typed, since "abc" is a String and `f` is of type `'a list -> 'a`

(j) Type: `('a -> 'b) * ('c -> 'a) -> 'c -> 'b`. Value: `op o`

(k) Type: `'a -> ('a -> 'a -> 'b) -> 'b`. Value: `fun f f x = x f f`

(l) Type: `('a -> bool) list -> ('a list -> 'a list) list`

(m) Not well-typed, since first argument of `filter` takes a function that evaluates to a `bool`.

3 Currying

Write two total functions `curry`, `uncurry` with the most general types being

```
uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

```
curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

Solution 3.0 `fun uncurry f (a,b) = f a b`

```
fun curry f a b = f (a,b)
```

4 HOFs

Recall that the SML built-in functions, `foldl` and `foldr`, have the following type:

```
('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

However, they combine data in a different evaluation order:

```
foldr f b [x1,x2,x3,...,xn] = f (x1, f (x2, f (x3, ... f (xn,b)...)))
foldl f b [x1,x2,x3,...,xn] = f (xn, ... f (x3, f (x2, f (x1,b)...)))
```

Implement the following functions using only `foldl`, `foldr`, and any anonymous function. Your function must not be recursive. You may not use or define any other helper functions. You may use builtin operators such as `case`, `::`, `if`, `andalso`, `orelse`, but you may NOT use `@`.

(a)
```
(* reverse L evaluates to list L reversed *)
fun reverse (L : 'a list) :  'a list =
```

(b)
```
(* length L evaluates to length of L *)
fun length (L : 'a list) :  int =
```

(c)
```
(*find L evaluates to SOME x if x exists in L, NONE otherwise *)
fun find (L : ''a list) (x :  ''a) :  ''a option =
```

(d)
```
(* map f L evaluates to list with f applied to all elements in L,
* kept in same order *)
fun map (f :  'a -> 'b) (L : 'a list) :  'b list =
```

Solution 4.0

(a)
```
fun reverse L =
    foldl (fn (a,b) => a::b) [] L
```

(b)
```
fun length L =
    foldl (fn (a,b) => 1+b) 0 L
```

(c)
```
fun find x L =
    foldl (fn (a,b) => if (a=x) then SOME(x) else b) NONE L
```

(d)
```
fun map f L =
    foldr (fn (a,b) => (f a)::b) [] L
```

5 Tree Product (Proof, Big-O)

Recall the tree datatype:

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree
```

Consider this function, which computes the product of integers in a tree:

```
fun mult T =
  case T of
    Empty => 1 (* multiplicative identity *)
  | mult (Node(L,x,R)) => x * (mult L) * (mult R)
```

(a) Using induction, prove that `mult T` evaluates to the product of all elements in tree `T`.

(b) Given that `T` is a balanced `int tree` with depth `d`, write recurrences that will represent the work and span of evaluating `mult T`, in terms of `d`.

(c) Using the recurrences you found in part (b), what are the respective big-$\mathcal{O}$ bounds? Show your work.

Solution 5.0

(a) Prove by structural induction on T.
BC: When Empty, return 1 by mult definition.
IH: For some T=Node(L,x,R), assume claim holes for L and R.
IS: We want to show claim holds for T.
From function definition, mult T = x * (mult L) * (mult R).
By IH, we know mult L evaluates to product of nodes in L, and mult R evaluates to product of nodes in R.

(b) $W_{mult}(0) = c$, $W_{mult}(d) = 2 * W_{mult}(d-1) + k$.
$S_{mult}(0) = c$, $S_{mult}(d) = S_{mult}(d-1) + k$.

(c) Work in $\mathcal{O}(2^d)$. Span in $\mathcal{O}(d)$.

6 Fib (Proof, Big-O)

Consider the following two implementations of `fib`:

```
fun fib1 n =
  case n of
    0 => 1
  | 1 => 1
  | _ => fib1(n-1)+fib1(n-2)
```

and

```
fun fib2_helper n =
  case n of
    0 => (0,1)
  | _ => let val (n1,n2) = fib2_helper (n-1) in (n2,n1+n2) end

fun fib2 n =
  case n of
    0 => 1
  | 1 => 1
  | _ => let (_,x) = fib2_helper n in x end
```

(a) Prove that `fib1 n` $\cong$ `fib2 n`

(b) Find the big-$\mathcal{O}$ runtime for both of the functions. Show your work.

$\boxed{\textbf{Solution 6.0}}$ **Lemma A:** First, prove `fib1` is total by Strong Induction on `n`.

Base Case: When `n = 0` or `n = 1`, `fib1 0` and `fib1 1` evaluate to 1, which is a value.

Inductive Step: Need to show `fib1 n` evaluates to a value for `n` ≥ 2.

Induction Hypothesis: Assume that `fib1 k` evaluates to a value for all `k` $<$ `n`.

`fib1 n`	$\cong$ `fib1(n-1) + fib1(n-2)`	[By 3rd Clause of `fib1`]
	$\cong$ `k1 + k2`	[IH, `fib1(n-1), fib1(n-2)` evaluate to `k1, k2` respectively]
	$\cong$ `k`	[Rule of Addition Over `int`, `k1, k2` are values]

By the Base Case and Inductive Step, `fib1` is total.

Lemma B: Now prove the stronger claim that `(fib1(n-1), fib1(n))` $\cong$ `fib2_helper n` for `n` ≥ 1 via Strong Induction on `n`.

Base Case: When `n = 1`:

`fib2_helper 1` $\cong$ `let val (n1, n2) = fib2_helper 0 in (n2, n1+n2) end`
[By 2nd Clause of `fib2_helper`]
$\cong$ `let val (n1, n2) = (0, 1) in (n2, n1+n2) end`
[By 1st Clause of `fib2_helper`]
$\cong$ `(1, 0+1)`
[Applying `let` with bindings `n1 = 0, n2 = 1`]
$\cong$ `(1, 1)`
[Stepping]
$\cong$ `(fib1 0, fib1 1)`
[By 1st and 2nd Clauses of `fib1`]

Inductive Step: Need to show `(fib1(n-1), fib1(n))` $\cong$ `fib2_helper n` for `n` ≥ 2.

Induction Hypothesis: Assume `(fib1(k-1), fib1(k))` $\cong$ `fib2_helper k` for all `k` $<$ `n`.

`fib2_helper n` $\cong$ `let val (n1, n2) = fib2_helper(n-1) in (n2, n1+n2) end`
[By 2nd Clause of `fib2_helper`]
$\cong$ `let val (n1, n2) = (fib1(n-2), fib1(n-1)) in (n2, n1+n2) end`
[IH]
$\cong$ `(fib1(n-1), fib1(n-2)+fib1(n-1))`
[Applying `let` with bindings `n1 = fib1(n-2), n2 = fib1(n-1)`]
$\cong$ `(fib1(n-1), fib1(n))`
[By 3rd Clause of `fib1`]

By the Base Case and Induction Step, `(fib1(n-1), fib1(n))` $\cong$ `fib2_helper n` for `n` ≥ 1.

Now to prove the main claim.

When `n = 0`:

`fib1 0` $\cong 1$ [By 1st Clause of `fib1`]
$\cong$ `fib2 0` [By 1st Clause of `fib2`, Ref. Trans.]

When `n > 0`:

$$
\begin{aligned}
\texttt{fib2 n} &\cong \texttt{let (\_,x) = fib2\_helper n in x end} \\
&\text{[By 3rd Clause of \texttt{fib2}]} \\
&\cong \texttt{let (\_,x) = (fib1(n-1), fib1(n)) in x end} \\
&\text{[\texttt{Lemma B}]} \\
&\cong \texttt{let (\_,x) = (v2, v1) in x end} \\
&\text{[\texttt{Lemma A: v1 = fib1(n), v2 = fib1(n-1)}]} \\
&\cong \texttt{v1} \\
&\text{[Applying \texttt{let} with binding \texttt{x = v1}]} \\
&\cong \texttt{fib1 n} \\
&\text{[Ref. Trans. with \texttt{v1 = fib1 n}]}
\end{aligned}
$$

This completes the proof.

The recurrences for `fib1`, `fib2`, and `fib2_helper` are given below:

$$W_{\texttt{fib1}}(0) = k_0 \qquad W_{\texttt{fib1}}(1) = k_1$$

$$W_{\texttt{fib1}}(n) = k_2 + W_{\texttt{fib1}}(n-1) + W_{\texttt{fib1}}(n-2)$$

$$W_{\texttt{fib2}}(0) = c_0 \qquad W_{\texttt{fib2}}(1) = c_1$$

$$W_{\texttt{fib2}}(n) = c_2 + W_{\texttt{fib2\_helper}}(n)$$

$$W_{\texttt{fib2\_helper}}(0) = t_0 \qquad W_{\texttt{fib2\_helper}}(n) = t_1 + W_{\texttt{fib2\_helper}}(n-1)$$

We claim $W_{\texttt{fib1}}(n) \in \mathcal{O}(2^n)$. Note the following is true since the work for `fib1` is monotonically increasing with respect to `n`:

$$W_{\texttt{fib1}}(n) = k_2 + W_{\texttt{fib1}}(n-1) + W_{\texttt{fib1}}(n-2) \leq k_2 + 2 * W_{\texttt{fib1}}(n-1)$$

By the tree method (draw this out for practice!), every level has $2^i C$ work total, where C is a constant. Note that there are n levels.
Then, the total work of the tree is as follows:

$$\sum_{i=0}^{n-1} 2^i C = (2^n - 1)C \in \mathcal{O}(2^n)$$

Recall: You can imagine the summation above with a counting argument using an n-bit binary string.

We claim $W_{\texttt{fib2\_helper}}(n) \in \mathcal{O}(n)$.
Intuitively, we only have a linear number of calls, with constant work on each call. Hence the total work is linear.

By the tree method, we have n levels, each with constant work C, so the total work of the tree is $nC \in \mathcal{O}(n)$.

Hence it is clear that $W_{\texttt{fib2}}(n) \in \mathcal{O}(n)$.

Follow-up Questions: What are the recurrences and Big-$\mathcal{O}$ for the span of the functions above?

7 Binary Generation (HOFs, Continuation)

Given two non-negative integers `m` and `n`, we are interested in all the possible binary numbers that can be formed using `m` 1's and `n` 0's. We will represent binary numbers as an `int list` of 0's and 1's. (For example, the binary number 100 would be represented as `[1,0,0]`.)

(a) Define the following recursive helper function that returns an `int list` of length `d` that only contains `n`'s, given that `d` is non-negative. The type of `listOfNs` is `int -> int -> int list`.

```
fun listOfNs n d =
  case d of
    0 =>
  | d =>
```

(b) Define a recursive function `bingen` where `m` and `n` are defined as above. (For example, `bingen 1 2 =>* [[1,0,0],[0,1,0],[0,0,1]]`. You are allowed to use `listOfNs`, `map`, `::`, `@`, and any anonymous functions, but no other helpers. The type of `bingen` is `int -> int -> int list list`.

```
fun bingen m n =
  case (m,n) of
    (0, _) =>
  | (_, 0) =>
  | (_, _) =>
```

(c) Using continuation, implement function `bingenC` of type `int -> int -> (int list list -> 'a) -> 'a`, where `bingenC m n k` $\cong$ `k (bingen m n)`. You are allowed to use `listOfNs`, `map`, `::`, `@`, and any anonymous functions, but no other helpers.

```
fun bingen m n k =
  case (m, n) of
  | (0, _) =>
  | (_, 0) =>
  | (_, _) =>
```

Solution 7.0

(a)
```
fun listOfNs n 0 = []
  | listOfNs n d = n::(listOfNs n (d-1))
```

(b)
```
fun bingen 0 n = [listOfNs 0 n]
  | bingen m 0 = [listOfNs 1 m]
  | bingen m n = (map (fn a => 1::a) (bingen (m-1) n))@
                 (map (fn b => 0::b) (bingen m (n-1)))
```

(c)
```
fun bingenC 0 n k = k [listOfNs 0 n]
  | bingenC m 0 k = k [listOfNs 1 m]
  | bingenC m n k =
        bingenC (m-1) n (fn x => bingenC m (n-1)
        (fn y => k ((map (fn b => 1::b) x) @ (map (fn a => 0::a) y))))
```