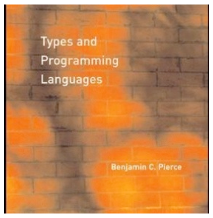
Lambda-Calculus (cont): Fixpoints, Naming

Lecture 10
CS 565



Recursion and Divergence

Consider the application:

$$\Omega \equiv ((\lambda\ x.(x\ x))\ (\lambda\ x.(x\ x)))$$

Ω evaluates to itself in one step.

It has no normal form.

A lambda term is in normal form if it does not contain any redex (i.e., a term that is subject to β-reduction)

Now, consider: $Y \equiv ((\lambda\ x.(f\ (x\ x)))(\lambda\ x.(f\ (x\ x))))$

$Y \rightarrow$

$(f\ ((\lambda\ x.(f\ (x\ x)))\ (\lambda\ x.(f\ (x\ x))))) \rightarrow$

$(f\ (f\ (\lambda\ x.(f\ (x\ x)))\ (\lambda\ x.(f\ (x\ x)))))) \rightarrow$

...

$(f(f(\ldots(f\ (\lambda\ x.(f\ (x\ x)))(\lambda\ x.(f(x\ x)))\ldots)))$

Normal forms and order of evaluation

No expression can be converted to two distinct normal forms (Church-Rosser Theorem 1)

Is there an order of evaluation guaranteed to terminate whenever a particular expression is reducible to normal form?

- Normal-order: leftmost, outermost reduction: no expression in the argument position of a redex is reduced until the redex is reduced
- If there is a reduction from `A` to `B` and `B` is in normal form, then there exists a normal order reduction from `A` to `B` (Church-Rosser Theorem 2)

Recursion

The previous definition applies f an infinite number of times

Basis for iterated application

But, how can we slow its rate of unfolding?

Consider:

```
Ωv ≡ (λ y. ((λ x. (λ y. (x x y)))
            (λ x . (λ y. (x x y)))
             y))
```

Ωv is in normal form. However, if it is applied to an argument it diverges.

Recursion (cont)

```
(Ωv v) →

((λ y. ((λ x. (λ y. (x x y)))
          (λ x. (λ y. (x x y)))
          y) v) →

((λ y. ((λ x. (λ y. (x x y)))
 (λ x. (λ y. (x x y)))
 y) v)  →
...
```

Recursion (cont)

Now, consider

```
  Zf ≡ (λ y. ((λ x . (f (λ y. (x x y))))
                     (λ x. (f (λ y. (x x y))))
                     y))
```

If we apply Zf to an argument:

```
((λ y. ((λ x. (f (λ y. (x x y))))
          (λ x. (f (λ y. (x x y))))
          y) v) →
(f (λ y .((λ x.(f (λ y.(x x y))))
         (λ x. (f (λ y. (x x y))))
          y)) v) →
```

Since the arguments to f are all values, this expression is equivalent to: f Z_f v

Recursion (cont)

How do we apply these insights?

```
f ≡ λ fact.
       λ n. if n = 0
                then 1
                else n * (fact (n – 1))
```

We can use Zf to turn f into a real factorial function:

Fixpoints

$Z_f\ 3 \rightarrow$

$f\ Z_f\ 3 \rightarrow$

$(\lambda\ fact.\ \lambda\ n.\ ...)\ Z_f\ 3 \rightarrow$

$\text{if } 3 = 0 \text{ then } 1 \text{ else } 3 * (Z_f\ 2) \rightarrow$

$3 * (f\ Z_f\ 2)$

...

We'll stop when n = 0

Fixpoints

Define $Z = \lambda\ f.\ Z_f$

Now, Z defines a fixpoint for any f:

```
Z ≡ λ f. (λ y. ((λ x. (f (λ y. (x x y))))
                 (λ x. (f (λ y. (x x y))))
                  y))
```

Z computes the least fixpoint of a function.

Fixpoints and order of evaluation

Consider an alternative definition:

```
Y ≡ λh.(λx.h( x x))(λx.h(x x))
```

- What happens if we apply Y to f (the factorial functional) with argument 3?
- Under normal-order evaluation:

```
Y f 3 ≡ (λx. f(x x))(λx. f(x x)) 3 →
  f ((λx. f(x x))(λx. f(x x))) 3
```

- What happens under applicative-order?

Naming and substitution

Although we claimed that lambda calculus essentially manipulates functions (it does), we've spent a lot of time thinking about variables

- substitutions
- free variables
- equivalence upto renaming

Implementations must consider these issues seriously

- Rename bound variables when performing substitutions with "fresh" names.
- Impose a condition that all bound variables be distinct from each other, and other free variables.
- Derive a canonical representation that does not require renaming at all.

Terms and Contexts

De Brujin indices:

- Have variable occurrences "point" directly to their binders rather than referring to them by name.
- Do so by replacing variable occurrences with numbers:

  number k stands for "the variable bound by the kth enclosing λ-term

  Example: `λ x.λ y.x (y x) ≡ λ.λ.1(0 1)`

  Similar to static offsets in an activation record or display.

Examples

```
identity ≡ λ x. x ≡ λ .0

true ≡ λ x. λ y. x ≡ λ.λ. 1

false ≡ λ x. λ y. y ≡ λ.λ. 0

two ≡ λ s. λ z. s (s z) ≡ λ . λ . (1 (1 0))
```

Contexts

How do we replace free variables with their binders?

Assume an ordered context listing all free variables that can occur, and map free variables to their index in this context (counting right to left)

Context: a, b

```
a ↦ 1, b ↦ 0
λ x. a  ≡ λ . 2
λ x. b ≡ λ. 1
λ x.b (λ y. a) ≡ λ.1(λ.3)
```

Shifting and substitution

When substituting into a λ term, indices must be adjusted:

```
λ y. x [ z/x] in context x,y,z
[2  ↦ 0] λ. 2 ≡ λ . [3 ↦ 1] 3 ≡ λ .1
```

Key point: context becomes longer when substituting inside an abstraction. Need to be careful to adjust free variables, not bound ones.

```
shift(d,c)(k) = k if k < c
                       k + d if k ≥ c
shift(d,c)(λ.t) = (λ.shift(d,c+1)(t))
shift(d,c)(t t')=
               (shift(d,c)(t)) (shift(d,c)(t'))
```

Example

```
shift(2,0)(λ.λ. 1 (0 2))
    λ.λ. 1 (0 4)

shift(2,0)(λ. 0 1 (λ. 0 1 2))
    λ. 0 3 (λ. 0 1 4)
```

Substitution

```
[ j ↦ s] k = s if k = j
                 k otherwise


[j ↦ s](λ .t ) = λ. [j+1 ↦ shift(1,0)s] t


[j ↦ s](t1 t2) = ([j ↦ s) t1) ([j ↦ s] t2)
```

Beta-reduction:

```
(λ t) v → shift(-1,0)([0 ↦ shift(1,0)(v)] t)
```

Examples

Assume context <a,b>

- Then, a ↦ 1, b ↦ 0

```
[a / b] b λ x. λ y. b
         [0 ↦ 1] 0 λ . λ . 2
         1 λ . λ . 3 ≡  a λ x. λ y. a

[(a (λ z. a)) / b] (b (λ x.b))
         [0 ↦ (1 (λ. 2))] (0 λ. 1)
         1 (λ . 2) (λ . (2 (λ . 3)))
         (a (λ z. a)) (λ x. (a (λ z. a)))
```

Examples

```
[a/b] (λ b. (b a))
      [0 ↦ 1] (λ . (0 2))
      (λ . (0 2))
      (λ b. (b a))
[a/b] (λ a. (b a))
      [0 ↦ 1] λ . (1 0)
      λ . (2 0)
     (λ a'. a a')
```