Church Numerals

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

Objectives

- Use lambda calculus to implement integers and booleans.
  - Define some operations on Church numerals: inc, plus, times.
  - Explain how to represent boolean operations: and, or, not, if.
- Use lambda calculus to implement arbitrary types.

What Is a Number?

- The lambda calculus doesn't have numbers.
- A number *n* can be thought of as a potential: someday we are going to do something *n* times.

Some Church Numerals

```
f0 = \f-> \x-> x
f1 = \f-> \x-> f x
f2 = \f-> \x-> f (f x)
f3 = \f-> \x-> f (f (f x))
```

```
Prelude> let show m = m (+1) 0
Prelude> show (\f x -> f (f x))
2
```


Incrementing Church Numerals, 0

- To increment a Church numeral, what do we want to do?

Running Example

```
finc = undefined
```

Incrementing Church Numerals, 1

- To increment a Church numeral, what do we want to do?
- First step, take the Church numeral you want to increment.

Running Example

```
finc = \m -> undefined
```


Incrementing Church Numerals, 2

- To increment a Church numeral, what do we want to do?
- First step, take the Church numeral you want to increment.
- Second step, *return* a Church numeral representing your result.

Running Example

```
finc = \m -> \f x -> undefined
```


Incrementing Church Numerals, 3

- To increment a Church numeral, what do we want to do?
- First step, take the Church numeral you want to increment.
- Second step, *return* a Church numeral representing your result.
- Third step, apply f to x, m times.

Running Example

```
finc = \m -> \f x -> m f x
```


Incrementing Church Numerals, 4

- To increment a Church numeral, what do we want to do?
- First step, take the Church numeral you want to increment.
- Second step, *return* a Church numeral representing your result.
- Third step, apply f to x, m times.
- Finally, apply f once more to the result.

Running Example

```
finc = \m -> \f x -> f (m f x)
```

Adding Church Numerals

- Similar reasoning can yield addition and multiplication.
- Here is addition. Can you figure our multiplication? Hint: What does (nf) do?
- Subtraction is a bit more tricky.

Running Example

```
fadd m n = \f x -> m f (n f x)
```


Implementing Booleans

- Church numerals represented integers as a potential number of actions.
- Church Booleans represent true and false as a choice.

$$
\begin{aligned}
T &\equiv \lambda ab.a \\
F &\equiv \lambda ab.b
\end{aligned}
$$

```
true = \ a b -> a
false = \ a b -> b
showb f = f True False
```

- Type these into a REPL and try them out!
- Next slide: `and` and `or`. Try to figure it out before going ahead!

And and Or

- There are a couple of ways to do it.

$$
\begin{aligned}
and &\equiv \lambda xy.xyF \\
or &\equiv \lambda xy.xTy \\
if &\equiv \lambda cte.cte
\end{aligned}
$$

```
and = \x y -> x y false
or = \x y -> x true y
cif = \c t e -> c t e
```


Representing Arbitrary Types

- Suppose we have an algebraic data type with n constructors.
- Then the Church representation is an abstraction that takes n parameters.
- Each parameter represents one of the constructors.

$$
\begin{aligned}
T &\equiv \lambda ab.a \\
F &\equiv \lambda ab.b
\end{aligned}
$$

The Maybe Type

- The `Maybe` type has two constructors: `Just` and `Nothing`.

```haskell
data Maybe a = Just a
             | Nothing
```

- Can you give the lambda-calculus representation for `Just 3`?

$$\begin{aligned} \mathit{Just}\ a &\equiv \lambda jn.ja \\ \mathit{Nothing} &\equiv \lambda jn.n \end{aligned}$$

The Maybe Type

- The `Maybe` type has two constructors: `Just` and `Nothing`.

```haskell
data Maybe a = Just a
             | Nothing
```

- Can you give the lambda-calculus representation for `Just 3`?

$$\begin{aligned} \mathit{Just}\ a &\equiv \lambda jn.ja \\ \mathit{Nothing} &\equiv \lambda jn.n \end{aligned}$$

$$\mathit{Just}\ 3 \equiv \lambda jn.j \lambda fx.f(f(fx))$$

- Try to figure out how to represent linked lists ....

Linked Lists

- A list has two constructors: `Cons` and `Nil`.

```haskell
data List a = Cons a (List a)
            | Nil
```

- Can you give the lambda-calculus representation for `Cons True (Cons False Nil)`?

$$\begin{aligned} \mathit{Cons}\ x\ y &\equiv \lambda cn.cxy \\ \mathit{Nil} &\equiv \lambda cn.n \end{aligned}$$

Linked Lists

- A list has two constructors: `Cons` and `Nil`.

```haskell
data List a = Cons a (List a)
            | Nil
```

- Can you give the lambda-calculus representation for `Cons True (Cons False Nil)`?

$$\begin{aligned} \mathit{Cons}\ x\ y &\equiv \lambda cn.cxy \\ \mathit{Nil} &\equiv \lambda cn.n \end{aligned}$$

$$\begin{aligned} & \lambda c_1 n_1.c_1(\lambda ab.a)(\lambda c_2 n_2.c_2(\lambda ab.b)(\lambda c_3 n_3.n_3)) \\ \mathit{or}\ldots\ & \lambda cn.c(\lambda ab.a)(c(\lambda ab.b)n) \end{aligned}$$

- Write a function `length` that determines the length of one of these lists. Assume you are allowed to use recursion. (Note, Haskell's type system will not let you write this.)

Length

$$\begin{aligned} \mathit{Cons}\ x\ y &\equiv \lambda cn.cxy \\ \mathit{Nil} &\equiv \lambda cn.n \end{aligned}$$

$$\mathit{Length}\ x = x(\lambda xy.\mathit{inc}\ (\mathit{Length}\ y))\ \mathit{zero}$$

Higher Order Abstract Syntax

- It is possible to represent lambda-calculus in lambda calculus!
- We can let variables represent themselves.
- This is a non-recursive version:

$$\begin{aligned} M &= \lambda fa.[\![M]\!]^f_a \\ [\![\mathit{Var}\ x]\!]^f_a &= x \\ [\![\mathit{Abs}\ x\ M]\!]^f_a &\equiv f\lambda x.[\![M]\!]^f_a \\ [\![\mathit{App}\ e_1\ e_2]\!]^f_a &\equiv a[\![e_1]\!]^f_a[\![e_2]\!]^f_a \end{aligned}$$

- You can then write an interpreter for this!

Higher Order Abstract Syntax

- It is possible to represent lambda-calculus in lambda calculus!
- We can let variables represent themselves.
- This is a non-recursive version:

$$\begin{aligned} M &= \lambda fa.[\![M]\!]^f_a \\ [\![\mathit{Var}\ x]\!]^f_a &= x \\ [\![\mathit{Abs}\ x\ M]\!]^f_a &\equiv f\lambda x.[\![M]\!]^f_a \\ [\![\mathit{App}\ e_1\ e_2]\!]^f_a &\equiv a[\![e_1]\!]^f_a[\![e_2]\!]^f_a \end{aligned}$$

- You can then write an interpreter for this!
  - Abstraction: $\lambda x.x$
  - Application: $\lambda e_1 e_2.e_1 e_2$