Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.827 Multithreaded Parallelism: Languages and Compilers

Problem Set 1

Please Remember: 1) You can work in groups of up to three people; include the names of all group members on all problems. 2) Turn in the answer to each problem as a separate packet. 3) Comment your code carefully and include output from sample runs.

Problem 1 Numerical Integration

This problem is intended to make you comfortable programming in functional languages, namely Haskell. The choice of the programming environment is entirely up to you – you can take a look at

`http://www.haskell.org/implementations.html`

and choose whatever you like. We recommend that you download and install The Haskell Interpreter Hugs available from

`http://www.haskell.org/hugs/`

In order to make sure things work correctly, type the following excerpt as seen in Lecture 2 and save it as `test.hs` in your home directory:

```
apply_n f n x = if n==0 then x
                else apply_n f (n-1) (f x)

plus a b = apply_n ((+) 1) b a
mult a b = apply_n ((+) a) b 0
expon a b = apply_n ((*) a) b 1
```

then run hugs and load the file by typing

```
:load test.hs
```

You can now call one of the functions you have just specified by typing

```
plus 3 4
```

(Obviously, you should obtain 7).

Now, back to the problem set. In this first problem, we examine two simple algorithms for numerical integration based on Simpson's rule.

Part a:

Given a function f and an interval $[a, b]$, Simpson's rule says that the integral can be approximated as follows:

$$I = \frac{h}{3}\left[f(a) + 4f(a+h) + f(a+2h)\right]$$

where

$$h = \frac{b-a}{2}$$

For better accuracy, the interval of integration $[a, b]$ is divided into several smaller subintervals. Simpson's rule is applied to each of the subintervals and the results are added to give the total integral. If the subintervals of $[a, b]$ are all of the same size p, where $p = 2h$, then we have the *Composite Strategy* for integration.

Write a Haskell program containing a function

```
composite_strategy f a b n
```

where

f is the function to integrate

a,b are the endpoints of the integration interval

n is the number of subintervals such that $h = (b-a)/2n$

Integrate some simple functions and try different values for n.

Part b:

If the subintervals of the integration are not all equal and can be changed as required, then we obtain an *Adaptive Strategy* for integration. A simple adaptive integration algorithm can be described as follows:

1. Approximate the integral using Simpson's rule over the entire interval $[a, b]$. Call this approximation *old_approx*.

2. Compute the midpoint x of the interval: $x = (b + a)/2$.
3. Approximate a new interval by applying Simpson's rule to the subintervals $[a, x]$ and $[x, b]$ and add the two results. Call this approximation *new_approx*.
4. If the absolute value of the difference between *new_approx* and *old_approx* is within some limit *sigma* then return *new_approx*.
5. Otherwise, apply the adaptive strategy recursively to each of the subintervals, add the two results, and return the sum as the answer.

Write a Haskell program that contains the function

```
adaptive_strategy f a b sigma
```

Make sure `adaptive_strategy` takes advantage of the parallelism available in the algorithm. Try integrating several functions while varying the *sigma* parameter.

Part c:

How do the algorithms vary in complexity? Compare the accuracy of the answers. What factors affect the execution time and efficiency of the two strategies?

Problem 2 Using λ combinators

The next two problems on this problem set focus on the pure λ-calculus. We recommend that you take a look at the pH Book, Appendix A, before you move on with this problem set. The idea is to become comfortable with the reduction rules used, and with the important differences between some of the reduction strategies which can be used when applying those rules.

In this problem, we shall write a few combinators in the pure λ-calculus to get familiar with the rules of λ-calculus. Here are the definitions of some useful combinators.

$$\begin{array}{lcl}
\text{TRUE} & = & \lambda x.\lambda y.x \\
\text{FALSE} & = & \lambda x.\lambda y.y \\
\text{COND} & = & \lambda x.\lambda y.\lambda z.x\ y\ z \\
\text{FST} & = & \lambda f.f\ \text{TRUE} \\
\text{SND} & = & \lambda f.f\ \text{FALSE} \\
\text{PAIR} & = & \lambda x.\lambda y.\lambda f.f\ x\ y \\
\underline{n} & = & \lambda f.\lambda x.(f^n\ x) \\
\text{SUC} & = & \lambda n.\lambda a.\lambda b.a\ (n\ a\ b) \\
\text{PLUS} & = & \lambda m.\lambda n.m\ \text{SUC}\ n \\
\text{MUL} & = & \lambda m.\lambda n.m\ (\text{PLUS}\ n)\ \underline{0}
\end{array}$$

Now, write the λ-terms corresponding to the following functions.

- The boolean AND function.

- The boolean OR function.
- The boolean NOT function.
- The exponentiation function (EXP). You should write two expressions, one using MUL and the other without MUL (note: don't eliminate MUL by substituting the body of the MUL combinator into your first definition—MUL only "stands for" its definition in the first place, so you've done nothing).
- The function ONE? which tests whether the given number is 1. (Hint: use the data structure combinators. Don't try to construct a lambda term from whole cloth.)
- The function PRED which subtracts 1 from the given number. You may decide what to do when 0 is passed as an argument to PRED. (Extra credit: can you come up with a term T for which (SUC T) reduces to $\underline{0}$? If so, give the term; if not, explain why.)

In addition to the given combinators, you are free to define any others which you think would be useful.

Problem 3 Evaluation strategies for the λ calculus

In lecture, Prof. Arvind discussed interpreters for the λ calculus, and gave two examples: call-by-name, written *cn(E)*, and call-by-value, written *cv(E)*. We consider both of these interpreters to be finished when they return an answer in *Weak Head Normal Form.* In this problem, we're going to look at similar interpreters which yield answers in *β normal form*—that is, an expression which cannot possibly be β-reduced anymore.

Part a:

First, consider the *applicative order* λ-calculus. Here, we pursue a leftmost innermost strategy (choose the leftmost redex, or the innermost such redex if the leftmost redex *contains* a redex). Reduce the following term, step by step, using this strategy. You will probably want to parenthesize the term fully before you decide which redex is innermost (remember if two redexes are equally far "in" when fully parenthesized, you must choose the leftmost one).

$$(\lambda x.\lambda y.x)\ (\lambda z.(\lambda x.\lambda y.x)\ z\ ((\lambda x.z\ x)(\lambda x.z\ x)))$$

Part b:

Write an interpreter, *li(E)*, for the applicative order λ-calculus. Your answer should be similar to the call-by-value interpreter from class.

Part c:

Now consider the *normal order* λ-calculus. It should yield answers in normal form by using a leftmost outermost strategy (choose the leftmost redex, making sure it's the outermost if several redexes are nested). Using this strategy, reduce the following term:

$$(\lambda x.\lambda y.x)\ (\lambda z.(\lambda x.\lambda y.x)\ z\ ((\lambda x.x\ x)(\lambda x.x\ x)))$$

Part d:

Write a normal order interpreter *lo(E)*. Your answer should work in a manner similar to the call-by-name interpreter from class, though it will require a more elaborate set of rules. Hint: you'll probably want to use two slightly different sets of mutually recursive reduction rules. If you're having trouble, figure out what your interpreter does with the term $(\lambda y.\lambda z.y\,((\lambda x.x\,x)(\lambda x.x\,x)))\,(\lambda x.\lambda y.y)$.

Part e:

What happens when you run *li* on the term from part c? Is the applicative order λ-calculus strongly normalizing?

Problem 4 — Recursion and confluence in λ and λ_{let}

Consider the following very simple term in the λ calculus with let bindings:

```
let g =  λx. g (g x);
in  g (λx. x)
```

This term uses binding to set up a simple recursive function definition. In this problem we will examine how such a term might be represented in λ calculus. In doing so, we'd like to develop an intuition about why non-confluence might break down in let-bound calculi.

Part a:

Using instantiation, give two reductions of the above term which are not confluent—that is, they can never be brought back together. Argue why they must always remain distinct.

Part b:

We can translate the above term into λ calculus, yielding the following term; note how the subterms of the original term are kept intact:

```
(λg.  g (λx. x)) ((λg. (λx. g (x x)) (λx. g (x x))) (λg.  λx. g (g x)))
```

Give a translation for the two terms in the previous exercise, again preserving the right-hand sides of let bindings and the "in" part of the block.

Part c:

Can the above translation of the initial term be reduced to your translation of either of the other two terms in the λ calculus without let binding? If so, give the reduction. If it cannot, argue why such a translation should not be possible or give an alternative translation for which it will be possible.

Part d:

Can the translation of your two terms be brought together in the λ calculus? If so, show the reductions for each term. If not, explain why it would be impossible or give an alternative translation for which it is possible.