Computing with Inductive Data Types

Outline

We want to add datatypes to λ-calculus (natural numbers, lists, booleans, pairs, etc.) so that we can compute with them.

There are different ways to do this:

1. Encoding without extending the calculus. (Church encoding.)
2. Adding recursion operators with reductions.
3. Adding matching operators with reductions.

Theoretically, $1, 2$ are better than 3.

Practically, 3 is better than 2 better than 1.

Independent of the chosen approach, we will need to understand inductive data types.

Induction for Natural Numbers

Let's have a look at natural numbers:

The principle of complete induction is defined as follows (translated from an analysis text book):

Let E be property. If

1. 0 has property E,
2. n has property E implies that $(n+1)$ also has property E,

then all natural numbers have property E.

One can use it to prove e.g.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Induction for Natural Numbers (2)

Why is this possible?

Induction for Natural Numbers (3)

Because all natural numbers are either 0, or reachable from 0 by a finite number of increasements.

There are no hidden, 'dark' natural numbers.

Recursion

Next comes recursion:

```
double int fact( unsigned int x )
{
   if( x == 0 )
      return 1.0;
   else
      return x * fact( x - 1 );
}
```

Why is this possible?

Again, there are no 'dark' natural numbers. (every number has a path to 0.)

But there is something more:

Every natural number has a unique path to 0.

Without this uniqueness property, induction (as logical rule) would still be valid.

At the same time recursion would not work anymore, because `fact` would not necessariiy be a function.

For example, when $3 + 1 = 3$, then $\text{fact}(3) = \text{fact}(4) = 4.\text{fact}(3)$.

Freely Generated Datatypes

We say that the natural numbers are freely generated by 0 and succ.

In Haskell notation:

```
data Natural
   Zero | Succ Natural
```

The functions that construct the element of the datatype are called constructors.

Do you know other Freely Generated Datatypes? What are their constructors?

Booleans, Enumeration Types, Lists

The Booleans: We have **t** and **f**.

There are no other Booleans, and **f** $\neq$ **t**.

In general, all finite enumeration types are freely generated.
Otherwise, `switch` statements would not make sense.

Recursion in its Basic Form

Recursion is a way of defining functions from natural numbers to something. We need two things:

1. The value f_0 for 0.
2. How to get a value for $n+1$ if we have a value for n. For this, we need a function f_s which takes n, the value for n, and constructs the value for $n+1$.

$(F\ 0) = f_0$.

$(F\ (\text{succ}\ 0)) = (f_s\ 0\ f_0)$.

$(F\ (\text{succ}\ (\text{succ}\ 0))) = (f_s\ (\text{succ}\ 0)\ (f_s\ 0\ f_0))$.

Recursion Operator

$$
\begin{array}{lcl}
(\mathbf{rec}_N\ f_0\ f_s\ 0) & \Rightarrow & f_0, \\
(\mathbf{rec}_N\ f_0\ f_s\ (\mathbf{succ}\ n)) & \Rightarrow & (f_s\ n\ (\mathbf{rec}_N\ f_0\ f_s\ n)).
\end{array}
$$

Let's define a few well-known functions:

$m + n$**:** We use Currying, so we define $(+\ m)$:

$$f_0 = m$$
$$f_s = \lambda m_1\ \lambda m_2\ (\mathbf{succ}\ m_2)$$

$$+ = \lambda m\ (\mathbf{rec}_N\ m\ \lambda m_1\ \lambda m_2\ (\mathbf{succ}\ m_2)\ )$$

$m - n$**:** First define $\mathbf{pred} = (\mathbf{rec}_N\ 0\ \lambda m_1\ \lambda m_2\ m_1)$.

After that

$$f_0 = m$$
$$f_s = \lambda m_1\ \lambda m_2\ (\mathbf{pred}\ m_2)$$

$$- = \lambda m\ (\mathbf{rec}_N\ m\ \lambda m_1\ \lambda m_2\ (\mathbf{pred}\ m_2)\ )$$

$\times$ Multiplication is repeated addition. We use Currying again, so we define $(\times\ n)$:

$$f_0 = 0$$
$$f_s = \lambda m_1\ \lambda m_2\ (+\ m_2\ n)$$
$$\times = \lambda m\ (\mathbf{rec}_N\ 0\ \lambda m_1\ \lambda m_2\ (+\ m_2\ m)\ )$$

Booleans

What do we know about Booleans?

- There are two of them.
- They are called **f** and **t**.
- They are distinct.
- There are no other Booleans.

Hence: They are freely generated.

What does one need to define a function from Booleans?

Equality

The smartest way that I could come up with, is by using $\geq$.

One has $n \geq m$ iff $(\mathbf{pred}^n\ m) = 0$.

So, we can define

$$
\begin{array}{lcl}
\mathbf{eq} & = & (\mathbf{and}\ (\leq\ m\ n)\ (\leq\ n\ m)) \\
\mathbf{iszero} & = & (\mathbf{rec}_N\ \mathbf{t}\ (\lambda m\, \lambda n\ \mathbf{f})) \\
\leq & = & \lambda m\, \lambda n\ (\mathbf{iszero}\ (\mathbf{rec}_N\ m\ (\lambda m\, \lambda n\ (\mathbf{pred}\ n))\ n))
\end{array}
$$

Recursion Operator for Booleans

We need:

A value f_f for $\mathbf{f}$.

A value f_t for $\mathbf{t}$.

$$\begin{aligned}(\mathbf{rec}_B\ f_f\ f_t\ \mathbf{f}) &\Rightarrow f_f,\\ (\mathbf{rec}_B\ f_f\ f_t\ \mathbf{t}) &\Rightarrow f_t.\end{aligned}$$

One can define other Boolean operators, for example

$$\begin{aligned}&\mathbf{if} = \lambda b\ \lambda t_1\ \lambda t_2\ (\mathbf{rec}_B\ t_2\ t_1\ b)\\ &\mathbf{not} = (\mathbf{rec}_B\ \mathbf{t}\ \mathbf{f})\\ &\mathbf{or} = \lambda b_1\ \lambda b_2\ (\mathbf{rec}_B\ (\mathbf{rec}_B\ \mathbf{f}\ \mathbf{t}\ b_2)\ \mathbf{t}\ b_1)\\ &\mathbf{and} = \lambda b_1\ \lambda b_2\ (\mathbf{rec}_B\ \mathbf{f}\ (\mathbf{rec}_B\ \mathbf{f}\ \mathbf{t}\ b_2)\ b_1)\end{aligned}$$

Lists

Lists are constructed by **nil** and **cons**.

Lists are usually written with [and] :

$$
\begin{array}{lcl}
[\,] & = & \mathbf{nil} \\
[n_1] & = & (\mathbf{cons}\ n_1\ \mathbf{nil}) \\
[n_1, n_2] & = & (\mathbf{cons}\ n_1\ (\mathbf{cons}\ n_2\ \mathbf{nil})) \\
[n_1, n_2, n_3] & = & (\mathbf{cons}\ n_1\ (\mathbf{cons}\ n_2\ (\mathbf{cons}\ n_3\ \mathbf{nil})))
\end{array}
$$

The recursion operator $\mathbf{rec}_L$ has the following reductions:

$$
\begin{array}{lcl}
(\mathbf{rec}_L\ f_n\ f_c\ \mathbf{nil}) & \Rightarrow & f_n, \\
(\mathbf{rec}_L\ f_n\ f_c\ (\mathbf{cons}\ f\ r)) & \Rightarrow & (f_c\ f\ r\ (\mathbf{rec}_L\ f_n\ f_c\ r)).
\end{array}
$$

Lists (2)

$$\textbf{length} = (\textbf{rec}_L\ 0\ (\lambda x\ \lambda a\ \lambda n\ (\textbf{succ}\ n))\ )$$

$$\textbf{append} = \lambda l_1\ \lambda l_2\ (\textbf{rec}_L\ l_2\ (\lambda f\ \lambda r\ \lambda x\ (\textbf{cons}\ f\ x))\ l_1\ ).$$

By now, we probably have strong desire for a typed language with an automatic type checker. Fortunately, such language exists: Haskell.

Pairs and Unions

Pairs and Unions are also recursive datatypes.

Pairs are constructed by **pair** : The recursion operator $\mathbf{rec}_P$ for pair has the following reduction:

$$(\mathbf{rec}_P\ f_p\ (\mathbf{pair}\ x\ y)) \Rightarrow (f_p\ x\ y).$$

One can for example define:

$$\begin{aligned} \mathbf{first} = &\quad (\mathbf{rec}_P\ (\lambda x\ \lambda y\ x)) \\ \mathbf{second} = &\quad (\mathbf{rec}_P\ (\lambda x\ \lambda y\ y)) \end{aligned}$$

Union has two constructors **union1** and **union2**.

$$\begin{aligned} (\mathbf{rec}_U\ f_1\ f_2\ (\mathbf{union1}\ x)) &\Rightarrow (f_1\ x) \\ (\mathbf{rec}_U\ f_1\ f_2\ (\mathbf{union2}\ y)) &\Rightarrow (f_2\ y) \end{aligned}$$

If you can handle both types, you can handle the union.

Optional

An optional object is an object that is either present or not present. We have two constructors:

- (**just** t) means that we have t.
- **none** means that we have nothing.

The recursion operator is defined by

$$\begin{aligned} (\mathbf{rec}_O\ f_j\ f_n\ (\mathbf{just}\ t)\ ) &\Rightarrow (f_j\ t) \\ (\mathbf{rec}_O\ f_j\ f_n\ \mathbf{none}\ ) &\Rightarrow f_n \end{aligned}$$

Now can redefine pred as a partial function:

$$\mathbf{pred} = (\mathbf{rec}_N\ \mathbf{none}\ \lambda m_1\ \lambda m_2\ (\mathbf{just}\ m_1)).$$

Church Encoding

In text books and on the internet, you can find something called Church Encoding.

It works for all recursive data types.

Instead of adding **rec**-operators and constructors to the calculus, one takes the possibility to define them as definition:

The essential part of 'being a natural number' is 'being able to be used in recursive definitions', so the number if a function that returns the result of the recursion.

$$
\begin{array}{lcl}
0 & = & \lambda f_0 \lambda f_s \; f_0 \\
1 & = & \lambda f_0 \lambda f_s \; (f_s \; 0 \; f_0) \\
2 & = & \lambda f_0 \lambda f_s \; (f_s \; 1 \; (f_s \; 0 \; f_0)) \\
\mathrm{succ} & = & \lambda n \; \lambda f_0 \lambda f_s \; (f_s \; n \; (n \; f_0 \; f_s))
\end{array}
$$

Church Encoding (2)

Usually (in almost all textbooks), the first argument of f_s is omitted. One gets

$$\begin{array}{lcl} 0 & = & \lambda f_0 \lambda f_s\ f_0 \\ 1 & = & \lambda f_0 \lambda f_s\ (f_s\ f_0) \\ 2 & = & \lambda f_0 \lambda f_s\ (f_s\ (f_s\ f_0)) \\ \text{succ} & = & \lambda n\ \lambda f_0 \lambda f_s\ (f_s\ (n\ f_0\ f_s)) \end{array}$$

This definition is based on the primitive recursion operator:

$$\begin{array}{lcl} (\mathbf{rec}_N\ f_0\ f_s\ 0) & \Rightarrow & f_0, \\ (\mathbf{rec}_N\ f_0\ f_s\ (\mathbf{succ}\ n)) & \Rightarrow & (f_s\ (\mathbf{rec}_N\ f_0\ f_s\ n)). \end{array}$$

Most functions can be easily defined with primitive recursion, but defining for example **pred** is hard.

The other recursion operator can be recovered by using pairs.

Church Encoding (3)

$$\begin{aligned} \mathbf{f} &= \lambda f_f \lambda f_t \; f_f \\ \mathbf{t} &= \lambda f_f \lambda f_t \; f_t \end{aligned}$$