

---

## Reading

---

1. Chapter 0 and 1 of *Real World Haskell* <http://book.realworldhaskell.org/>.
2. Chapter 5 of the text (Concepts in Programming Languages) except 5.4.5. Haskell versions of the code examples from Chapter 5 are posted on the Handouts section of the CS242 Web site <http://www.stanford.edu/class/cs242/handouts/>.
3. Chapter 8, Sections 8.1–8.3 (only).
4. Chapter 4, Section 4.4 (only).
5. (Optional) *A history of Haskell: being lazy with class* <http://doi.acm.org/10.1145/1238844.1238856>.

---

## Problems

---

1. .... Modifying functional programs

Quicksort is a well-known sorting algorithm. The algorithm works by choosing some element of the list to be sorted, called the *pivot*, and then splitting the list into elements less than the pivot and elements greater than the pivot. Then each sublist is sorted and the results concatenated. While the worst-case behavior of Quicksort is not very good, the average behavior is excellent. Here are implementations of Quicksort in Haskell, a pure functional language, and C, an impure and not really functional language.

The first five lines below are the Haskell program. The first line says that `qsort` of the empty list is the empty list. The second line applies to a list argument matching the pattern `x:xs`, binding `x` to the first element of the list and `xs` to the list of remaining elements. Lines 2–5 of the code “say” that Quicksort of a nonempty list `x:xs` is the list obtained by concatenating the Quicksort of the elements less than `x` with `x` itself and the Quicksort of the elements greater than or equal to `x`. Questions about the programs appear after the code.

```
qsort [] = []
qsort (x:xs) = qsort elts_lt_x ++ [x] ++ qsort elts_greq_x
               where
                 elts_lt_x    = [y | y <- xs, y < x]
                 elts_greq_x = [y | y <- xs, y >= x]
```

```

qsort( a, lo, hi ) int a[], hi, lo;
{
    int h, l, p, t;

    if (lo < hi) {
        l = lo;
        h = hi;
        p = a[hi];

        do {
            while ((l < h) && (a[l] <= p))
                l = l+1;
            while ((h > l) && (a[h] >= p))
                h = h-1;
            if (l < h) {
                t = a[l];
                a[l] = a[h];
                a[h] = t;
            }
        } while (l < h);

        t = a[l];
        a[l] = a[hi];
        a[hi] = t;

        qsort( a, lo, l-1 );
        qsort( a, l+1, hi );
    }
}

```

- (a) Which list element is used as the pivot in each example?
- (b) Which code seems easier to understand? Why? What if you had never programmed in either language before?
- (c) Each Haskell list comprehension, such as  $[y \mid y \leftarrow xs, y < x]$ , is computed by a separate recursive pass through the list  $xs$ . What might be a better approach?
- (d) Explain how to modify the C code to use a randomly chosen list element as the pivot.
- (e) Do you see a straightforward way to modify the Haskell version to use a random pivot? (*Hint: Just give a one-sentence answer.*)
- (f) Suppose you had to explain the modified code to an Army General who is going to use this in a system to control launch of nuclear missiles. The Army General has never taken a programming course, or perhaps knows a little Cobol. In which case would it be easier to convince a skeptic that the code is actually correct?
- (g) How much extra space (memory), in addition to the space used to represent the input list, do you think the Haskell program uses? Since we haven't talked about how Haskell is implemented, just try to make a reasonable, educated estimate. How does this compare to the memory requirements of the C program?

## 2. .... Lazy and Eager Evaluation

Function calls in most languages, including ML, Lisp, C, and Java, are evaluated by first evaluating the arguments to the function and then performing the function call. This is called *eager evaluation*. In the functional language Haskell, function arguments are not evaluated until they are needed—arguments are left unevaluated until the program actually uses their value. This is called *lazy evaluation*. In lazy evaluation, if an argument is never used, then it is never evaluated.

Lazy evaluation can be implemented using *thunks*, which are no-argument functions that are called when an expression needs to be evaluated. For example, a lazy expression  $f(e)$  can be compiled to eager code that passes a thunk for  $e$  to the function body of  $f$ . The thunk for  $e$  will only be evaluated if both  $f$  is evaluated during the course of the program, and the body of  $f$  needs the value of  $e$ . For example, if the body of  $f$  contains  $e + 1$ , then the thunk for  $e$  will be evaluated whenever  $f$  is.

Consider the following code fragment, written in a Javascript-like language.

```
function times3 (x) { return x + x + x; }
times3 (5 * 7);
```

Let us think about how we can implement the eager and lazy evaluation strategies for this code using the (eager) Javascript language.

Because Javascript is an eager language, we don't have to make any changes at all to implement an eager evaluation strategy in Javascript. Simply running the code as written will first compute  $5 * 7 = 35$  and then call `times3` with the argument 35.

We can implement lazy evaluation in Javascript by writing the following code:

```
function times3 (x_thunk) { return x_thunk() + x_thunk() + x_thunk (); }
times3 (function () { return 5 * 7; } );
```

The `times3` function expects a thunk that produces the value of its argument, and calls the function where the value of the argument is needed. Rather than receiving the value 35, instead `times3` now takes a function that evaluates  $5 * 7$  when needed.

- (a) Haskell is a pure functional language. What does this imply about the three values of  $x$  in any call to `times3`?
- (b) Since `times3(x)` contains three occurrences of  $x$ , the thunk for the function argument will be called three times. What trick could you use in a Haskell compiler to improve the efficiency of the compiled code for `times3`?
- (c) Translate the following eager Javascript code into a lazy version using thunks.

```
function g (x, y) {
  if (x < 0) return 0;
  else return y;
}

function g ( _____ , _____ ) {
  if ( _____ < 0 )
  return 0;
  else _____ ;
}
```

- (d) Does the choice of evaluation strategy change the eventual output of a pure functional program? More specifically, if  $f$  is a function and  $e_1 e_2 \dots e_n$  are arguments, is the value of  $f(e_1, e_2, \dots, e_n)$  the same under eager and lazy evaluation strategies, assuming that all are written in a pure functional language? If so, explain briefly, if not, provide a counterexample and brief explanation. (*Hint*: Consider the example functions used in this question, and all possible arguments they might be applied to.)

### 3. .... Lazy Evaluation and Parallelism

In a “lazy” language, a function call  $f(e)$  is evaluated by passing the *unevaluated* argument to the function body. If the value of the argument is needed, then it is evaluated as part of the evaluation of the body of  $f$ . For example, consider the function  $g$  defined by

```
fun g(x, y) = if x = 0
              then 1
              else if x + y = 0
                    then 2
                    else 3;
```

In a lazy language, the call  $g(3, 4 + 2)$  is evaluated by passing some representation of the expressions  $3$  and  $4 + 2$  to  $g$ . The test  $x = 0$  is evaluated using the argument  $3$ . If it were `true`, the function would return  $1$  without ever computing  $4 + 2$ . Since the test is `false`, the function must evaluate  $x + y$ , which now causes the actual parameter  $4 + 2$  to be evaluated. Some examples of lazy functional languages are Miranda, Haskell and Lazy ML; these languages do not have assignment or other imperative features with side effects.

If we are working in a pure functional language without side-effects, then for any function call  $f(e_1, e_2)$ , we can evaluate  $e_1$  before  $e_2$  or  $e_2$  before  $e_1$ . Since neither can have side-effects, neither can affect the value of the other. However, if the language is lazy, we might not need to evaluate both of these expressions. Therefore, something can go wrong if we evaluate both expressions and one of them does not terminate.

As Backus argues in his Turing Award lecture, an advantage of pure functional languages is the possibility of parallel evaluation. For example, in evaluating a function call  $f(e_1, e_2)$  we can evaluate both  $e_1$  and  $e_2$  in parallel. In fact, we could even start evaluating the body of  $f$  in parallel as well.

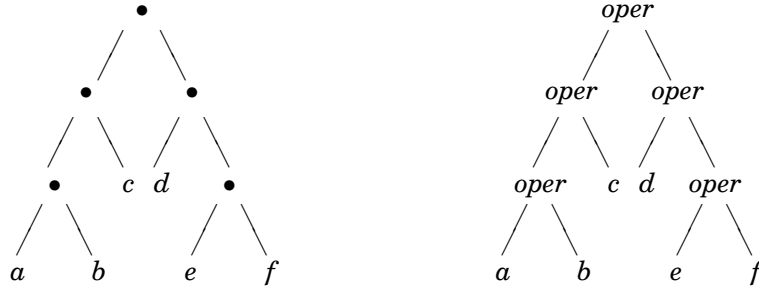
- (a) Assume we evaluate  $g(e_1, e_2)$  by starting to evaluate  $g$ ,  $e_1$ , and  $e_2$  in parallel, where  $g$  is the function defined above. Is it possible that one process will have to wait for another to complete? How can this happen?
- (b) Now, suppose the value of  $e_1$  is zero and evaluation of  $e_2$  terminates with an error. In the normal (i.e., eager) evaluation order that is used in C and other common languages, evaluation of the expression  $g(e_1, e_2)$  will terminate in error. What will happen with lazy evaluation? Parallel evaluation?
- (c) Suppose you want the same value, for every expression, as lazy evaluation, but you want to evaluate expressions in parallel to take advantage of your new pocket-sized multiprocessor. What actions should happen, if you evaluate  $g(e_1, e_2)$  by starting  $g$ ,  $e_1$ , and  $e_2$  in parallel, if the value of  $e_1$  is zero and evaluation of  $e_2$  terminates in an error?
- (d) Suppose, now, that the language contains side-effects. What if  $e_1$  is  $z$ , and  $e_2$  contains an assignment to  $z$ . Can you still evaluate the arguments of  $g(e_1, e_2)$  in parallel? How? Or why not?

### 4. .... Haskell Reduce for Trees

The binary tree datatype

```
data Tree a = Leaf a |
             Node (Tree a) (Tree a)
```

describes a binary tree for any type, but does not include the empty tree (i.e., each tree of this type must have at least a root node).



- Write a function

$$\text{reduce} :: (a \rightarrow a \rightarrow a) \rightarrow \text{Tree } a \rightarrow a$$

that combines all the values of the leaves using the binary operation passed as a parameter. In more detail, if  $\text{oper} : a \rightarrow a \rightarrow a$  and  $t$  is the nonempty tree on the left in this picture, then  $\text{reduce } \text{oper } t$  should be the result obtained by evaluating the tree on the right. For example, if  $f$  is the function

$$\begin{aligned} f &:: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\ f \ x \ y &= x + y \end{aligned}$$

then  $\text{reduce } f \ (\text{Node } (\text{Node } (\text{Leaf } 1) (\text{Leaf } 2)) (\text{Leaf } 3)) = (1 + 2) + 3 = 6$ . Explain your definition of  $\text{reduce}$  in one or two sentences.

- Write a function  $\text{toList} :: \text{Tree } a \rightarrow [a]$  that returns a list of the elements in the argument tree.
- Write a function  $\text{reduceList} :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a$  that reduces a list according to the supplied function argument.
- Use  $\text{toList}$  and  $\text{reduceList}$  to write a predicate  $\text{prop\_reduceTest}$ , suitable for testing with QuickCheck.

A possible layout for the entire code is given below :

```
{-# OPTIONS -XTypeSynonymInstances #-}

import Test.QuickCheck

data Tree a = Leaf a | Node (Tree a) (Tree a)

reduce :: (a->a->a) -> Tree a -> a
reduce f t = < ... >

toList :: Tree a -> [a]
toList t = < ... >

reduceList :: (a->a->a) -> [a] -> a
reduceList f l = < ... >

prop_reduceTest :: (Int->Int->Int) -> TS -> Bool
prop_reduceTest f tree = < ... >

type TS = Tree Int
instance Arbitrary TS where
  arbitrary = do
    n <- choose (1,2) :: Gen Int
    case n of
```

```

1 -> do i <- arbitrary
      return (Leaf i)
2 -> do t1 <- arbitrary
      t2 <- arbitrary
      return (Node t1 t2)

```

```

quickCheck (prop_reduceTest (+))
quickCheck (prop_reduceTest (*))

```

You need to fill in the blank spaces (< ... >) that appear in the above code. The above code may not compile on some of the older versions of GHC. For older versions, you could try starting `ghci` with the `-fglasgow-exts` flag ie running `ghci -fglasgow-exts`.

## 5. .... Exceptions

Consider the following functions, written in JavaScript:

```

function Excpt(value){ //Excpt is used as an exception-object
  this.val=value;
  this.name="Exception";
}
function twice(f,x){
  try{
    return f(f(x));
  }
  catch(e){
    return e.val;
  }
}
function pred(x){
  if(x==0)
    throw new Excpt(x);
  else
    return x-1;
}
function dumb(x){
  throw new Excpt(x);
}
function smart(x){
  try{
    return 1+pred(x);
  }
  catch(e){
    return 1;
  }
}

```

What is the result of evaluating each of the following expressions?

- (a) `twice(pred,1);`
- (b) `twice(dumb,1);`
- (c) `twice(smart,0);`

In each case, be sure to describe which exception gets raised and where.

## 6. .... Exceptions and Recursion

Here is a JavaScript function that uses an exception called `OddExcept`.

```
function OddExcept() {
    this.desc="Exception";
}
function f(n) {
    if (n==0)
        return 1;
    if (n==1)
        throw new OddExcept;
    if (n==3)
        return f(3-2);
    try{return f(n-2);}
    catch(e){return -n;}
}
```

When `f(11)` is executed, the following steps will be performed:

```
call f(11)
call f(9)
call f(7)
...
```

Write down the remaining steps that will be executed. Include only the following kinds of steps:

- function call (with argument)
- function return (with return value)
- raise an exception
- pop activation record of function off stack without returning control to the function
- handle an exception

Assume that if `f` calls `g` and `g` raises an exception that `f` does not handle, then the activation record of `f` is popped off the stack without returning control to the function `f`.

## 7. .... Tail Recursion and Continuations

(a) Explain why a tail recursive function, such as

```
function fact(n){
    function f(n,a){if (n==0) return a
                    else return f(n-1, a*n)
    }
    return f(n,1)
};
```

can be compiled so that the amount of space required to compute `fact(n)` is independent of `n`.

(b) The function `f` used in the following definition of factorial is “formally” tail recursive: the only recursive call to `f` is a call that need not return.

```
function fact(n){
  function f(n,g){
    if (n==0) return g(1);
    else return f(n-1, function(x){return g(x)*n})
  };
  return f(n, function(x){return x})
};
```

**How much space is required to compute `fact(n)`, measured as a function of argument `n`? Explain how this space is allocated during recursive calls to `f` and when the space may be freed.**