
CS 421 – Spring 2007

Lecture Notes Set 29:

Church Numerals

Elsa L. Gunter¹

Transcribed by: Pooja Mathur

Corresponding to Slides: 11-lambda-data (slides 8-18)

Made available: April 9, 2007

Revision 1.0

1 Change Log

1.0 Initial Release.

2 Representing data

2.1 Idea (slide 8)

Let's say we are declaring a type, we'll call it `blah` and `blah` has a bunch of constructors, C_1, C_2, \dots, C_k that all have their own set of input arguments. So it may look something like:

```
type blah =  
  C1 ty11 ty12 ... ty1n  
  | C2 ty21 ty22 ... ty2m  
  ...  
  | Ck tyk1 tyk2 ... tykp
```

So we have this type, and it is basically a Union of other types that already exist. So the question is now, how can we handle type constructors that take arguments. Now our match function is going to have to handle these arguments.

Now, let's assume that we have a piece of data that has some type. Then we also have a constructor applied to a bunch of terms. And we are trying to find out, if given a collection of arguments, what are we supposed to do? We want to pair up these arguments with their appropriate constructors in the type.

So, if you had:

$$C_1 t_{11} \dots t_{1n} \rightarrow \lambda x_1 \dots x_n. x_i t_{11} \dots t_{1n}$$

what this is trying to say is: the stuff before the arrow is that you are given all of your arguments. After the arrow, you have your cases, the x_i 's, and then we're saying that with that information, we are going to find a specific x_i (a specific case) that can be used with the arguments. So once we match an x_i to the arguments, we give the arguments to the case.

2.2 Pairs (slide 9)

So, using that idea, how can we represent pairs. Pairs have one constructor and takes two arguments. So from our general example above, we only are going to use C_1 . The match statement only needs one branch because there is only one thing you can do with the arguments you are given.

So the constructor is the comma and the comma is applied to an a and a b . So we ask, what function needs an a and a b and I'll give it the a and b . In general, pairing means...you give me a first component, you give me a second

¹© 2007, Share and Enjoy

component, then you give me a function that needs a first component and second component and I will apply the function to the first and second components.

2.3 Simple example

Let's play a game!! This game is called "represent that type"!!

We have the type... type data = I of int | S of string | F of float

The challenge in this game is to figure out how to represent S (a string constructor) applied to s (an argument) in λ terms. So, we have $S\ s =$. What should come next? On your marks...get set...go!

(Jeopardy music plays in background)

Nothing yet? Here's a hint, think about how you would describe a the three constructors in general, like we have been talking about so far...we would have $S\ s = \lambda$...

(More Jeopardy music)

Well, we have to list all our constructors, right? So let's give our three constructors the names, ic (for int case), sc (for string case) and fc (for float case), and we just list them after the λ like we did in the general case.

So far we have $S\ s = \lambda\ ic\ sc\ fc.$... But there's more...

If you didn't get the first part, you have a reprieve. This is now the second round, can you figure out what goes after the list of cases?

(Jeopardy music...)

Well, times up, I hope you had the following for your final answer. $S\ s = \lambda\ ic\ sc\ fc.\ sc\ s$

Bonus question: why is that the right answer?

(starting to get bored of the Jeopardy music...)

Answer: Like we have discussed for the general case, we apply the constructor we want to use to the arguments, and they figure out what to do from there. We wanted to make a string, so we used the string case (sc) and gave it the argument, which was s .

But wait, we're not done yet. There is still...the Lightning Round (everyone together: "oooooooo"). Lightning round question: what is S ? We found out what $S\ s$ equaled. But what about S , by itself?

(plays last segment of Jeopardy music...)

$S = \lambda\ s\ (\lambda\ ic\ sc\ fc.\ sc\ s) = \lambda\ s\ ic\ sc\ fc.\ sc\ s$

So this is saying, give me a string, give me a function that wants to do something with an integer, string, and float, and I will pick out the part that wants do something with the string and apply the string given to me to that function.

Well that's the end of the game. Thank you for playing "represent that type"!!

3 Functions over data

3.1 Pairs functions (slide 10)

Let's look at the function fst . This function takes in a pair and returns the first component of it. A pair can be thought of like a function that takes in two arguments itself, and then needs to be told what to do with those values.

That is fst is is a function that takes in a pair, and then has two cases, x and y .

$fst \equiv \lambda\ p.\ p\ (\lambda\ x\ y.\ x)$

The function part is saying that have the two cases, x and y , and we want to apply the pair to the x case. And what does the x case do, it returns the first component of the pair.

Now let's say we have the pair (u, v) , in λ terms that looks like: $((\lambda\ a\ b\ x.\ x\ a\ b)\ u\ v) \rightarrow (\lambda\ x.\ x\ u\ v)$. If we called the fst on this pair, what would we get? Well, with some reductions

$fst\ (u,v) \rightarrow$
 $(\lambda\ p.\ p\ (\lambda\ x\ y.\ x))(\lambda\ x.\ x\ u\ v) \rightarrow$
 $(\lambda\ x.\ x\ u\ v)(\lambda\ x\ y.\ x) \rightarrow$
 $((\lambda\ x\ y.\ x)\ u\ v) \rightarrow$
 $((\lambda\ y.\ u)\ v) \rightarrow$
 $u \rightarrow$

So, the fst function and the pair seem to switch sides a couple times, but that is just because they are replacing variables in the other term.

The function snd works very similarly, the only difference is at the end when we threw away the y before, we would be throwing away the x here and keeping everything else the same.

4 Recursive Data

4.1 Representation (slide 11)

Now, we are going to go throw and look at how to include things like lists. With this we have to worry about some type being part of it's own constructor. So when we try to declare this type, we have to make another call to it, to enable us to continue.

In the general case this looks like:

$$C_i \rightarrow \lambda t_{i1} \dots t_{ij} x_1 \dots x_n. x_i t_{i1} \dots (t_{ih} x_1 \dots x_n) \dots t_{ij}$$

Basically, this is saying the usual in the beginning. The t_j 's are the arguments and the x_i 's are the cases. Then we pick a specific x_i and apply those arguments. The different here is that those arguments might contain a component that is the type of what we are trying to construct (namely, t_{ih}). That argument causes us to do another application.

So let's look at an example of this. Lists for example can have recursive structures and non-recursive structures. Lists are defined by the cons function that takes two arguments, an element of the list, and the rest of the list. That would look like: $\text{cons} \rightarrow \lambda h t \lambda n c. c h (t c n)$. What this is saying is there are two cases for lists. Either you have the head and tail case, or you have the nil case. If we have the head/tail case, we call cons (the c) on the head (the h) and what ever is left. What's left you may be asking, well, it's the $t c n$ part. That is the recursive call. We call the cons function again, on the tail and nil parts, to finish cons-ing everything together. And that's how recursion works here, you end up applying arguments to all the different cases.

4.2 Examples (slides 12-14)

We will start by looking at the natural numbers which will be represented by a data type. Here, natural numbers are either 0 or the successor applied to the natural numbers.

So to represent this ($\text{nat} = \text{Suc nat} \mid 0$) in λ terms, we have to have two cases. In the 0 case, we don't have to do any recursion, we don't have to worry about anything else, all we do is look at what we have and give it back, the identity function, $\lambda f x. x$. So, x is the base case.

Successors are represented like this: $\text{Suc } n = \lambda f x. f (n f x)$. This is saying that in this recursive case, we will recurse on n . This means that f is the recursive case. And to recurse on n , we will apply both cases to it. So, $(n f x)$ is the recursive call. So the $(n f x)$ will evaluate to something, and then after the recursive call returns, we will apply f to that value. In the end, what this comes down to is, simply that we will perform f, n times.

So, what do we get when we try $\text{Suc } 0$? Well, it's going to work like this:

$$\begin{aligned} \text{Suc } 0 &= (\lambda n f x. f (n f x))(\lambda f x. x) \rightarrow \\ &\lambda f x. f ((\lambda f x. x) f x) \rightarrow \\ &\lambda f x. f ((\lambda x. x) x) \rightarrow \\ &\lambda f x. f x \end{aligned}$$

We started with the Successor function $(\lambda n f x. f (n f x))$ being applied to the 0 $(\lambda f x. x)$. With some reductions, we were able to get to our base case.

Try doing $\text{Suc } (\text{Suc } 0)$ and see if you can come up with the solution of $\lambda f x. f (f x)$. The step by step solution can be found on the slides.

So in general you will see that $\text{Suc}^n 0 = \lambda f x. f^n x$, That is, applying the Successor function, n times to 0 will translate to applying f to 0, n times.

4.3 Addition (slide 15)

Now we want to be able to do things like addition. Well we just learned that n was $\equiv \lambda f x. f^n x$. And say we had another value m, that was equivalent to $\lambda f x. f^m x$. If we want to add n to m we would be saying that we want to

apply f to x $n+m$ times. That is equivalent to applying f , n times to f applied to x , m times.
This ends up looking like: $+ \equiv \lambda n m f x. n f (m f x)$

4.4 Multiplication (slide 16)

This follows the same idea. $n * m$ is the same as applying f to x $n*m$ times.
This ends up looking like: $* \equiv \lambda n m f x. n f (m f) x$

4.5 Primitive Recursion (slide 17)

Now let's step it up a bit. We are going to now looking at the idea of folding.

So, fold needs three arguments, f , z , and n . In the base case, the value, in this case, z , should be returned. Otherwise it's the recursive case, and with that you apply f once but first you make an another recursive call.

So, fold in general looks like $\lambda f x z. n f z$ where f is the plan on what is to be done in the successor case, and then the z is what should be done in the base case.