Functions are Data!!!
Programming Languages (CSCI 3300)

Prof. Harley Eades (heades@gru.edu).

We have finally completed defining the λ-calculus. It is both simple, and powerful, well, at least I have been telling you that it is powerful. In this short note we are going to look at just how powerful it really is by defining data types as functions.

The idea is to treat certain λ-expressions has having a particular meaning – like being a boolean or a number – and then assuming that meaning define operations on that data (function). We determine a λ-expressions meaning by its behavior. That is, if the function acts like a boolean, then we will call it a boolean. This holds with my slogan that programs give data meaning.

Throughout this note we will define several λ-expressions and to make this easier we will use a special syntax to give λ-expressions names. An example is the following

$$\mathsf{id} := \lambda x.x$$

The name on the left like id is an alias of the expression on the right $\lambda x.x$ and so we will treat them as being identical. It is important to note that the name we give is not actually part of the λ-calculus, but is just for connivence.

The datatype encodings given here are known collectively as Church Encodings. Named after Alonzo Church – the inventor of the λ-calculus.

1 Functions are Booleans!

We have been using booleans quite a lot this semester so we are well versed in their operations. So what functions can we consider as being true, false, and an if-expression? We select the following for the first two:

$$\begin{aligned}\mathsf{True} &:= \lambda x.\lambda y.x\\ \mathsf{False} &:= \lambda x.\lambda y.x\end{aligned}$$

You might be wondering why we would choose such a definition? Well, it is because using these we can define the if-expression:

$$\mathsf{if} := \lambda b.\lambda x.\lambda y.b\ x\ y$$

The above λ-expression first takes in a boolean – well we are assuming that it does – b and then any other terms x and y where x is what happens if b is true, and y is what happens if b is false. As we mentioned above we completely choose these expressions based on their behavior. Consider an example:

$$\begin{aligned}\mathsf{if}\ \mathsf{True}\ (\lambda x.x)\ (\lambda x.x\ x) &= \underline{(\lambda b.\lambda x.\lambda y.b\ x\ y)\ \mathsf{True}}\ (\lambda x.x)\ (\lambda x.x\ x)\\ &\rightsquigarrow \underline{(\lambda x.\lambda y.\mathsf{True}\ x\ y)\ (\lambda x.x)}\ (\lambda x.x\ x)\\ &\rightsquigarrow \underline{(\lambda y.\mathsf{True}\ (\lambda x.x)\ y)\ (\lambda x.x\ x)}\\ &\rightsquigarrow \underline{\mathsf{True}\ (\lambda x.x)\ (\lambda x.x\ x)}\\ &= (\lambda x.\lambda y.x)\ (\lambda x.x)\ (\lambda x.x\ x)\\ &\rightsquigarrow \underline{(\lambda y.(\lambda x.x))\ (\lambda x.x\ x)}\\ &\rightsquigarrow (\lambda x.x)\end{aligned}$$

So if certainly acts like an if-expression when we give it True, but what about False?

$$
\begin{aligned}
\mathsf{if}\ \mathsf{False}\ (\lambda x.x)\ (\lambda x.x\ x) &= \underline{(\lambda b.\lambda x.\lambda y.b\ x\ y)\ \mathsf{False}}\ (\lambda x.x)\ (\lambda x.x\ x) \\
&\rightsquigarrow \underline{(\lambda x.\lambda y.\mathsf{False}\ x\ y)\ (\lambda x.x)}\ (\lambda x.x\ x) \\
&\rightsquigarrow \underline{(\lambda y.\mathsf{False}\ (\lambda x.x)\ y)\ (\lambda x.x\ x)} \\
&\rightsquigarrow \mathsf{False}\ (\lambda x.x)\ (\lambda x.x\ x) \\
&= \underline{(\lambda x.\lambda y.y)\ (\lambda x.x)}\ (\lambda x.x\ x) \\
&\rightsquigarrow \underline{(\lambda y.y)\ (\lambda x.x\ x)} \\
&\rightsquigarrow (\lambda x.x\ x)
\end{aligned}
$$

If you look at the definition if closely you will see that really all it does is call the boolean. Thus, it is the data True and False that are really determining the behavior of if. This is the essence of these types of datatype encodings.

Now that we have the if-expression we can define any other boolean operation we want. For example:

$$
\begin{aligned}
\mathsf{not} &= \lambda b.\mathsf{if}\ b\ \mathsf{False}\ \mathsf{True} \\
\mathsf{and} &= \lambda b_1.\lambda b_2.\mathsf{if}\ b_1\ (\mathsf{if}\ b_2\ \mathsf{True}\ \mathsf{False})\ \mathsf{False} \\
\mathsf{or} &= \lambda b_1.\lambda b_2.\mathsf{if}\ b_1\ (\mathsf{if}\ b_2\ \mathsf{True}\ \mathsf{True})\ \mathsf{False}
\end{aligned}
$$

Now these expressions get quite large:

$$
\begin{aligned}
\mathsf{and} &= \lambda b_1.\lambda b_2.b_1\ (\mathsf{if}\ b_2\ \mathsf{True}\ \mathsf{False})\ \mathsf{False} \\
&= \lambda b_1.\lambda b_2.\underline{(\lambda b.\lambda x.\lambda y.b\ x\ y)\ b_1}\ (\mathsf{if}\ b_2\ \mathsf{True}\ \mathsf{False})\ \mathsf{False} \\
&\rightsquigarrow \lambda b_1.\lambda b_2.\underline{(\lambda x.\lambda y.b_1\ x\ y)\ (\mathsf{if}\ b_2\ \mathsf{True}\ \mathsf{False})}\ \mathsf{False} \\
&\rightsquigarrow \lambda b_1.\lambda b_2.\underline{(\lambda y.b_1\ (\mathsf{if}\ b_2\ \mathsf{True}\ \mathsf{False})\ y)\ \mathsf{False}} \\
&\rightsquigarrow \lambda b_1.\lambda b_2.b_1\ (\mathsf{if}\ b_2\ \mathsf{True}\ \mathsf{False})\ \mathsf{False} \\
&= \lambda b_1.\lambda b_2.b_1\ (\underline{(\lambda b.\lambda x.\lambda y.b\ x\ y)\ b_2}\ \mathsf{True}\ \mathsf{False})\ \mathsf{False} \\
&\rightsquigarrow \lambda b_1.\lambda b_2.b_1\ (\underline{(\lambda x.\lambda y.b_2\ x\ y)\ \mathsf{True}}\ \mathsf{False})\ \mathsf{False} \\
&\rightsquigarrow \lambda b_1.\lambda b_2.b_1\ (\underline{(\lambda y.b_2\ \mathsf{True}\ y)\ \mathsf{False}})\ \mathsf{False} \\
&\rightsquigarrow \lambda b_1.\lambda b_2.b_1\ (b_2\ \mathsf{True}\ \mathsf{False})\ \mathsf{False} \\
&= \lambda b_1.\lambda b_2.b_1\ (b_2\ (\lambda x.\lambda y.x)\ (\lambda x.\lambda y.y))\ (\lambda x.\lambda y.y)
\end{aligned}
$$

Note that these expressions behave like the booleans if and only if they are given the booleans we have defined above as input. Now there is nothing stopping someone from giving these expressions any term they like. After all, there are no types! Now if something other than a boolean is given, then the behavior of these expressions will not be boolean-like.

Try out some other boolean expressions. In fact, we have just basically proven that the λ-calculus subsumes Iffy-Lang! The λ-calculus can do anything Iffy-Lang can do, and so much more.

2 Functions are Numbers

We can encode arithmetic in a similar fashion to the booleans, but first lets introduce a particular representation of arithmetic that will make understanding the Church-encoded numbers easier.

2.1 Peano Arithmetic

We can define all of the set of peano numbers, denoted $\mathbb{P}$, using the following rules:

$$
\frac{}{0 \in \mathbb{P}}\ \textsc{Zero} \qquad \frac{n \in \mathbb{P}}{(s\ n) \in \mathbb{P}}\ \textsc{Suc}
$$

We call the unary function s the successor function, and it should be thought of as taking in a natural number n and then returning $n + 1$.

A list of the first ten peano numbers:

0. 0
1. $s\,0$
2. $s\,s\,0$
3. $s\,s\,s\,0$
4. $s\,s\,s\,s\,0$
5. $s\,s\,s\,s\,s\,0$
6. $s\,s\,s\,s\,s\,s\,0$
7. $s\,s\,s\,s\,s\,s\,s\,0$
8. $s\,s\,s\,s\,s\,s\,s\,s\,0$
9. $s\,s\,s\,s\,s\,s\,s\,s\,s\,0$
10. $s\,s\,s\,s\,s\,s\,s\,s\,s\,s\,0$

Therefore, a natural number $n \in \mathbb{N} = \{0, 1, 2, 3, 4, \ldots\}$ can be defined as the peano number $\hat{n} = s^n 0$ where $s^n\,0 = \underbrace{s\,\cdots\,s}_{n}\,0$.

Peano numbers are a particular representation of the natural numbers, in fact, they look a lot like a datatype. So we better make sure we can define the usual arithmetic operations. First, we try and understand how we might define addition.

The addition operation on peano numbers is best described if we think in a recursive fashion. Suppose we want to add the numbers $\hat{3}$ and $\hat{4}$ to obtain $\hat{7}$. How might we do this? We can start by noticing that:

$$\hat{3} = s\,s\,s\,0 \quad \hat{4} = s\,s\,s\,s\,0$$

Now we want to obtain $\hat{7} = s\,s\,s\,s\,s\,s\,s\,0$, but notice that

$$s\,s\,s\,s\,s\,s\,s\,0 = s\,s\,s\,(s\,s\,s\,s\,0) = s\,s\,s\,(s^4\,0) = s^3\,(s^4\,0) = s^{3+4}\,0$$

How might we do this recursively? We can peal off a successor from the first number and add it to the second:

$$\begin{array}{ll} \hat{3} = s\,s\,s\,0 & \hat{4} = s\,s\,s\,s\,0 \\ \hat{2} = s\,s\,0 & \hat{5} = s\,s\,s\,s\,s\,0 \\ \hat{1} = s\,0 & \hat{6} = s\,s\,s\,s\,s\,s\,0 \\ \hat{0} = 0 & \hat{7} = s\,s\,s\,s\,s\,s\,s\,0 \end{array}$$

Then the final solution is on the right when we hit 0 on the left.

This is well and good, but rather inefficient. It turns out that we can do a little better. Examining the above will reveal that if we have a peano number like $\hat{4}$, then to add say $\hat{3}$ to it, we really just need to replace the zero in $\hat{4}$ with $\hat{3}$. Then we would obtain $\hat{7}$ in one fell swoop. Using λ-calculus notation if we could transform $\hat{4} = s\,s\,s\,s\,0$ into the function $\lambda z.s\,s\,s\,s\,z$, then $\hat{7} = (\lambda z.s\,s\,s\,s\,z)\,\hat{3}$. In fact, this is exactly how we will model addition in the λ-calculus.

How about multiplication? Instead of adding a single successor on the right for every successor on the left, we make a copy of all successors on the right, and add them to the number on the right for every successor on the left. The following is the step by step process of multiplying $\hat{3}$ and $\hat{4}$:

$$\begin{array}{ll} \hat{3} = s\,s\,s\,0 & \hat{4} = s\,s\,s\,s\,0 \\ \hat{2} = s\,s\,0 & \hat{8} = s\,s\,s\,s\,s\,s\,s\,s\,0 \\ \hat{1} = s\,0 & \hat{12} = s\,s\,s\,s\,s\,s\,s\,s\,s\,s\,s\,s\,0 \end{array}$$

Now our base case is one instead of zero. We can see that:

$$\hat{12} = s\,s\,s\,s\,s\,s\,s\,s\,s\,s\,s\,s\,0 = \hat{12} = s^4\,s\,s\,s\,s\,s\,s\,s\,s\,0 = s^4\,s^4\,s\,s\,s\,0 = s^4\,s^4\,s^4\,0 = s^{4+4+4}\,0 = s^{3*4}\,0$$

This implies that we can also define multiplication in terms of addition. We denote the addition we defined above as plus. Now using λ-calculus notation if we could transform $\hat{3} = s\,s\,s\,0$ into the function $\lambda s.s\,s,\ s\,0$, then we could simply apply the latter to the function $\lambda n.\mathsf{plus}\,4\,n$, then we would obtain

$$\begin{aligned}
\hat{12} &= (\lambda s.s\,s\ s\,0)\,(\lambda n.\mathsf{plus}\,4\,n) \\
&\rightsquigarrow (\lambda n.\mathsf{plus}\,4\,n)((\lambda n.\mathsf{plus}\,4\,n)((\lambda n.\mathsf{plus}\,4\,n)0)) \\
&\rightsquigarrow (\lambda n.\mathsf{plus}\,4\,n)((\lambda n.\mathsf{plus}\,4\,n)(\mathsf{plus}\,4\,0) \\
&\rightsquigarrow (\lambda n.\mathsf{plus}\,4\,n)(\mathsf{plus}\,4\,(\mathsf{plus}\,4\,0)) \\
&\rightsquigarrow \mathsf{plus}\,4\,(\mathsf{plus}\,4\,(\mathsf{plus}\,4\,0))
\end{aligned}$$

This final definition is exactly how we will define multiplication in the λ-calculus.

Now by combining the previous two patterns we can use multiplication to define exponentiation. As an example, consider how to compute $\hat{2}^{\hat{3}}$. If we could transform multiplication – denoted **mult** – into $\lambda n.\mathbf{mult}\,\hat{2}\,n$, and transform $\hat{3}$ into the expression $\lambda s.\lambda z.s\,s\,s\,z$, then we could define exponentiation as the following:

$$\begin{aligned}
\hat{8} &= (\lambda s.\lambda z.s\,s\,s\,z)\,(\lambda n.\mathbf{mult}\,\hat{2}\,n)\,\hat{1} \\
&\rightsquigarrow (\lambda z.(\lambda n.\mathbf{mult}\,\hat{2}\,n)\,((\lambda n.\mathbf{mult}\,\hat{2}\,n)\,((\lambda n.\mathbf{mult}\,\hat{2}\,n)\,z)))\,\hat{1} \\
&\rightsquigarrow (\lambda n.\mathbf{mult}\,\hat{2}\,n)\,((\lambda n.\mathbf{mult}\,\hat{2}\,n)\,((\lambda n.\mathbf{mult}\,\hat{2}\,n)\,\hat{1}))) \\
&\rightsquigarrow (\lambda n.\mathbf{mult}\,\hat{2}\,n)\,((\lambda n.\mathbf{mult}\,\hat{2}\,n)\,(\mathbf{mult}\,\hat{2}\,\hat{1})) \\
&\rightsquigarrow (\lambda n.\mathbf{mult}\,\hat{2}\,n)\,(\mathbf{mult}\,\hat{2}\,(\mathbf{mult}\,\hat{2}\,\hat{1})) \\
&\rightsquigarrow \mathbf{mult}\,\hat{2}\,(\mathbf{mult}\,\hat{2}\,(\mathbf{mult}\,\hat{2}\,\hat{1}))
\end{aligned}$$

So in order to define exponentiation we had to not only abstract either the zero or the successor function, but both! This gives us the final representation of peano numbers in the λ-calculus.

2.2 Peano Numbers in the λ-Calculus

We define the peano numbers in the λ-calculus as follows:

$$\begin{aligned}
\hat{0} &:= \lambda s.\lambda z.z \\
\hat{1} &:= \lambda s.\lambda z.s\,z \\
\hat{2} &:= \lambda s.\lambda z.s\,(s\,z) \\
\hat{3} &:= \lambda s.\lambda z.s\,(s\,(s\,z)) \\
\hat{4} &:= \lambda s.\lambda z.s\,(s\,(s\,(s\,z))) \\
&\vdots \\
\hat{n} &:= \lambda s.\lambda z.s^n\,z
\end{aligned}$$

Now we can define addition using the same principle as the previous section:

$$\mathsf{plus} := \lambda n_1.\lambda n_2.(\lambda s.\lambda z.n_2\,s\,(n_1\,s\,z))$$

Remember, just as we saw for the booleans, when we are defining a new operation on datatypes we are allowed to assume some of the inputs are actually values of the datatype. Hence, we assume n_1 and n_2 are actually peano numbers. So they are λ-abstractions. Thus, the definition of addition applies n_1 to s and z, which results in a term identical to the peano numbers used in the previous section, and then we use this term to replace the zero in n_2. Thus, obtaining a term with the correct number of successors.

Similarly, we can now use addition to define multiplication:

$$\mathbf{mult} := \lambda n_1.\lambda n_2.(n_2\,(\mathsf{plus}\,n_1)\,(\lambda s.\lambda z.z))$$

This definition is identical to the definition we used in the previous section.

Finally, we can define exponentiation:

$$\mathbf{exp} := \lambda n_1.\lambda n_2.(n_2\,(\mathbf{mult}\,n_1)\,(\lambda s.\lambda z.(s\,z)))$$