The Lambda Calculus

The lambda calculus

- Powerful computation mechanism
- 3 simple formation rules
- 2 simple operations
- extremely expressive

Syntax

A term in the calculus has one of the following three forms.

Let t be a term, and v be a variable

Then

v	is a term
t t	is a tem
\ v . t	is a term

```
\ x . x

\ z . \ s . s z

\ n . snd (n (pair zero zero)
             (\ x . pair (succ (fst x)) (fst x)))

\ f . (\ x . f (x x)) (\ x . f (x x))
```

Variables

The variables in a term can be computed using the following algorithm

```
varsOf v = {v}
varsOf (x y) = varsOf x `union` varsOf y
varsOf (\ x . e) = {x} `union` varsOf e
```

Examples

```
varsOf (\ x . x) = {x}

varsOf (\ z . \ s . s z) = {s,z}

varsOf
 (\ n . snd (n (pair zero zero)
                (\ x . pair (succ (fst x)) (fst x))))
   = {n,snd,pair,zero,x,succ,fst}
```

Free Variables

The free variables can be computed using the following algorithm

```
freeOf v = {v}
freeOf (x y) = freeOf x `union` freeOf y
freeOf (\ x . e) = freeOf e -  {x}
```

Examples

```
freeOf (\ z . \ s . s z) = { }

freeOf
 (\ n . snd (n (pair zero zero)
               (\ x . pair (succ (fst x)) (fst x))))
   = {snd,pair,zero,succ,fst}
```

Alpha renaming

Terms that differ only in the name of their bound variables are considered equal.

(\ z . \ s . s z) = (\ a . \ b. b a)

Substitution

We can think of substituting a term for a variable in a lambda-term

sub x (\ y . y) (f x z) → (f (\ y . y) z)

We must be careful if the term we are substituting into has a lambda inside

sub x (g y) (\ y. f x y) → (\ y . f (g y) y)

|| $\nparallel$

sub x (g y) (\ w. f x w) → (\ w . f (g y) w)

Algorithm

sub v_1 new (v) = if v_1 = v then new else v

sub v_1 new (x y) =

(sub v_1 new x) (sub v_1 new y)

sub v_1 new (\ v . e) =

\ v' . sub v_1 new (sub v v' e)

Where v' is a new variable not in the free variables of new

Example

sub x (g y) (\ y. f x y) →

\ y’ . sub x (g y) (sub y y’ (f x y)) →

\ y’ . sub x (g y) (f x y’) →

\ y’ . f (g y) y’

```
sub v1 new (v) = if v1 = n then new else v
sub v1 new (x y) =
    (sub v1 new x) (sub v1 new y)
sub v1 new (\ v . e) =
  \ v’ . sub v1 new (sub v v’ e)
```

Beta - reduction

If we have a term with the form

(\ x . e) v

then we can take a step to get

sub x v e

Example

(\ n . \ z . \ s . n (s z) s) (\ z . \ s . z)

\ z . \ s . (\ z . \ s . z) (s z) s

\ z . \ s . (\ z . \ s . z) (s z) s

\ z . \ s . (\ s0 . s z) s

\ z . \ s . s z

What good is this?

How can we possible compute with this?

We have no data to manipulate

1. no numbers
2. no data-structures
3. no control structures (if-then-else, loops)

Answer

Use what we have to build these from scratch!

The church numerals

We can encode the natural numbers

```
zero = \ z . \ s . z
one = \ z . \ s . s z
two = \ z . \ s . s (s z)
three = \ z . \ s . s (s (s z))
four = \ z . \ s . s (s (s (s z))
```

What is the pattern here?

Can we use this.

The succ function

succ (one) → two

succ (\ z . \ s . s z) → (\ z . \ s . s (s z))

Can we write this? Lets try

succ = \ n . ???

Succ

succ = \ n . \ z . \ s . n (s z) s

succ one

(\ n . \ z . \ s . n (s z) s) one

\ z . \ s . one (s z) s

\ z . \ s . (\ z . \ s . s z) (s z) s

\ z . \ s . (\ s0 . s0 (s z)) s

\ z . \ s . s (s z)

Can we write the Add function?

add = \ x . \ y . \ z . \ s . x (y z s) s

what about multiply?

Can we build the booleans

We'll need

```
true:: Bool

false:: Bool

if:: Bool -> x -> x -> x
```

true = \ t . \ f . t

false = \ t . \ f . f

if = \ b . \ then . \ else . b then else

Lets try it out

if false two one

What about pairs?

we’ll need

```
pair:: a -> b -> Pair a b

fst:: Pair a b -> a

snd:: Pair a b -> b
```

```
pair = \ x . \ y . \ k . k x y

fst = \ p . p (\ x . \ y . x)

snd = \ p . p (\ x . \ y . y)
```

Can we write the pred function

```
pred = \ n . snd
            (n (pair zero zero)
            (\ x . pair (succ (fst x)) (fst x)))
```

How does this work?

Think about this

(\ x . x x) (\ x . x x)

The Y combinator

y = \ f0 . (\ x . f0 (x x)) (\ x . f0 (x x))

what happens if we apply? y f