CS312 Preliminary Exam 1
Cornell University, Spring 2005 March 10, 2005

Solutions

1. Short Answers [15 pts] (parts a–???)

   Briefly answer each of the questions below.

   (a) [3 pts] What are polymorphic functions and what is the advantage of using them?

   Answer: *Polymorphic functions are functions with arguments of arbitrary types. They allow code reuse, since the same function can be used on values of different types.*

   (b) [3 pts] Can different cases in a `case` construct evaluate to expressions of different types? If yes, show an example. If not, explain why.

   Answer: *No. The compiler must figure out a type for the* `case` *construct; since it cannot tell statically which case is going to be executed, all cases must have the same type.*

   (c) [3 pts] For each of the following specifications, indicate if it specifies a precondition, a post-condition, none, or both:

The `Requires` clause:	pre-condition
The `Checks` clause:	pre-condition
The representation invariant:	pre- and post-condition
The `Returns` clause:	post-condition

   (d) [3 pts] Can a curried function `foo x y z = x + z + y` be invoked with parameters `x` and `z` only? Explain briefly.

   Answer: *No, if a curried function is invoked with a subset of its parameters, those must be its first parameters. The first two parameters of* `f` *are* `x` *and* `y`*, not* `x` *and* `z`*.*

   (e) [3 pts] Can a call to an $O(n)$ function run much slower than a call to a $\Theta(n^2)$ function, for the same input? Explain briefly.

   Answer: *Yes, asymptotic complexity gives guarantees about the order of growth for large n's. It doesn't restrict the running times for small values of n, so an $O(n)$ function can run (much) slower than a $\Theta(n^2)$ function for small n's.*

2. Folding Lists [10 pts] (parts a–???)

   Write the following functions using `foldl`. Remember that function `foldl` is defined as follows:

```
fun foldl (f:'a*'b -> 'b) (v:'b) (l:'a list):'b =
  case l of [] => v
          | h::t => foldl f (f(h,v)) t
```

(a) [5 pts] Write function `alt` that takes a list $[a_1, a_2, a_3, ..., a_n]$ of integers and computes $a_1 - a_2 + a_3 - ... + (-1)^{n+1}a_n$. For instance:

```
alt([2,1,4,2]) = 3
```

Answer:

```
fun alt(l:int list):int =
  #1 (foldl (fn (x, (acc,b)) =>
             (if b then acc+x else acc-x, not b))
      (0,true) l)
```

(b) [5 pts] Write function `diff` that takes a list $[a_1, a_2, a_3, ..., a_n]$ of integers and produces a list $[a_1, a_2 - a_1, a_3 - a_2, ..., a_n - a_{n-1}]$ whose elements are differences btween the elements of the original list, except for the leftmost element, which remains unchanged. For instance:

```
diff([4,1,2,4]) = [4,~3,1,2]
```

Answer:

```
fun diff(l:int list):int list =
  rev (#1 (foldl (fn (x, (acc,last)) => (x-last::acc, x))
           ([], 0) l))
```

3. Zardoz [23 pts] (parts a–???)

For each of the following expressions, fill in the box with a *value* (not an expression) that causes each of the programs to evaluate to 42. Note that lists of values, anonymous functions, and datatype constructors with value arguments are all considered values.

(a) [5 pts]

```
case #zardoz (#ford [    ] ) of
  SOME (m) => m * (m-1)
| NONE => 41
```

Answer: `{ford = {zardoz = SOME(7)}}`

(b) [6 pts]

```
let
  val zardoz = fn x => fn y => y x
in
  zardoz 41 [ ] 43
end
```

Answer: One possible answer: `fn x => fn y => 42`

(c) [6 pts]

```
foldl (fn (x,y) => x*x + y) 1 [ ]
```

Answer: Possible answers: `[4,5]` or `[3,4,4]` or `[6,1,2]`

(d) [6 pts]

```
let
  datatype ford = Ford | Arthur of ford * ford
  fun zardoz(t:ford) : int =
    case t of
      Ford => 2
    | Arthur(l,r) => zardoz(l) * (zardoz(r) + 1)
in
  zardoz [ ]
end
```

Answer: Possible answer: `Arthur(Arthur(Ford,Ford), Arthur(Ford,Ford))`

4. Abstraction [30 pts] (parts a–???)

In this problem, you will work with a new queue ADT – a *double-ended queue*, or *deque* for short. A deque is a generalization of a standard queue. In a normal queue, you enqueue elements at one end and dequeue them at the other; in a deque, either operation can occur at either end. That is, given a deque containing the elements:

```
1, 2, 17, 8
```

you can enqueue the element 5 either from the left, to produce:

```
5, 1, 2, 17, 8
```

or from the right, to produce:

```
1, 2, 17, 8, 5
```

Starting with the original queue, you can also dequeue from the left, to produce:

```
2, 17, 8
```

or from the right, to produce:

```
1, 2, 17
```

All other aspects of deques are the same as for queues.

Below is a possible signature for a deque.

```
signature DEQUE = sig
   exception Empty
   exception NotFound

   datatype side = LHS | RHS
   type 'a deque

   (* empty is the empty deque. *)
   val empty : 'a deque
   (* isEmpty q is true iff q has no elements. *)
   val isEmpty : 'a deque -> bool
   (* size q is the number of elements of q. *)
   val size : 'a deque -> int

   (* enqueue s e q = deque q with e enqueued at side s *)
   val enqueue : side -> 'a -> 'a deque -> 'a deque
   (* dequeue s q = (e,q') where e is the element at side s of q,
    * and q' is the deque after removing e. *)
   val dequeue : side -> 'a deque -> 'a * 'a deque
   (* lookup p q = is the leftmost element in the queue for
    * which predicate p is true. *)
   val lookup : ('a -> bool) -> 'a deque -> 'a
end
```

(a) [4 pts] Two pre-conditions are missing in the above specifications. Indicate the functions that need to be fixed and write the missing specifications.

Answer:

`dequeue`*: Raises: Empty if deque q is empty.*

`lookup`*: Raises: NotFound if no element in q satisfies predicate p.*

(b) [3 pts] Users of this abstraction may consider that the specification for `lookup` is too strong. Propose a weaker, non-deterministic specification for this function.

Answer: `lookup p q` *is an element of q for which p holds.*

A possible implementation is using a pair of two lists that model the two ends of the deque. For instance, the following deque:

```
5, 1, 2, 17, 8
```

can be represented as the pair `([5,1], [8,17,2])`.) Using this representation, a partial implementation of a deque is as follows:

```
structure Deque :> DEQUE = struct
  exception Empty
  exception NotFound

  datatype side = LHS | RHS
  type 'a deque = ('a list) * ('a list)

  val empty = ([],[])

  fun isEmpty((l1,l2): 'a deque) : bool =
    (null l1) andalso (null l2)

  fun size((l1,l2):'a deque):int =
    (length l1) + (length l2)

  fun enqueue (s:side) (e:'a) ((l1,l2):'a deque):'a deque =
    case s of
      LHS => (e::l1,l2)
    | RHS => (l1,e::l2)
end
```

(c) [6 pts] Specify the abstraction function for this implementation.

Answer: *Abstraction Function: a pair* $([l_1, .., l_n], [r_1, .., r_m])$ *describes the deque* $l_1, .., l_n, r_m, .., r_1$.

(d) [6 pts] Below is a partial implementation for function `dequeue`. Fill in the rest of this function. Make sure your code works correctly for all cases.

Answer:

```
fun dequeue (s:side) ((l1,l2):'a deque):'a * 'a deque =
  case (s,l1,l2) of
    (LHS,h::t,l2) => (h, (t,l2))
  | (RHS,l1,h::t) => (h, (l1,t))
  | (LHS,[],_::_) => dequeue LHS  (rev l2, [])
  | (RHS,_::_,[]) => dequeue RHS  ([], rev l1)
  | (_,_,_) => raise Empty
```

(e) [5 pts] Your `dequeue` function is probably inefficient if one of the lists is empty and the program performs successive dequeue operations, alternating the ends from where elements are being removed. For instance:

```
dequeue RHS (#2 (deqeue LHS (#2 (dequeue RHS ([],[1,2,3])))))
```

Briefly describe an approach to improve the efficiency of the program in such situations. You don't have to write code.

Answer: *If the list on the corresponding side is empty when performing a dequeue, only move (and reverse) half of the other list, not the entire list. The deque will then have elements at both ends, so dequeues from alternating ends will not require any further moves.*

(f) [6 pts] Now you want to test your code. Assume that you perform Black-Box testing, so you cannot access the implementation structures. An example test case for `size` is "`size {4,3,2,1}`". Here, `{4,3,2,1}` is just a notation for a deque with four elements.

Give 8 good test cases for your `enqueue` and `dequeue` functions (4 test cases for each function). For each case, explain briefly how it provides good coverage (three or four words is enough). You will not receive points for redundant test cases.

Answer: *`enqueue LHS 1 {}` adds to empty list from left*

`enqueue RHS 1 {}` adds to empty list from right

`enqueue LHS 1 {2,3}` adds to nonempty list from left

`enqueue RHS 1 {2,3}` adds to nonempty list from right

`dequeue LHS {2,3}` left dequeue from nonempty list

`enqueue RHS {2,3}` right dequeue from nonempty list

`dequeue LHS {}` left dequeue from empty - should raise exception

`enqueue RHS {}` right dequeue from empty - should raise exception

5. Induction [22 pts] (parts a–???)

Consider the following function:

```
fun f(m:int, n:int, p:int) : int =
   if p = 0 then m
   else f(n, 2*n, p-1) + m
```

(a) [8 pts] Show the evaluation of the expression `f(3,1,2)` using the evaluation model discussed in class. At each step, underline the subexpression that is being reduced. You may use ellipses (...) for parts that remain unchanged from one step to another.

Answer:

```
f(3,1,2) → if 2 = 0 then 3 else f(1, 2*1, 2-1) + 3
         → if false then 3 else f(1, 2*1, 2-1) + 3
         → f(1, 2*1, 2-1) + 3
         → f(1, 2, 2-1) + 3
         → f(1, 2, 1) + 3
         → (if 1 = 0 then 1 else f(2, 2*2, 1-1) + 1) + 3
         → (if false then 1 else f(2, 2*2, 1-1) + 1) + 3
         → (f(2, 2*2, 1-1) + 1) + 3
         → (f(2, 4, 1-1) + 1) + 3
         → (f(2, 4, 0) + 1) + 3
         → ((if 0 = 0 then 2 else f(4, 2*4, 0-1) + 2) + 1) + 3
         → ((if true then 2 else f(4, 2*4, 0-1) + 2) + 1) + 3
         → (2 + 1) + 3
         → 3 + 3
         → 6
```

(b) [10 pts] Prove by induction that $\texttt{f}(\texttt{m}, \texttt{n}, \texttt{p}) = \texttt{m} + \texttt{n}(2^{\texttt{p}} - 1)$. Be sure that you clearly specify what you're inducting on, what you're proving, what the induction hypothesis is, and when you use it.

Answer: *The proof is by induction on p.*

The property is: $P(p) = f(m, n, p) = m + n(2^p - 1)$ for all m and n. We want to prove that $P(p)$ holds for all $p \geq 0$.

Base case: P(0). We need to prove that $f(m, n, 0) = m + n(2^0 - 1)$ for all m and n. Indeed, the evaluation of $f(m, n, 0)$ proceeds as follows:

```
f(m,n,0) → if 0 = 0 then m else f(n, 2*n, 0-1) + m
         → if true then m else f(n, 2*n, 0-1) + m
         → m
```

Since $m + n(2^0 - 1) = m + n(1 - 1) = m$, this proves $P(0)$.

Induction Hypothesis. Assume $P(p)$ holds for some p. That is, $f(m, n, p) = m + n(2^p - 1)$ for that p and for all m, n.

Induction Step. We want to prove that $P(p')$ holds, where $p' = p + 1$. Note that since $p \geq 0$, we know that $p' \geq 1$, so $p' \neq 0$. The evaluation of $P(p') = f(m, n, p')$ is as follows:

```
f(m,n,p') → if p' = 0 then m else f(n, 2*n, p'-1) + m
          → if false then m else f(n, 2*n, p'-1) + m
          → f(n, 2*n, p'-1) + m
          → f(n, 2*n, p) + m
          → f(n, n', p) + m
```

where $n' = 2n$. By Induction Hypothesis for n, n', and p, we get that:

$$f(n, n', p) \rightarrow n + n'(2^p - 1)$$

Therefore:

$$\begin{aligned} f(n, n', p) + m &\rightarrow (n + n'(2^p - 1)) + m \\ &= (n + 2n(2^p - 1)) + m \\ &= (n + 2n2^p - 2n) + m \\ &= (n2^{p+1} - n) + m \\ &= m + n(2^{p+1} - 1) \\ &= m + n(2^{p'} - 1) \end{aligned}$$

This proves that $P(p)$ holds for all $p \geq 0$.

Therefore, $f(m, n, p) = m + n(2^p - 1)$ for all $p \geq 0$ and all m and n.

(c) [4 pts] What is the asymptotic complexity of this function with respect to parameter `n`. That is, for fixed values of `m` and `p`, what is the order of growth of `f(m,n,p)` relative to `n`? You don't have to write a proof, but just state the complexity.

Answer: *The complexity of* `f(m,n,p)` *with respect to* `n` *is* $O(1)$ *because the running time of* `f(m,n,p)` *does not depend on* `n`*, but on* `p`*. For fixed* `p`*, the running time of* `f(m,n,p)` *is fixed.*