Comp 311 - Review 2

Instructor:
Robert "Corky" Cartwright
cork@cs.rice.edu
Review by Mathias Ricken
mgricken@cs.rice.edu

This review sheet provides a few examples and exercises. You do not need to hand them in and will not get points for your work. You are encouraged to do the exercises and work through the examples to make sure you understand the material. This material may be tested on the first exam.

1 Church Numerals

Assume we have a programming language that doesn't support numbers or booleans: a lambda is the only value it provides. It is an interesting question whether we can nonetheless create some system that allows us to count, add, multiply, and do all the other things we do with numbers.

Church numerals use lambdas to create a representation of numbers. The idea is closely related to the functional representation of natural numbers, i.e. to have a natural number representing "zero" and a function "succ" that returns the successor of the natural number it was given. The choice of "zero" and "succ" is arbitrary, all that matters is that there *is* a zero and that there *is* a function that can provide the successor.

Church numerals are an extension of this. All Church numerals are functions with two parameters:

$$\lambda f\ .\ \lambda x\ .\ \textit{something}$$

The first parameter, f, is the successor function that should be used. The second parameter, x, is the value that represents zero. Therefore, the Church numeral for zero is:

$$C_0 = \lambda f\ .\ \lambda x\ .\ x$$

Whenever it is applied, it returns the value representing zero. The Church numeral for one applies the successor function to the value representing zero exactly once:

$$C_1 = \lambda f\ .\ \lambda x\ .\ fx$$

The Church numerals that follow just have additional applications of the successor function:

$$\begin{array}{lcl} C_2 & = & \lambda f\ .\ \lambda x\ .\ f(fx) \\ C_3 & = & \lambda f\ .\ \lambda x\ .\ f(f(fx)) \\ C_4 & = & \lambda f\ .\ \lambda x\ .\ f(f(f(fx))) \\ C_n & = & \lambda f\ .\ \lambda x\ .\ f^n x \end{array}$$

It is important to note that in this minimal lambda calculus, we can't really do very much with these Church numerals. We can count and add and multiply (more about that below), but to understand the result, we have to count the applications of the successor function.

If we had a language that were a little more powerful, however, we could do the following:

$$\begin{array}{lcl} S & = & \lambda r\ .\text{"ring the small bell (ding) and apply } r\text{"} \\ Z & = & \lambda r\ .\text{"ring the big bell (dong)"} \end{array}$$

The application of C_4 to S and Z would then produce "ding ding ding ding dong", since C_4 is $f(f(f(fx)))$.

If we add numbers to our language, then we can convert Church numerals to decimal numbers if we define S and Z as follows:

$$\begin{array}{lcl} S & = & \lambda r \,.\, 1 + r() \\ Z & = & \lambda r \,.\, 0 \end{array}$$

Now C_4SZ is equivalent to $1+1+1+1+0$, so it actually evaluates to 4. Without using "side effects" like ringing bells or being able to turn Church numerals into decimal numbers, we have to count the applications – there's no way around that – and this violates the encapsulation of functions. We do not treat lambdas as black boxes anymore, we look at their bodies and count...

1.1 Addition

We can easily perform addition using Church numerals if we realize that they do everything relative to the value they consider zero. C_1 is one more than C_0, and C_4 is one more than C_3; therefore, C_1 represents 1 relative to C_0, and C_4 represents 1 relative to C_3.

If we want to add 3 to 4 using Church numerals, we simply create a new Church numeral and use one of the summands as zero for the other:

$$C_{3+4} \quad = \quad \lambda f \,.\, \lambda x \,.\, C_3 \; f \; (C_4 \; f \; x)$$

C_{3+4} is a function with two parameters – just like any Church numeral – but it applies C_3 to f, the successor function, and C_4fx, which now acts as value for zero. Written out, C_{3+4} is (parameters have been renamed to avoid shadowing)

$$\begin{array}{lcl} C_{3+4} & = & \lambda f \,.\, \lambda x \,.\, (\lambda f_3 \,.\, \lambda x_3 \,.\, f_3(f_3(f_3 x_3))) \; f \; (\lambda f_4 \,.\, \lambda x_4 \,.\, f_4(f_4(f_4(f_4 x_4))) \; f \; x) \\ & = & \lambda f \,.\, \lambda x \,.\, f(f(f(\lambda f_4 \,.\, \lambda x_4 \,.\, f_4(f_4(f_4(f_4 x_4))) \; f \; x))) \\ & = & \lambda f \,.\, \lambda x \,.\, f(f(f(f(f(f(fx)))))) \\ & = & C_7 \end{array}$$

We can therefore define a function *add* that takes two Church numerals M and N and returns the sum of them:

$$\begin{array}{lcl} add & = & \lambda M \,.\, \lambda N \,.\, \lambda f \,.\, \lambda x \,.\, N \; f \; (M \; f \; x) \\ add \; C_4 \; C_7 & = & C_7 \end{array}$$

add actually has four lambdas, not just two for M and N, since the result is a Church numeral, which is a function with two parameters.

1.2 Multiplication

We can perform a similar trick for multiplication. C_2 is two steps away from C_0, but so is C_4 if your step size is twice as large. For addition, we changed the value that was considered 0. We can achieve multiplication by changing the successor function.

If we want to multiply 2 and 3 using Church numerals, we want to take two steps of 3 starting from zero.

$$C_{2*3} \quad = \quad \lambda f \,.\, \lambda x \,.\, C_2 \; (C_3 \; f) \; x$$

C_4 is a function with two parameters, but since they are individual lambdas, it can also be interpreted as a function with one parameter, which returns a function with another parameter. The lambda calculus has this feature built in since it only knows functions with one parameter; in other languages, you may have to do this manually using a technique called "currying" (named after the logician Haskell Curry). Our language JAM is such a language. The function *add*

```
let add := map x, y to x + y; in ...
```

can be curried into

```
let addc := map x to map y to x + y; in ...
```

This has the advantage that we can easily define a function *add*2 that always adds 2 to the number passed to it:

```
let add2 := map x to addc(x) in ...
```

We make use of this and apply f, the successor function, to C_3. The result is a function with one parameter that applies f three times to whatever was given to it:

$$\begin{array}{ll} & C_3\ f \\ = & (\lambda f_3\ .\ \lambda x\ .\ (f_3(f_3(f_3 x))))f \\ = & \lambda x\ .\ f(f(fx)) \end{array}$$

If we use this as successor function for C_2, then it will make two steps of three, as desired:

$$\begin{array}{lll} C_{2*3} & = & \lambda f\ .\ \lambda x\ .\ C_2\ (\lambda x_3\ .\ f(f(fx_3)))\ x \\ & = & \lambda f\ .\ \lambda x\ .\ (\lambda f_2\ .\ \lambda x_2\ .\ f_2(f_2 x_2))\ (\lambda x_3\ .\ f(f(fx_3)))\ x \\ & = & \lambda f\ .\ \lambda x\ .\ (\lambda x_3\ .\ f(f(fx_3)))((\lambda x_{3a}\ .\ f(f(fx_{3a})))x) \\ & = & \lambda f\ .\ \lambda x\ .\ f(f(f((\lambda x_{3a}\ .\ f(f(fx_{3a})))x))) \\ & = & \lambda f\ .\ \lambda x\ .\ f(f(f(f(fx)))) \\ & = & C_6 \end{array}$$

We can therefore define a function *multiply* that takes two Church numerals M and N and returns the product of them:

$$\begin{array}{lll} multiply & = & \lambda M\ .\ \lambda N\ .\ \lambda f\ .\ \lambda x\ .\ N\ (Mf)\ x \\ multiply\ C_2\ C_3 & = & C_6 \end{array}$$

How would you define *exponentiate*, a function that takes two Church numerals M and N and returns the Church numeral representing M^N?

2 Church Booleans

We can ask the same question we asked about numbers about booleans: Can we represent them using just functions? Yes, we can, and in a way very similar to Church numerals.

A Church boolean is a function with two parameters, the first represents what the function should return if it is true, the second what the function should return if it is false:

$$\begin{array}{lll} tru & = & \lambda x\ .\ \lambda y\ .\ x \\ fls & = & \lambda x\ .\ \lambda y\ .\ y \end{array}$$

Again, with just the minimal lambda calculus, we can't really do a lot, and we always have to look *inside* the Church boolean to figure out what it is. If we had a more powerful language, we could do the following:

$$\begin{array}{rcl} T & = & \text{“ring the small bell (ding)”} \\ F & = & \text{“ring the big bell (dong)”} \end{array}$$

Now $tru\ T\ F$ produces "ding" and $fls\ T\ F$ produces "dong", so we could easily distinguish them. It is interesting to note that Church booleans have an `if-then-else` construct almost built-in: $b\ T\ F$, where b is a Church boolean, is almost the same as `if (b) then T else F`. We can generalize this by defining a function *test* with three parameters b, c, and a, that applies the consequence c and alternative a to the Church boolean b:

$$test \quad = \quad \lambda b\ .\ \lambda c\ .\ \lambda a\ .\ b\ c\ a$$

2.1 Boolean Arithmetic

Just like with Church numerals, we can also perform arithmetic with Church booleans. It is easy to define functions for *and*, *or*, and *not*:

$$\begin{array}{rcl} and & = & \lambda M\ .\ \lambda N\ .\ M\ (N tru\ fls)\ fls \\ or & = & \lambda M\ .\ \lambda N\ .\ M\ tru\ (N\ tru\ fls) \\ not & = & \lambda M\ .\ M\ fls\ tru \end{array}$$

3 SKI and SK Combinator Calculi

Even though the untyped lambda calculus already is very minimalistic, we can further reduce it to just three (S, K, I) or even two (S, K) symbols, or combinators. All terms that can be expressed in the lambda calculus can also be expressed using just the combinators S, K, I and parentheses, and the I isn't even necessary.

Since these new languages contain only applications and the combinators S, K and maybe I, certain properties are easier to prove in these calculi than in the lambda calculus, which is comprised of applications, lambdas, and variables. The SKI and SK combinator calculi are "identifier-free"; the lambda calculus is not.

3.1 SKI Combinator Calculus

Let S, K, I be the following functions:

$$\begin{array}{rcl} I\ x & = & x \\ K\ x\ y & = & x \\ S\ x\ y\ z & = & x\ z\ (y\ z) \end{array}$$

To convert an expression e in the lambda calculus to an expression in the SKI combinator calculus, we can define a function $\varphi(e)$:

If e is of the form ...

1. $\lambda x\ .\ x$, then $\varphi(e) = I$.
2. $\lambda x\ .\ c$, then $\varphi(e) = (Kc)$.

3. $\lambda x\ .\ (\alpha\ \beta)$, then $\varphi(e) = (S(\lambda x\ .\ \varphi(\alpha))(\lambda x\ .\ \varphi(\beta)))$.

Now we iteratively apply this function to the innermost lambda of an expression e until all lambdas have disappeared. In the following example, the innermost lambda(s) has been underlined:

$$
\begin{array}{lcll}
C_2 & = & \lambda f\ .\ \underline{\lambda x\ .\ f(fx)} & \text{(case 3)} \\
 & \rightarrow & \lambda f\ .\ \underline{(S(\lambda x\ .\ f)}(\underline{\lambda x\ .\ (fx)})) & \text{(cases 2, 3)} \\
 & \rightarrow & \lambda f\ .\ (S(\underline{Kf})(S(\underline{\lambda x\ .\ f})(\underline{\lambda x\ .\ x}))) & \text{(cases 2, 1)} \\
 & \rightarrow & \underline{\lambda f\ .\ (S(Kf)(S(Kf)I))} & \text{(case 3)} \\
 & \rightarrow & \underline{(S(\lambda f\ .\ (S(Kf)))}(\underline{\lambda f\ .\ (S(Kf)I))}) & \text{(case 3)} \\
 & \rightarrow & (S(\underline{S(\lambda f\ .\ S)}(\underline{\lambda f\ .\ (Kf)}))(\lambda f\ .\ (S(Kf)I))) & \text{(cases 1, 3)} \\
 & \rightarrow & (S(S(KS)(S(\underline{\lambda f\ .\ K})(\underline{\lambda f\ .\ f})))(\lambda f\ .\ (S(Kf)I))) & \text{(cases 2, 1)} \\
 & \rightarrow & (S(S(KS)(S(\underline{KK})I))(\underline{\lambda f\ .\ (S(Kf)I)})) & \text{(case 3)} \\
 & \rightarrow & (S(S(KS)(S(KK)I))(\underline{S(\lambda f\ .\ (S(Kf)))}(\underline{\lambda f\ .\ I}))) & \text{(cases 3, 2)} \\
 & \rightarrow & (S(S(KS)(S(KK)I))(S(\underline{S(\lambda f\ .\ S)}(\underline{\lambda f\ .\ (Kf)}))(KI))) & \text{(cases 2, 3)} \\
 & \rightarrow & (S(S(KS)(S(KK)I))(S(S(KS)(S(\underline{\lambda f\ .\ K})(\underline{\lambda f\ .\ f})))(KI))) & \text{(cases 2, 1)} \\
 & \rightarrow & (S(S(KS)(S(KK)I))(S(S(KS)(S(\underline{KK})I))(\underline{KI}))) &
\end{array}
$$

Were we to expand this expression using the definitions for S, K, I given above and simplify the term, we would end up with the expression for C_2 again.

3.2 SK Combinator Calculus

The SKI combinator calculus is not minimal, it can still be reduced further. We can remove the I combinator by noting that $I = (SKK)$:

$$(SKKx)\ \rightarrow\ (Kx(Kx))\ \rightarrow\ x$$

If we substitute (SKK) wherever we find I, we can express any term in the lambda calculus using only the combinators S and K and parentheses for grouping.