德布朗指数(de Bruijn index)

基础软件理论与实践公开课

张宏波

Review: Church numerals

- Notation:
  - do not curry, e.g. $\lambda(x, y).\, x$
  - do not omit parentheses when applying functions, e.g. $\lambda(f, x, y).\, f(x, y)$
- Chain of natural numbers

Review: Church numerals

idea
基础软件中心

- 0

$$\lambda(s, z).\, z$$

- succ

$$\lambda n.\, \lambda(s, z).\, s(n(s, z))$$

- 1 : `succ(0)`

$$\lambda(s, z).\, s(z)$$

- 2 : `succ(1)`

$$\lambda(s, z).\, s(s(z))$$

- 3 : `succ(2)`

$$\lambda(s, z).\, s(s(s(z)))$$

idea 基础软件中心

Review: Church numerals

Num	Peano	Lambda
0	Z	$\lambda(s, z).\, z$
$\downarrow$	$\downarrow$	$\downarrow$
1	$S(Z)$	$\lambda(s, z).\, s(z)$
$\downarrow$	$\downarrow$	$\downarrow$
2	$S(S(Z))$	$\lambda(s, z).\, s(s(z))$
$\downarrow$	$\downarrow$	$\downarrow$
$\ldots$	$\ldots$	$\ldots$

Review: Church numerals

In ReScript:

```
type rec nat = Z | S(nat)
let peano_zero = Z
let peano_one = S(Z)
let peano_two = S(S(Z))

type cnum<'a> = ('a => 'a, 'a) => 'a;
let church_zero = (s, z) => z
let church_one = (s, z) => s(z)
let church_two = (s, z) => s(s(z))

let peano_succ = (x) => S(x)
let church_succ = (n) => (s, z) => s(n(s, z))
```

- Notation:

$$\bar{n} = \lambda(s, z).\, s^n(z)$$

Review: Church numerals

Isomorphism in ReScript:

```
let church_to_peano = (n) => n(x => S(x), Z)
let rec peano_to_church = (n) => {
  switch n {
    | Z => church_zero
    | S(n) => church_succ(peano_to_church(n))
  }
}
```

Review: Predecessor

Num	Pair	Fst
0	$(0, 0)$	0
$\downarrow$	$\downarrow$	$\downarrow$
1	$(0, 1)$	0
$\downarrow$	$\downarrow$	$\downarrow$
2	$(1, 2)$	1
$\downarrow$	$\downarrow$	$\downarrow$
$\dots$	$\dots$	$\dots$
n	$(n-1, n)$	$n-1$

Review: Predecessor

$$
\begin{aligned}
\text{f} &= \lambda p.\ \ \text{pair}\ \ (\text{second}\ \ p)\ (\text{succ}\ \ (\text{second}\ \ p)) \\
\text{zero} &= (\lambda f.\, \lambda x.\ \ x) \\
\text{pc0} &= \text{pair}\ \ \text{zero}\ \ \text{zero} \\
\text{pred} &= \lambda n.\ \ \text{first}\ \ (n\ \ \text{f}\ \ \text{pc0})
\end{aligned}
$$

$$
\begin{aligned}
\text{pred three} &= \text{first}\ \ (\text{f}\ \ (\text{f}\ \ (\text{f}\ \ (\text{pair}\ \ \text{zero}\ \ \text{zero})))) \\
&= \text{first}\ \ (\text{f}\ \ (\text{f}\ \ (\text{pair}\ \ \text{zero}\ \ \text{one}))) \\
&= \text{first}\ \ (\text{f}\ \ (\text{pair}\ \ \text{one}\ \ \text{two})) \\
&= \text{first}\ \ (\text{pair}\ \ \text{two}\ \ \text{three}) \\
&= \text{two}
\end{aligned}
$$

idea
基础软件中心

Review: Predecessor

In ReScript:

```
let pred = (n) => {
  let init = (church_zero, church_zero)
  let iter = ((_, y)) => (y, church_succ(y))
  let (ans, _) = n(iter, init)
  ans
}

let church_decode = (n) => n((x) => x + 1, 0)

Js.Console.log(church_decode(church_two)) // 2
Js.Console.log(church_decode(pred(church_two))) // 1
```

References (for interested audience)

- TAPL: Sec 5-7

Review

- Evaluate closed term, call by value

```
let rec eval = (t: lambda) => {
  switch t {
  | Var(_) => assert false
  | Fn(_, _) => t
  | App(f, arg) => {
    let Fn(x, body) = eval(f)
    let va = eval(arg)
    eval(subst(x, va, body)) // substitution explained later
    }
  }
}
```

基础软件中心

Substitution without free variables

```
// v must be closed. a[v/x]
let rec subst = (x, v, a) => {
  switch a {
    | Var(y) => if x == y { v } else { a }
    | Fn(y, b) => if x == y { a } else { Fn(y, subst(x, v, b)) }
    | App(b, c) => App(subst(x, v, b), subst(x, v, c))
  }
}
```

- For example,

$$(\lambda y.\,(\lambda z.\,z + a)\;y)[\bar{1}/a] \to (\lambda y.\,(\lambda z.\,z + \bar{1})\;y)$$

Substitution with non-closed terms

The substitution of N for free occurrence of x in M, denoted by $M[N/x]$, is defined as follows:

$$
\begin{aligned}
x[N/x] &= N, & \\
y[N/x] &= y, & \text{if } x \neq y, \\
(MP)[N/x] &= (M[N/x])(P[N/x]), & \\
(\lambda x.\, M)[N/x] &= \lambda x.\, M, & \\
(\lambda y.\, M)[N/x] &= \lambda y.\, (M[N/x]), & \text{if } x \neq y \text{ and } y \notin FV(N),
\end{aligned}
$$

Question: what if the *binder* $y \in FV(N)$, for example

$$(\lambda y.\, xy)[\lambda z.\, yz/x] = ?$$

Rename the binder to unique names

- The terms $\lambda x.\, x$ and $\lambda y.\, y$ are essentially the same
- Roughly speaking, names do not matter

```
// the new name must be unique like JS new symbol
let rename = (t, old, new) => {
  let rec go = (t) => {
    switch t {
    | Var(x) => if x == old { Var(new) } else { t }
    | Fn(x, a) => if x == old { Fn(new, go(a)) } else {Fn(x, go(a))}
    | App(a, b) => App(go(a), go(b))
    }
  }
  go(t)
}
```

Alpha equivalence

Formally,

$$\lambda x.\, M =_\alpha \lambda y.\, (M\{y/x\})$$

where $M\{y/x\}$ is the result of renaming x as y in M, defined as:

$$\begin{aligned}
x\{y/x\} &= y, & \\
z\{y/x\} &= z, & \text{if } x \neq z, \\
(MN)\{y/x\} &= (M\{y/x\})(N\{y/x\}), & \\
(\lambda x.\, M)\{y/x\} &= \lambda y.\, (M\{y/x\}), & \\
(\lambda z.\, M)\{y/x\} &= \lambda z.\, (M\{y/x\}), & \text{if } x \neq z.
\end{aligned}$$

- The new name y must be *fresh*

Substitution

- Always rename

```
// t[u/x] where u might have free variables
let rec subst = (t, x, u) => {
  switch t {
  | Var(y) => if x == y { u } else { t }
  | Fn(y, b) => if x == y { t } else {
    let y' = fresh_name ()
    let b' = rename(b, y, y')
    Fn(y', subst(b', x, u))
  }
  | App(a, b) => App(subst(a, x, u), subst(b, x, u))
  }
}
```

- Free variables calcuation is not needed when we always do he renaming

Example

$$
\begin{aligned}
(\lambda x.\, yx)[\lambda z.\, xz/y] &=_\alpha (\lambda x'.\, yx')[\lambda z.\, xz/y] \\
&\to_\beta \lambda x'.\, (\lambda z.\, xz)x'
\end{aligned}
$$

- why substitution matters when the interpreter with environment is more efficient?

Example

- Capture-free Inlining

```
let x = ... in
let f = fun a -> x + a in
let g = fun x -> f(x) + x in ...
```

- wrong result

```
let x = ... in
let f = fun a -> x + a in
let g = fun x -> (let a = x in x + a) + x in ...
```

- what would be correct?

De Bruijn index

- Names don't matter. So we don't need them
- For example, $\lambda x.\, \lambda y.\, x(y(x))$ corresponds to $\lambda.\, \lambda.\, 1(0(1))$
- The number i stands for the variable bound by the ith binder λ
- Exercise: write down the de bruijn term for $Y = \lambda f.\, (\lambda x.\, f\ (x\ x))(\lambda x.\, f\ (x\ x))$

- Used in Caml light, MoscowML VM
- We already learnt the De bruijn index in our tiny languages.

idea
基础软件中心

Binders

- Let expression binds a function or expression to a variable

for example, in ReScript:

```
let f = a => a
let x = 2
f(x)
```

- Pattern matching introduces binders

for example, in ReScript:

```
switch p {
| (a, b) => a + b
}
```

Binders

- let-expression: tiny language 2

```
type rec expr =
  ...
  | Var (int)
  | Let (expr, expr)
```

for example, $\textsf{Let}(x, 1, \textsf{Add}(\textsf{Var}(x), \textsf{Cst}(1))$ becomes $\textsf{Let}(1, \textsf{Add}(\textsf{Var}(0), \textsf{Cst}(1))$

- Pattern matching

```
switch p {
  C(a, b) => a + b
}
```

```
switch Var(i) {
  C(_, _) => Var(0) + Var(1)
}
```

Substitution in De Bruijn notation

- The index for the *substituted* variable can change

$$(x \times (\lambda y.\, x + y)\ (z))[\bar{2}/x] = \bar{2} \times (\lambda y.\, \bar{2} + y)(z)$$

$$(0 \times (\lambda\ \ .\, 1 + 0)\ (3))[\bar{2}/0] = \bar{2} \times (\lambda\ \ .\, \bar{2} + 0)(3)$$

```
// t[u/i]: use u to replace Var(i) in term t
let rec subst = (t, i, u) => {
  switch t {
  ...
  | Fn(b) => Fn(subst(b, i+1, u))
  ...
  }
}
```

Substitution in De Bruijn notation

- Shift the term `u`

$$(x \times (\lambda y.\, x + y)\ (z))[w/x] = w \times (\lambda y.\, w + y)(z)$$

$$(0 \times (\lambda\ \ .1 + 0)\ (3))[2/0] = \textcolor{red}{2} \times (\lambda\ \ .\textcolor{red}{3} + 0)(3)$$

- Notice: w becomes $\mathsf{Var}(3)$ when substitution goes under a binder

```
// t[u/i]: use u to replace Var(i) in term t
let rec subst = (t, i, u) => {
  switch t {
  | Var(j) => if j == i { u } else { t }
  | Fn(b) => Fn(subst(b, i+1, shift(1, u)))
  | App(a, b) => App(subst(a, i, u), subst(b, i, u))
  }
}
```

Shift

- Shift should be only applied to *unbound variables*
- For example,

$$\uparrow^1 \mathsf{Var}(i) = \mathsf{Var}(i+1)$$

- How about

$$\uparrow^1 (\lambda.\, 0)(1) = ???$$

shift_aux(i, d, t) where `d` is the cutoff

- Formally,

$$
\begin{aligned}
\uparrow^i_d (j) &= j, && \text{if } j < d \\
\uparrow^i_d (j) &= i + j, && \text{if } j \geq d \\
\uparrow^i_d (\lambda.\, t) &= \lambda.\uparrow^i_{d+1} (t), \\
\uparrow^i_d (t_1 t_2) &= \uparrow^i_d (t_1) \uparrow^i_d (t_2)
\end{aligned}
$$

- shift(i, t) = shift_aux(i, 0, t)
- unbound variables shifted by `i`, bounded varaibles kept intact

idea
基础软件中心

Implementation

- Shift

```
// Var(j) becomes Var(i+j) if j >= d
let rec shift_aux = (i, d, u) => {
  switch u {
  | Var(j) => { if j >= d { Var(i+j) } else { u } }
  | Fn(b) => Fn(shift_aux(i, d+1, b))
  | App(a, b) =>
      App(shift_aux(i, d, a), shift_aux(i, d, b))
  }
}
let shift = (i, u) => shift_aux(i, 0, u)
```

基础软件中心

Implementation

```
// t[u/i]: use u to replace Var(i) in term t
let rec subst = (t, i, u) => {
  switch t {
  | Var(j) => if j == i { u } else { t }
  | Fn(b) => Fn(subst(b, i+1, shift(1, u)))
  | App(a, b) => App(subst(a, i, u), subst(b, i, u))
  }
}
```

- `shift` is a `nop` for closed term.
- Why we generalized `shift` for numbers other than 1 ?

基础软件中心

Interpreter

Don't forget shift the index by -1 after we drop a binder

For example,

$$(\lambda x.\, (\lambda y.\, x + y)(a)) \to_\beta (\lambda x.\, x + a)$$
$$(\lambda \;\; .\, (\lambda \;\; .\, 1 + 0)(3)) \to_\beta (\lambda \;\; .\, \color{red}{0} + 3)$$

Summary

How represent variables

- Symbolically: fresh names to avoid capture
- Symbolically with constraint: stamp
- More aggressive: single assignment for binders
- De Bruijn index: no need to rename but hard to manipulate
- Combinatory logic: no variables involved

Homework

- Complete the de Bruijn index based interpreter in natural semantics
- Apply the de Bruijn index for extended lambda calculus (+ Let)

```
type rec debru =
  | Var (int)
  | App (debru, debru)
  | Fun (debru)
  | Let (debru, debru)
```