CS 311 Homework 9

due 16:40, Tuesday, 30^{th} November 2010

Homework must be submitted on paper, in class. There are three optional "extra credit" questions. These are intended to enable students who are concerned about their grade to earn some extra points.

Several questions ask you to turn in some output from the *LambdaCalculator*. This is in addition to, not instead of, your written narrative! Use cut and paste (on the computer, or with scissors and glue) to create a coherent document that explains your work.

Instructions for downloading and running the *LambdaCalculator* are on the class web page. Don't forget to type "main" after loading the *LambdaCaculator*. Be warned: use the version from Prof Sheard's web page, not from *Hackage*, since the definitions for the Church numerals in the *Hackage* version are different.

Question 1. [20 pts.]

Give an informal description of a Turing machine that decides the language

$\{w \mid w \in \{0,1\}^*$ and w contains more 1s than 0s$\}$

What is an "informal description"? Here is an example:

1. Move right until you find a blank, and replace it by X.
2. Move left until you find a Y or a blank. If you find a blank, accept.
3. Move left replacing all 1s by As until you read a blank
4. Go to step 1.

Question 2. [20 pts.]

The **Simple** language was defined in the lecture on "Other Models of Computation", slides 10–12. Using the **Simple** language primitives and any of the macros on Slide 12, define a macro that computes X := 0 when A > B, and X := succ(0) otherwise.

Question 3. [20 pts.]

Show that a RESET Turing machine is equivalent in power to an ordinary Turing machine. (This means that you must show bi-simulation, that is, that either machine can simulate the other.)

A RESET Turing machine has a tape that is bounded to the left but infinite to the right. At each move, after reading a symbol from and writing a symbol to the tape, the head can move RIGHT, or it can RESET to the start of the tape. It cannot move one space to the left.

Question 4. [20 pts.; 10 pts each]

In the λ-calculus, define:

a. The function monus on Church numerals. The Church numeral for n is already defined in the *LambdaCalculator* as $\#n$.

$$\mathsf{monus}\ x\ y = \begin{cases} x - y & \text{if } x > y \\ 0 & \text{otherwise} \end{cases}$$

b. The boolean implication function implies.

In class, we defined

$$\begin{aligned} \mathsf{true} &= \lambda t.\lambda f.t \\ \mathsf{false} &= \lambda t.\lambda f.f \end{aligned}$$

These definitions are known to the *LambdaCalculator*.

Check your answers by running them in the *LambdaCalculator*. Test all four cases:

implies true true ⇒ true
implies true false ⇒ false
implies false true ⇒ true
implies false false ⇒ true

Turn in your output showing the definitions and the tests of both functions.

Question 5. [10 pts.]

The function ifZero is defined in the LambdaCalculator as $\lambda x.xtrue(\lambda y.false)$. Try out ifZero on the Church numerals zero, one, and two. Explain in your own words why ifZero works.

Question 6. [Optional; for extra Credit (20 pts.)]

Using the above representations for the Booleans, define

a. xor,

b. nand, and

c. not.

Test them exhaustively as you did for implies in Question 4.*b*.

Hint: It may help to start with definitions that use if-then-else. However, if is actually superfluous: look at the way that it is defined in the *LambdaCalculator*! The Church Booleans can be viewed as values, but also as *control structures* that make a choice.

Turn in your output.

Question 7. [Optional; for extra Credit (20 pts.)]

The usual recursive definition of factorial is

$$\mathsf{factorial} = \lambda n.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times \mathsf{factorial}(n-1) \tag{1}$$

You can of course re-write the right hand side using ifZero, mult and pred, but it still won't be a legal definition in the *LambdaCalculator*. Why?

However, suppose that we apply β-conversion to the rhs of (1). Then we get:

$$\mathsf{factorial} = (\lambda f.\ \lambda n.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times f(n-1))\ \mathsf{factorial} \tag{2}$$

Equation (2) has the form factorial $= h$ factorial. This says that factorial is a fixed point of the function h. (Refer to the slides from lecture 14 if you don't understand this remark.)

Define h in the *LambdaCalculator*.

Now $(y\ h)$ should be the[1] fixed point of h, i.e., factorial. Try applying $(y\ h)$ to some (small!) Church-numerals in the *LambdaCalculator* to test this definition.

Turn in:

1. your explanation as to *why* Equation (1) can't be a legal definition in the *LambdaCalculator*,
2. your definition of h, and
3. the output of (y h) #3. (You may use :b to do a bunch of reductions in one interaction.)

Question 8. [Optional; for extra Credit (30 pts.)]

In the lecture (slide 29) I defined the factorial function by first defining two auxiliary λ-expressions, base and step, and then using the fact that the Church numeral $\#n$ applies its second argument n times to its first. This gave us, in effect, an *iterative* definition of factorial.

You can play a similar game with the Fibonacci function, fib, defined by

$$\begin{aligned} \mathsf{fib}\ 0 &= 0 \\ \mathsf{fib}\ 1 &= 1 \\ \mathsf{fib}\ n &= (\mathsf{fib}\ (n-1)) + (\mathsf{fib}\ (n-2)) \end{aligned}$$

Your task is to define fib in the *lambdaCalculator* using an iterative form.

Hint. As with factorial, you will need functions that operate on *pairs* of Church numerals, because you will need to carry more than one number through the iteration. I defined a

[1] Strictly, we should be worried about *which* fixed point, if there is more than one. We will defer that worry to a more advanced course.

function nextFibPair that takes as argument an adjacent pair of Fibonacci numbers, like $\langle 3, 5\rangle$, and produces the next (overlapping) pair, $\langle 5, 8\rangle$. I also defined a (constant) function firstFibPair that produces $\langle 0, 1\rangle$.

Turn in some output from the *lambdaCalculator* that shows your function working on small numbers.