Models of computation Week 11

More on λ-calculus

λ-calculus for computation

- facilities that it supports
  - using functions as parameters "first-class citizens"
  - conditions, iteration, recursion
  - types

Steve Vickers

More

- { adding constants $*$ $+$ 0 1 2 / pure λ-calculus }
- β-reduction is computation
- computed value = normal form (unique, by Church-Rosser)
- say $M = N$ if they have the same normal form (or none)

Higher order terms

- use functions as parameters

example (not higher order yet)

$Q \stackrel{def}{=} \lambda x.\ * \ x\ x$ — squares its parameter $x \longmapsto x^2$

$P_4 \stackrel{def}{=} \lambda x.\ Q(Qx)$ — squares 2 times $x \longmapsto x^4$

$P_8 \stackrel{def}{=} \lambda x.\ Q(Q(Qx))$ — squares 3 times $x \longmapsto x^8$

Digression

$Q \stackrel{def}{=} \lambda x.\ * \ x\ x$

$P_4 \stackrel{def}{=} \lambda x.\ Q(Qx)$

$P_8 \stackrel{def}{=} \lambda x.\ Q(Q(Qx))$

Q, P_4, P_8 etc give efficient way to compute powers x^n

e.g. 27 = binary 11011

16 +	8 +		2 +	1
x^{16}	x^8	x^4	x^2	x
x^{16}	x^8		x^2	x

$x^{27} = x^{16} * x^8 * x^2 * x$

$= P_{16}x * P_8 x * Qx * x$

3 multiplications here + 4 more to calculate $Qx, P_4 x$ etc

Very efficient for large n!

Can do this for any f, not just Q

Want $\quad T\ f\ x = f\,(f\,(f\ x))$

Define $\quad T \stackrel{def}{\equiv} \lambda f\ x.\ f\,(f\,(f\ x))$

$P_8 \stackrel{def}{\equiv} T\ (Q)$

parameter is a function

T is higher order

result is a function too

Conditionals

Add a constant zero? that has the effect of doing conditionals

$$\text{zero?}\ n\ x\ y = \begin{cases} x & \text{if } n = 0 \\ y & \text{if } n > 0 \end{cases}$$

(not defined for $n < 0$)

(n==0? x : y)

if n==0 value is x

otherwise value is y

Iteration

e.g. apply f n times

Can we define I so that

$$I\ n\ f\ x = f^n(x)\ ?$$

not λ-notation

recursion base case $\quad I\ 0\ f\ x = x$

if $n > 0 \quad I\ n\ f\ x = I\ (n-1)\ f\ (f\ x)$

for loops

```
double a=x;
for(int i=0;
    i<n;i++){
  a = f(a);
}
return a;
```

Java (almost)

not Java

```
/* requires n >= 0 */
public static double I(int n, function f, double x){
    if (n==0) {
        return x;
    } else {
        return I(n-1, f, f(x));
        // f^n(x) = f^(n-1)(f(x))
    }
}
```

(How to do it in Java?)

Function o

```
/* requires n >= 0 */
public static double I(int n, function f, double x){
    if (n==0) {
        return x;          o      o.f(x)
    } else {
        return I(n-1, f, f(x));
            // f^n(x) = f^(n-1)(f(x))
    }
}
```

(in red: "function f" circled; "f," and "f(x)" crossed out)

```
public interface Function {
    public double f(double x);
}
```

---

recursion base case $\underline{I}\ 0\ f\ x \sim x$

if $n>0$ $\underline{I}\ n\ f\ x \sim \underline{I}\ (n-1)\ f\ (f\ x)$

$$\underline{I}\ n\ f\ x = zero?\ n\ x\ (\underline{I}\ (pred\ n)\ f\ (f\ x))$$

pred $n = n-1$

predecessor

$$\underline{I} = \lambda n f x.\ zero?\ n\ x\ (\underline{I}\ (pred\ n)\ f\ (f\ x))$$

not a definition — recursion

can't write $\underline{I} \stackrel{def}{=} \ldots$

Fixpoints

$$Z \stackrel{def}{=} \lambda z x.\ x\ (z z x)$$

$$Y \stackrel{def}{=} Z Z$$

$$
\begin{aligned}
Y\ M &= Z\ Z\ M \\
&= (\lambda z x.\ x\ (z z x))\ Z\ M \\
&\to_\beta M\ (Z\ Z\ M) \\
&= M\ (Y\ M)
\end{aligned}
$$

Def^n a is a fixpoint of M if

$$M\ a = a$$

Have just shown whatever M is, YM is a fixpoint of it

Y is a fixpoint combinator — found by Turing

Problem find a fixpoint

Solution apply Y

Sometimes fixpoint can mean never get an answer - no normal form

Careless reduction

$YM \to_\beta M\,(YM)$

$\to_\beta M\,(M\,(YM))$

$\to_\beta M\,(M\,(M\,(YM)))$

$\to_\beta \ldots$

reduces forever

To get a normal form, must use our knowledge of what M is.

Some examples

$M \stackrel{def}{=} \lambda x.x$ – everything is a fixpoint

$Ma \stackrel{def}{=} (\lambda x.x)a = a$

$YM \to_\beta M\,(YM) \stackrel{def}{=} (\lambda x.x)(YM) \to_\beta YM$

no normal form

Some examples

$N \stackrel{def}{=} \lambda n.\ zero?\ n\ 1\ 0$

$N\ 0 = 1$

$N\ 1 = N2 = \ldots = 0$

No fix points

$YN = N\,(YN) = zero?\ (YN)\ 1\ 0$

Suppose YN has normal form a (an integer).

If $a = 0$ then $YN = 0$ $\quad zero?\ (YN)\ 1\ 0 = 1$

If $a > 0$ then $YN > 0$ $\quad zero?\ (YN)\ 1\ 0 = 0$

$\therefore YN$ has no normal form like that.

Solving a fixpoint equation:

$I = \lambda n f x.\ zero?\ n\ x\ (I\,(pred\,n)\ f\,(f\,x))$

λ-term M applied to I

We want fixpoint of M

$M \stackrel{def}{=} \lambda I.\ \lambda n f x.\ zero?\ n\ x\ (I\,(pred\,n)\ f\,(f\,x))$

Want $I = M\ I$

Solution is $I \stackrel{def}{=} Y\,M$

Example

Reduce $I\ 2\ g\ y$ — should be $g\,(g\,y)$

$\stackrel{def}{=}\ Y M\ 2\ g\ y$

$\to_\beta\ M\ (YM)\ 2\ g\ y$

$\stackrel{def}{=}\ (\lambda I.\lambda n f x.\ zero?\ n\ x\ (I\,(pred\,n)\ f\,(f\,x)))\ (YM)\ 2\ g\ y$ — (I, n, f, x)

$\to_\beta\ zero?\ 2\ y\ ((YM)\ (pred\ 2)\ g\ (g\,y))$

$\to\ (YM)\ (pred\ 2)\ g\ (g\,y)$ — $2 > 0$

$\to\ (YM)\ 1\ g\ (g\,y)$ — $pred\ 2 = 2-1 = 1$

Continuing

$YM\ 1\ g\ (g\,y) \to_\beta M\ (YM)\ 1\ g\ (g\,y)$

$\stackrel{def}{=}\ (\lambda I.\lambda n f x.\ zero?\ n\ x\ (I\,(pred\,n)\ f\,(f\,x)))\ (YM)\ 1\ g\ (g\,y)$ — (I, n, f, x)

$\to_\beta\ zero?\ 1\ (g\,y)\ ((YM)\ (pred\ 1)\ g\ (g\,(g\,y)))$

$\to\ (YM)\ (pred\ 1)\ g\ (g\,(g\,y))$

$\to\ (YM)\ 0\ g\ (g\,(g\,y))$

$\to_\beta\ M\ (YM)\ 0\ g\ (g\,(g\,y))$

$\stackrel{def}{=}\ (\lambda I.\lambda n f x.\ zero?\ n\ x\ (I\,(pred\,n)\ f\,(f\,x)))\ (YM)\ 0\ g\ (g\,(g\,y))$

$\to_\beta\ zero?\ 0\ (g\,(g\,y))\ ((YM)\ (pred\ 0)\ g\ (g\,(g\,(g\,y))))$

$\to\ g\,(g\,y)$

$I\ 2\ g\ y$ should be $g\,(g\,y)$

λ-calculus as a model of computation

Definition Suppose some calculus has an effective way of computing the result of functions it can define — λ-calculus uses reduction

Then it is **Turing complete** if it can define all the functions from natural numbers to natural numbers that are computable on Turing machines

Is the λ-calculus Turing complete?

Answer ①: Yes

– if you add constants for $zero?$, $pred$ (predecessor), $succ$ (successor)

$$pred\ n = \begin{cases} n-1 & \text{if } n>0 \\ 0 & \text{if } n=0 \end{cases}$$

$succ\ n = n+1$

and all natural numbers 0, 1, 2 . . .

Essentially enough to have:
- arithmetic – pred, succ
- program constructs – zero? – conditionals; Y – recursion

With all the constants

(obviously)

- can represent all natural numbers
- can build algorithms
  - zero? gives conditionals
  - Y gives recursion & iteration (with pred & zero?)
- succ and pred – very basic arithmetic
  with algorithms can build up +, *, – etc
  & all computable functions
  e.g. $+ = \lambda m n.\ I\ m\ \text{succ}\ n$
  add 1 m times to n

Is the λ-calculus Turing complete?

Answer ②: Yes – in pure λ-calculus (no constants)

Have to define natural numbers as λ-terms
n coded up as term $\lambda f.\ f^n$

e.g. $0 \stackrel{\text{def}}{=} \lambda f\,x.\,x$ — $0\,f$ = identity fn. $0\,f\,x = x$

$1 \stackrel{\text{def}}{=} \lambda f\,x.\,f\,x$ — $1\,f = f$, $1\,f\,x = f\,x$

$2 \stackrel{\text{def}}{=} \lambda f\,x.\,f\,(f\,x)$ — $2\,f\,x = f(f\,x)$

⋮

Arithmetic: succ

$n\,f$ is meant to be f^n (apply f n times)

$n\,f\,x$ .... $f^n(x)$

$\text{succ}\ n\,f\,x$ .... $f^{n+1}(x)$

i.e. $f\,(f^n(x)) = f\,(n\,f\,x)$

$\therefore \text{succ} \stackrel{\text{def}}{=} \lambda n f x.\ f\,(n\,f\,x)$

Arithmetic: zero?

If n is zero then $n\,f\,x = x$ (regardless of f)

Consider $f = \lambda u.y$

$(\lambda u.y)\,x = y = (\lambda u.y)^2\,x = (\lambda u.y)^3\,x = \cdots$

$\therefore n\,(\lambda u.y)\,x = \begin{cases} x & \text{if } n \text{ is zero} \\ y & \text{if } n > 0 \end{cases}$

$(\lambda u.y)^n(x)$

∴ define

$\text{zero?} \stackrel{\text{def}}{=} \lambda n\,x\,y.\ n\,(\lambda u.y)\,x$

Arithmetic: pred

$\text{pred} \stackrel{\text{def}}{=} \lambda n f x.\ n\ (\lambda g h.\ h(g f))\ (\lambda u.x)(\lambda u.u)$

$= \lambda n f x.\ n\ T\ (\lambda u.x)\ (\lambda u.u)$

where $T = \lambda g h.\ h\ (g f)$

$0\ T\ (\lambda u.x) \to_\beta \lambda u.x$

$1\ T\ (\lambda u.x) \to_\beta T\ (\lambda u.x)$

$\to_\beta \lambda h.\ h((\lambda u.x) f) \to_\beta \lambda h.\ h x$

different, but
$(\lambda u.x)(\lambda u.u) \to_\beta x$
$(\lambda h.hx)(\lambda u.u)$
$\to_\beta (\lambda u.u)x \to_\beta x$

More ...

where $T = \lambda g h.\ h\ (g f)$
$0\ T\ (\lambda u.x) \to_\beta \lambda u.x$
$1\ T\ (\lambda u.x) \to_\beta T\ (\lambda u.x)$
$\to_\beta \lambda h.\ h((\lambda u.x) f) \to_\beta \lambda h.\ h x$

$2\ T\ (\lambda u.x) \to_\beta T\ (T\ (\lambda u.x))$

$\to_\beta \cdots\ T\ (\lambda h.hx)$

$\to_\beta \lambda h.\ h((\lambda h.hx) f) \to_\beta \lambda h.\ h(f x)$

$n\ T\ (\lambda u.x) \to_\beta \cdots\ \lambda h.\ h(f^{n-1} x)$

informally – come back to this in exercises

$\text{pred} = \lambda n f x.\ n\ T\ (\lambda u.x)(\lambda u.u)$

$\therefore \text{pred}\ n \sim \lambda f x.\ (\lambda h.\ h(f^{n-1} x))\ (\lambda u.u)$

$\sim \lambda f x.\ (\lambda u.u)(f^{n-1} x) \sim \lambda f x.\ f^{n-1} x$

Types

Pure λ-calculus "everything is a function & it can be applied to anything"

- not always sensible e.g. log (sin)

Typing Keep track of –

- what are functions?
- what types are their parameters & results?

Function types

If σ, τ are types:

$\sigma \to \tau$ is type for functions with

- parameter of type σ
- result of type τ

e.g.

With constants for arithmetic (natural numbers)

$0, 1, 2, \ldots$ of type nat

succ, pred of type nat $\to$ nat (function: parameter of type nat, result of type nat)

zero? – depends on type of alternatives

If x, y of type σ, so is zero? $n\ x\ y$

zero? of type nat $\to \sigma \to \sigma \to \sigma$ (first parameter n, 2nd param x, 3rd param y, result)

Bracketing?

zero? of type nat $\to \sigma \to \sigma \to \sigma$ (first parameter n, 2nd param x, 3rd param y, result)

$\sigma \to \tau \to \upsilon$ means $\sigma \to (\tau \to \upsilon)$

parameter – type σ

result – type $\tau \to \upsilon$

Another function: parameter τ, result υ

– Just how we dealt with multiple parameters

In general for types without brackets

$\sigma_1 \to \sigma_2 \to \ldots \to \sigma_n \to \tau$ (parameter types: $\sigma_1 \ldots \sigma_n$; result type: τ)

or: $\sigma_1 \to (\sigma_2 \to (\ldots \to (\sigma_n \to \tau)\ldots))$

Higher order types

parameter is a function

e.g. $\lambda f x.\ f\ (f\ (f\ x))$

If x is type σ, f must be $\sigma \to \sigma$

$\therefore \lambda f x.\ f\ (f\ (f\ x))$ has type

$(\sigma \to \sigma) \to \sigma \to \sigma$ (must have brackets; type of f, type of x, type of result)

or $(\sigma \to \sigma) \to (\sigma \to \sigma)$

Context free grammar for types τ

$$\tau ::= c \mid \tau \to \tau$$

c: constants, e.g. nat, int, real

$\tau \to \tau$: function types

Ambiguous! Leaves out brackets

"Well typed"

A term is "well typed" if can say what its type is.

Use annotations to say what types variables are

$x : \sigma$ – "x has type σ"

Then the variable is well typed.

Abstractions

Abstractions always get a function type

If M is well typed, type τ and all uses of x in M are $x:\sigma$

Then $\lambda x:\sigma.\ M$ is well typed, type $\sigma \to \tau$

Applications

If M, N both well typed

M – type $\sigma \to \tau$ ← formal

N – type σ ← actual

formal / actual parameter types must match

then MN is well typed, type τ

Is a term well typed?

Use type variables to stand for types while you work out constraints.

e.g. $\lambda f x.\ f x$ Say $x:A$, $f:B$

For $f x$ to be well typed:

B a function type $A \to C$, some C

$x : A$ $f : A \to C$ $f x : C$

$\lambda x.\ f x : A \to C$

$\lambda f x.\ f x : (A \to C) \to A \to C$

In full:

$\lambda f{:}A\to C.\ \lambda x{:}A.\ f{:}A\to C\ \ x{:}A$

Refining the types

$\lambda f : A \to C.\ \lambda x : A.\ f x$

Suppose term was in a bigger term

$(\lambda f x.\ f x)(\lambda u.u)\ 3$ Say $u : D$

$(A \to C) \to A \to C$ $D \to D$ int

for application, $A \to C = D \to D$ $\therefore A = C = D$

$(\lambda f x.\ f x)(\lambda u.u)\ 3$

$(D \to D) \to (D \to D)$ $D \to D$ int

$D \to D$

for this application must have $D = \text{int}$

$(\lambda f : \text{int} \to \text{int}.\ \lambda x : \text{int}.\ f x)(\lambda u : \text{int}.\ u)\ 3$

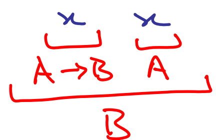
Some terms aren't well typed!

e.g $x\,x$

if $x : A$

for application $x\,x$,

$A = A \to B$, some B



But can't find types A, B with $A = A \to B$

e.g. $Y = (\lambda z x.\ x\,(z z x))(\lambda z x.\ x\,(z z x))$

- similar reason

Simply typed λ-calculus

Same as λ-calculus, except

- every variable has a type
- only allow terms that are well typed

Very well behaved, because –

Theorem

Every well typed term has a normal form

PCF

"programming computable functions" – Scott, Plotkin

Turing completeness with well typed λ-terms

- nat as a type (natural numbers)
- constants for all natural numbers
- succ, pred : nat → nat
- $zero?_\sigma$: nat $\to \sigma \to \sigma \to \sigma$ for each σ

BUT Y not well typed ∴ need special symbols

- $Y_\sigma : (\sigma \to \sigma) \to \sigma$ for each type σ

+ reduction rule $Y_\sigma f \to f\,(Y_\sigma f)$

Theorem PCF is Turing complete