CS442 Notes

1 Functional Programming

- to begin: Functional Programming in general, not a particular language
- need a model
    - simple
    - powerful enough to be useful

Church-Turing Thesis: Any algorithm can be simulated on a Turing machine.

Alan Turing

- Turing machine
- not an inspiring model for programming

Alonzo Church

- λ-calculus
- equivalent to Turing Machines
- the basis for all of functional programming

1.1 Untyped Lambda-Calculus

Calculus

- a system or method of calculation
- syntax + rules for manipulating syntax

Lambda-Calculus (1934) - "a calculus of functions"

Syntax: Abstract Syntax

$$\begin{aligned} & \quad\ \ \textit{variable}, \textit{abstraction}, \textit{application} \\ <\textit{expr}> &::= <\textit{var}> \mid <\textit{abs}> \mid <\textit{app}> \\ <\textit{var}> &::= a|b|c \\ & \quad\ \ \text{variable and body} \\ <\textit{abs}> &::= \lambda <\textit{var}> . <\textit{expr}> \\ & \quad\ \ \text{rator and rand} \\ <\textit{app}> &::= <\textit{expr}><\textit{expr}> \end{aligned}$$

Use parenthesis to disambiguate parses

Conventions:

1. Abstractions extend as far to the right as possible

   e.g. $\lambda x.y\ z = \lambda x.(y\ z) \neq (\lambda x.y)\ z$

2. Applications associate left-to-right

   e.g. $x\ y\ z = (x\ y)\ z \neq x\ (y\ z)$

Interpretation

- Var: Self explanatory
- Abs: $\lambda x.E =$ "function" taking argument x and returning expr E
- App: $M\ N$ - result of applying "function" M to argument N

Consider:

$\lambda x.x$ - Here the first x is a binding occurrence and second x is a bound occurrence.

$\lambda x.x\ y\ x$ - The y is a free occurrence (fully parenthesized function is $\lambda x.((x\ y)\ x)$)

$\lambda x.x\ (\lambda x.x)\ x$ - Same variable name might be bound in difference places as long as the scope is not clashing

$x\ \lambda x.x$ - The x in the front is free while the other one is bound

Informally:

A variable is bound if it has a bound occurrence.

A variable is free if it has a free occurrence

The same variable can be used in both bound and free occurrences

Formally:

Definition: Let E be an expression. The bound variables of E, $B \cup [E]$ are given by

$B \cup [x] = \emptyset$

$B \cup [\lambda x.E] = B \cup [E] \cup \{x\}$

$B \cup [M\ N] = B \cup [M] \cup B \cup [N]$

x is bound in E if $x \in B \cup [E]$

The free variables of E, $F \cup [E]$ are given by

$F \cup [x] = \{x\}$

$F \cup [\lambda x.E] = F \cup [E] \setminus \{x\}$

$F \cup [M\ N] = F \cup [M] \cup F \cup [N]$

x is free in E if $x \in F \cup [E]$

A variable can be both bound and free: $F \cup [x\ \lambda x.x] = B \cup [x\ \lambda x.x] = \{x\}$

But each occurrence is either bound or free, not both

Two occurrences of a variable x "mean the same thing" if

1. they are both free occurrences
2. OR they have the same binding occurrence

$\lambda x.\lambda y.y\ x$ and $\lambda a.\lambda b.b\ a$ these should mean the same thing

$\lambda x.y\ x$ and $\lambda x.w\ x$ should not mean the same thing because y and w do not have to be the same free variable

You can change the names of bound variables, but not free ones.

Formally:

Definition (α-conversion): For all x, y, M, $\lambda x.M =_\alpha \lambda y.M[y/x]$ if $y \notin F \cup [M]$

$M[N/x]$ = "substitute N for x in M"

More generally, if $C[\lambda x.M]$ denotes an expr in which $\lambda x.M$ occurs as a subexpression, then $C[\lambda x.M] =_\alpha C[\lambda y.M[y/x]]$ if $y \notin F \cup [M]$.

e.g.

$\lambda x.x =_\alpha \lambda y.y$

$x\ \lambda x.x =_\alpha x\ \lambda a.a$

$\lambda a.b\ a =_\alpha \lambda c.b\ c \neq_\alpha \lambda b.b\ b \neq_\alpha \lambda a.d\ a$

Computation:

Expr. $(\lambda x.M)\ N$ is called a (β-)redex (reductible expression)

We expect: $(\lambda x.M)\ N \Rightarrow$ evaluate M, with N substituted for x i.e. $M[N/x]$

Definition (β-reduction): For all M, N, x, $(\lambda x.M)\ N \rightarrow_\beta M[N/x]$

More generally - for all contexts C, $C[(\lambda x.M)\ N] \to_\beta C[M[N/x]]$

$\to_\beta$ is a binary relation on terms

$A \to_\beta B$: A beta reduces to B in one step

$A \to_\beta^n B$: A beta reduces to B in n steps

$A \to_\beta^* B$: A beta reduces to B in 0 or more steps

$A \to_\beta^+ B$: A beta reduces to B in 1 or more steps

Evaluating $M[N/x]$

Want: Substitute N for all free occurrences of x in M

Definition (substitution, naive(wrong)):

$x[E/x] = E$

$y[E/x] = y$, $y \neq x$

$(M\ N)[E/x] = M[E/x]\ N[E/x]$

$(\lambda x.P)[E/x] = \lambda x.P$

$(\lambda y.P)[E/x] = \lambda y.P[E/x]$, $y \neq x$

E.g.

$$\begin{aligned}(\lambda x.x\ y)\ z &\to_\beta (x\ y)[z/x]\\ &= x[z/x]\ y[z/x]\\ &= z\ y\end{aligned}$$

$$(\lambda x.x)\ a \to_\beta x[a/x] = a$$

$$\begin{aligned}(\lambda x.\lambda y.x)\ a\ b &\to_\beta (\lambda y.x)[a/x]\ b\\ &= (\lambda y.x[a/x])\ b\\ &= (\lambda y.a)\ b\\ &\to_\beta a[b/y]\\ &= a\end{aligned}$$

$$\begin{aligned}(\lambda x.\lambda y.y)\ a\ b &\to_\beta (\lambda y.y)[a/x]\ b\\ &= (\lambda y.y[a/x])\ b\\ &= (\lambda y.y)\ b\\ &\to_\beta y[b/y]\\ &= b\end{aligned}$$

Consider:

$$\begin{aligned}(\lambda x.\lambda y.x)\ y\ w &\to_\beta (\lambda y.x)[y/x]\ w\\ &= (\lambda y.x[y/x])\ w\\ &= (\lambda y.y)\ w\\ &\to_\beta y[w/y]\\ &= w\end{aligned}$$

We can see from this that the last rule of $(\lambda y.P)[E/x] = \lambda y.P[E/x]$, $y \neq x$ must be wrong.

What happened? Free variable y became bound after substitution

As a result, the binding occurrence of x changed

$\lambda x.\lambda y.x \leftarrow$ the inner x is bound to the λx on the outside.

This is called **Dynamic binding** - meaning of variables uncertain until runtime

We want **static binding** - meanings of variables fixed before runtime

Definition (substitution, fixed):

$x[E/x] = E$

$y[E/x] = y$, $y \neq x$

$(M\ N)[E/x] = M[E/x]\ N[E/x]$

$(\lambda x.P)[E/x] = \lambda x.P$

$(\lambda y.P)[E/x] = \lambda y.P[E/x]$, $y \notin F \cup [E]$

$(\lambda y.P)[E/x] = \lambda z.(P[z/y][E/x])$, $y \in F \cup [E]$, z a "fresh" variable

Now

$$\begin{aligned}(\lambda x.\lambda y.x)\ y\ w &\to_\beta (\lambda y.x)[y/x]\ w\\ &= (\lambda z.x[z/y][y/x])\ w\\ &= (\lambda z.x[y/x])\ w\\ &= (\lambda z.y)\ w\\ &\to_\beta y[w/z]\\ &= y\end{aligned}$$

Computation: β-reduction until you reach a **Normal Form**.

Definition (β-Normal Form): An expr is in β Normal Form if it has no β-redices.

Definition (Weak Normal Form): An expr is in Weak Normal Form if the only β-redices are inside abstractions. e.g. $\lambda z.(\lambda x.x)\ y$.

Usually, "Normal Form" will mean β-Normal Form.

Consider:

$$\begin{aligned}(\lambda x.x)((\lambda y.y)\ z) &\to_\beta x[((\lambda y.y)\ z)/x] \\ &= (\lambda y.y)\ z \\ &\to_\beta y[z/y] \\ &= z\end{aligned}$$

Or

$$\begin{aligned}(\lambda x.x)((\lambda y.y)\ z) &\to_\beta (\lambda x.x)(y[z/y]) \\ &= (\lambda x.x)\ z \\ &\to_\beta x[z/x] \\ &= z\end{aligned}$$

- two difference reduction sequences. Does it matter which we take?

Theorem (Church-Rosser): Let E_1, E_2, E_3 be expressions such that $E_1 \to_\beta^* E_2$ and $E_1 \to_\beta^* E_3$. Then there is an expression E_4 such that (up to α-equivalence) $E_2 \to_\beta^* E_4$ and $E_3 \to_\beta^* E_4$.

Immediate Consequence - An expr E can have at most one β-Normal Form (modulo α-equivalence)

Do all expressions have a β-NF? No!

Consider:

$$\begin{aligned}(\lambda x.x\ x)(\lambda x.x\ x) &\to_\beta (x\ x)[(\lambda x.x\ x)/x] \\ &= (\lambda x.x\ x)(\lambda x.x\ x)\end{aligned}$$

It does not have a β-NF because it always reduces to itself.

Which exprs have a β-NF? - undecidable - equivalent to Halting Problem

Consider:

$$(\lambda x.y)((\lambda x.x\ x)(\lambda x.x\ x)) \to_\beta (\lambda x.y)((\lambda x.x\ x)(\lambda x.x\ x)) \to_\beta \cdots$$

Or, the alternate way of reducing this

$$\begin{aligned}(\lambda x.y)((\lambda x.x\ x)(\lambda x.x\ x)) &\to_\beta y[(\lambda x.x\ x)(\lambda x.x\ x)/x] \\ &= y\end{aligned}$$

$\therefore$ Order does matter when ∞-reductions are possible.

Reduction Strategies - "plans" for choosing a redex to reduce.

Applicative Order Reductions (AOR): Choose the leftmost, innermost redex.

- Innermost = not containing any other redex

- This is called "eager evaluation"

Normal Order Reduction (NOR): Choose the leftmost,outermost

- outermost = not contained in any other redex

- "lazy evaluation"

To the example earlier, the one the results in infinite reduction is AOR and the one that go the β-NF form is NOR.

e.g.

$$(\lambda x.x)((\lambda y.y)\ z) \xrightarrow{\text{AOR}}_\beta (\lambda x.x)\ z \xrightarrow{\text{AOR}}_\beta z$$
$$(\lambda x.x)((\lambda y.y)\ z) \xrightarrow{\text{NOR}}_\beta (\lambda y.y)\ z \xrightarrow{\text{NOR}}_\beta z$$

Theorem (Standardization): If an expr as a β-NF, then NOR is guaranteed to reach it.

But most proramming languages use applicative order, including Scheme.

η-reduction: Consider $(\lambda x.y\ x)\ z \rightarrow_\beta (y\ x)[z/x] = y\ z$

So $\lambda x.y\ x$ behaves exactly like z

Definition (η-reduction): $\lambda x.M\ x \rightarrow_\eta M$ if $x \notin F \cup [M]$

More generally, $C[\lambda x.M\ x] \rightarrow_\eta C[M]$ if $x \notin F \cup [M]$

So we have $\lambda x.y\ x \rightarrow_\eta y$

1.2 Programming in the λ-Calculus

Shorthand for convenience: $[[\cdot]]$: (real-world programming language) $\rightarrow$ λ-calculus

For a real-world expr E, $[[E]]$ is our representation of E in the λ-calculus

e.g. $[[id]] = \lambda x.x$

Note: $[[\cdot]]$ is just shorthand. An expression containing $[[\cdot]]$ is not λ-calculus until all $[[\cdot]]$s have been replaced with what they represent.

Booleans: Let [[true]] $= \lambda x.\lambda y.x$ and [[false]] $= \lambda x.\lambda y.y$.

if ¡bool¿ then ¡true-part¿ else ¡false-part¿

if(b, t, f) = if b then t else f

[[if]] $= \lambda b.\lambda t.\lambda f.b\ t\ f$

Note that $\lambda b.\lambda t.\lambda f.b\ t\ f \rightarrow_\eta^2 \lambda b.b$

Alternatively, [[if B then T else F]] $= [[B]][[T]][[F]]$

Does it work?

$$\begin{aligned}
\text{[[if true then P else Q]]} &= [[true]][[P]][[Q]] \\
&= (\lambda x.\lambda y.x)[[P]][[Q]] \\
&\rightarrow_\beta [[P]]
\end{aligned}$$

$$\begin{aligned}
\text{[[if false then P else Q]]} &= [[false]][[P]][[Q]] \\
&= (\lambda x.\lambda y.y)[[P]][[Q]] \\
&\rightarrow_\beta [[Q]]
\end{aligned}$$

Now, for the definition of **Not**:

$$\begin{aligned}
[[not]] &= \lambda b.\text{[[if b then false else true]]} \\
&= \lambda b.b[[false]][[true]] \\
&= \lambda b.b(\lambda x.\lambda y.y)(\lambda x.\lambda y.x)
\end{aligned}$$

We can test it:

$$\begin{aligned}
\text{[[not true]]} &= (\lambda b.b[[false]][[true]])\ [[true]] \\
&\rightarrow_\beta [[true]][[false]][[true]] \\
&= (\lambda x.\lambda y.x)[[false]][[true]] \\
&\rightarrow_\beta^2 [[false]]
\end{aligned}$$

For the definition of **And**:

$$\begin{aligned}[[and]] &= \lambda p.\lambda q.[[\text{if p then q else false}]]\\ &= \lambda p.\lambda q.p\ q\ [[false]]\end{aligned}$$

For the definition of **Or**:

$$\begin{aligned}[[and]] &= \lambda p.\lambda q.[[\text{if p then true else q}]]\\ &= \lambda p.\lambda q.p\ [[true]]\ q\end{aligned}$$

Storage: Use lists. Scheme - lists based on pairs.

(cons a b) creates the pair

(cons a (cons b c)) creates the list

∴ nested pairs create lists

Need to implement:

- cons
- nil - empty list
- null? - is the list empty?
- car - first component of the pair
- cdr - 2nd component of the pair

Modelling pairs: pair - function that takes a **selector** as a parameter

If the selector is true - return the first component

If the selector is true - return the second component

i.e. $[[pair]] = \lambda s.[[\text{if s then h else t}]]$ where s is the selector, h is the head, t is the tail.

So $[[pair]] = \lambda s.s\ h\ t$

Then $[[cons]] = \lambda h.\lambda t.\lambda s.s\ h\ t$

e.g.

$$\begin{aligned}[[\text{cons a (cons b nil)}]] &= [[cons]]\ a\ ([[cons]]\ b\ [[nil]])\\ &= (\lambda h.\lambda t.\lambda s.s\ h\ t)\ a\ ([[cons]]\ b\ [[nil]])\\ &\to_\beta^2 \lambda s.s\ a\ ([[cons]]\ b\ [[nil]])\\ &= \lambda s.s\ a((\lambda h.\lambda t.\lambda s.s\ h\ t)\ b\ [[nil]])\\ &\to_\beta^2 \lambda s.s\ a\ (\lambda s.s\ b\ [[nil]])\end{aligned}$$

car: return the head $\Rightarrow$ pass the selector "true"

$[[car]] = \lambda l.l\ [[true]] = \lambda l.l\ (\lambda x.\lambda y.x)$

Similarly, **cdr:** return the tail $\Rightarrow$ pass the selector "false"

$[[cdr]] = \lambda l.l\ [[false]] = \lambda l.l\ (\lambda x.\lambda y.y)$

null?

- must return false when given a pair $(\lambda s.s\ h\ t)$

- pass a selector that consumes h and t, returns false

i.e. $[[null?]] = \lambda l.l(\lambda a.\lambda b.[[false]]) = \lambda l.l(\lambda a.\lambda b.\lambda x.\lambda y.y)$

$$\begin{aligned} [[null?]](\lambda s.s\ h\ t) &= (\lambda l.l(\lambda a.\lambda b.\lambda x.\lambda y.y))(\lambda s.s\ h\ t) \\ &\rightarrow_\beta (\lambda s.s\ h\ t)(\lambda a.\lambda b.\lambda x.\lambda y.y) \\ &\rightarrow_\beta (\lambda a.\lambda b.\lambda x.\lambda y.y)\ h\ t \\ &\rightarrow_\beta^2 \lambda x.\lambda y.y \\ &= [[false]] \end{aligned}$$

nil: something to make null? return true.

$[[nil]] = \lambda s.[[true]] = \lambda s.\lambda x.\lambda y.x$

$$\begin{aligned} [[null?]][[nil]] &= (\lambda l.l(\lambda a.\lambda b.\lambda x.\lambda y.y))(\lambda s.\lambda x.\lambda y.x) \\ &\rightarrow_\beta (\lambda s.\lambda x.\lambda y.x)(\lambda a.\lambda b.\lambda x.\lambda y.y) \\ &\rightarrow_\beta \lambda x.\lambda y.x \\ &= [[true]] \end{aligned}$$

Numbers

- Consider only non-negative integers

- Easy - encode n as a list of length n

  $[[0]] = [[nil]]$
  $[[1]] = \lambda s.s\ ?\ [[nil]]$ where ? is just whatever
  $[[2]] = \lambda s.s\ ?\ (\lambda s.s\ ?\ [[nil]])$

Primitives: $[[pred]], [[succ]], [[isZero?]]$

$[[isZero?]] = [[null?]]$

$[[pred]] = [[cdr]]$

$[[succ]] = \lambda n.\lambda s.s\ ?\ n (= \lambda n.[[cons]]\ ?\ n)$

Exercise: $[[pred\ (succ\ m)]] =_\beta m$

Clever Solution: Church Numberals

Represent n as the act of applying a function f n times to an argument x.

$[[0]] = \lambda f.\lambda x.x$

$[[1]] = \lambda f.\lambda x.f\ x$

$[[2]] = \lambda f.\lambda x.f\ (f\ x)$

$[[3]] = \lambda f.\lambda x.f\ (f\ (f\ x))$

$[[n]]\ f\ x = f^n(x)$

addition: $m + n$ - apply f n times to x, then apply f m times to the result

$[[+]] = \lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x)$

$$
\begin{aligned}
[[+\ 2\ 3]] &= (\lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x))(\lambda f.\lambda x.f\ (f\ x))(\lambda f.\lambda x.f(f(f\ x))) \\
&\rightarrow^2_\beta \lambda f.\lambda x(\lambda f.\lambda x.f(f\ x))\ f\ ((\lambda f.\lambda x.f(f(f\ x)))\ f\ x) \\
&\rightarrow_\beta \lambda f.\lambda x.(\lambda x.f(f\ x))(\lambda f.\lambda x.f(f(f\ x)))\ f\ x) \\
&\rightarrow_\beta \lambda f.\lambda x.f(f((\lambda f.\lambda x.f(f(f\ x)))\ f\ x)) \\
&\rightarrow_\beta \lambda f.\lambda x.f(f((\lambda x.f(f(f(\ x)))\ x) \\
&\rightarrow_\beta \lambda f.\lambda x.f(f(f(f(f\ x)))) \\
&= [[5]]
\end{aligned}
$$

Special case: $[[succ]] = \lambda n.\lambda f.\lambda x.n\ f\ (f\ x)$ (or $f(n\ f\ x)$)

Subtraction: Harder - find a function to apply n times to produce $[[n-1]]$

Consider: Start with $[[cons\ 0\ 0]]$

apply the function $f : [[cons\ a\ b]] \rightarrow [[cons\ a+1\ a]]$

n times: $[[cons\ n\ n-1]]$ - then take the cdr.

$[[pred]] = \lambda n.[[cdr]]\ (n\ (\lambda p.[[cons]]([[succ]][[car\ p]])\ [[car\ p]])([[cons]][[0]][[0]]))$

$[[-]] = \lambda m.\lambda n.n\ [[pred]]\ m$

Multiplication: Apply the n-fold repetition of f, m times

$[[*]] = \lambda m.\lambda n.\lambda f.\lambda x.m\ (n\ f)\ x =_\eta \lambda m.\lambda n.\lambda f.m\ (n\ f)$

Since $n\ f = f^n$, then $m\ (n\ f) = (f^n)^m$

Notice that if you change m and n to f and g, and f to x, you get $f(g(x))$ which is just function composition.

Exponentiation: m^n - the n-fold repetition of m itself

$[[\hat{}]] = \lambda m.\lambda n.\lambda f.\lambda x.n\ m\ f\ x =_\eta \lambda m.\lambda n.n\ m$

Recursion: find the length of a list

$$\begin{aligned}[[len]] &= \lambda l.[[\text{if (null? } l\text{) 0 (succ (len (cdr l)))}]] \\ &= \lambda l.([[\text{null?}]]\ l)\ [[0]]\ ([[\text{succ}]]\ ([[\text{len}]]\ ([[\text{cdr}]]\ l)))\end{aligned}$$

WRONG! [[len]] defined in terms of itself.

How do we get a closed-form representation of len? Solve the equation.

Aside: fixed points. A **fixed point** of a function f is a value x such that $x = f(x)$

e.g. $f(x) = x^2 - 6$ has a fixed point of 3 since $f(3) = 3$

$$\begin{aligned}[[len]] &= \lambda l.([[null?]]\ l)[[0]]([[succ]]([[len]]([[cdr]]\ l))) \\ &\leftarrow_\beta (\lambda f.\lambda l.([[null?]]\ l)[[0]]([[succ]](f\ ([[cdr]]\ l))))\ [[len]] \\ &=_\beta F([[len]]) \text{ where F} = (\lambda f.\lambda l.([[null?]]\ l)[[0]]([[succ]](f\ ([[cdr]]\ l))))\end{aligned}$$

[[len]] satisfies $[[len]] = F([[len]])$

$\therefore$ Need a fixed point of F

Consider:

$$\begin{aligned}X &= (\lambda x.f(x\ x))(\lambda x.f(x\ x)) \\ &\rightarrow_\beta f((\lambda x.f(x\ x))(\lambda x.f(x\ x))) \\ &= f(X)\end{aligned}$$

$\therefore$ X is a fixed point of f

Now parameterize by f, get

$$Y = \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$$

This is **Curry's Paradoxical Combinator** (or Y combinator)

- returns the fixed point of any function

$$\begin{aligned} Y\ g &= (\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x)))\ g \\ &\to_\beta (\lambda x.g(x\ x))(\lambda x.g(x\ x)) \\ &\to_\beta g((\lambda x.g(x\ x))(\lambda x.g(x\ x))) \\ &\to_\beta g(Y\ g) \end{aligned}$$

$\therefore$ for any g**,** $Y\ g =_\beta g(Y\ g)$. i.e. Y g is a fixed point of g.

Any combinator (closed expression) C such that for all g, $C\ g =_\beta g(C\ g)$ is called a **fixed-point combinator**.

$\therefore$ $[[len]] = Y\ F = Y(\lambda f.\lambda l.([[null?]]\ l)\ [[0]]\ ([[succ]]\ (f\ ([[cdr]]\ l)))$

e.g.

$[[len]]$ $([[cons]]\ a\ ([[cons]]\ b\ [[nil]])$
$=_\beta [[len]]\ (\lambda s.s\ a\ (\lambda s.s\ b\ [[nil]]))$
$= Y; F(\lambda s.s\ a\ (\lambda s.s\ b\ [[nil]]))$
$= (\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))\ F\ (\lambda s.s\ a\ (\lambda s.s\ b\ [[nil]]))$ done through NOR reduction
$\rightarrow_\beta (\lambda x.F(x\ x))(\lambda x.F(x\ x))(\lambda s.s\ a\ (\lambda s.s\ b\ [[nil]]))$
$\rightarrow_\beta F((\lambda x.F(x\ x))(\lambda x.F(x\ x)))(\lambda s.s\ a\ (\lambda s.s\ b\ [[nil]]))$
$= (\lambda f.\lambda l.([[null?]]\ l)\ [[0]]\ ([[succ]]\ (f\ ([[cdr]]\ l)))((\lambda x.F(x\ x))(\lambda x.F(x\ x)))(\lambda s.s\ a\ (\lambda s.s\ b\ [[nil]]))$
$\rightarrow_\beta (\lambda l.([[null?]]\ l)\ [[0]]\ ([[succ]]\ (((\lambda x.F(x\ x))(\lambda x.F(x\ x)))\ ([[cdr]]\ l))))(\lambda s.s\ a\ (\lambda s.s\ b\ [[nil]]))$
$\rightarrow_\beta ([[null?]]\ (\lambda s.s\ a\ (\lambda s.s\ b\ [[nil]])))\ [[0]]\ ([[succ]]\ (((\lambda x.F(x\ x))(\lambda x.F(x\ x)))\ ([[cdr]]\ (\lambda s.s\ a\ (\lambda s.s\ b\ [[nil]])))))$
$\rightarrow_\beta^* [[false]]\ [[0]]\ [[succ]](((\lambda x.F(x\ x))(\lambda x.F(x\ x)))\ ([[cdr]]\ (\lambda s.s\ a\ (\lambda s.s\ b\ [[nil]]))))$
$\rightarrow_\beta^2 [[succ]]\ (((\lambda x.F(x\ x))(\lambda x.F(x\ x)))\ ([[cdr]]\ (\lambda s.s\ a\ (\lambda s.s\ b\ [[nil]]))))$
$\rightarrow_\beta^* [[succ]](((\lambda x.F(x\ x))(\lambda x.F(x\ x)))\ (\lambda s.s\ b\ [[nil]]))$ Not NOR-done for brevity
We can see a pattern here
$\rightarrow_\beta^* [[succ]]([[succ]]\ ((\lambda x.F(x\ x))(\lambda x.F(x\ x)))([[cdr]]\ (\lambda s.s\ b\ [[nil]]))))$
$\rightarrow_\beta^* [[succ]]([[succ]]((\lambda x.F(x\ x))(\lambda x.F(x\ x)))[[nil]])$
$\rightarrow_\beta [[succ]]([[succ]](F\ ((\lambda x.F(x\ x))(\lambda x.F(x\ x))))[[nil]]])$
$= [[succ]]([[succ]]((\lambda f.\lambda l.([[null?]]\ l)\ [[0]]\ ([[succ]]\ (f\ ([[cdr]]\ l)))\ ((\lambda x.F(x\ x))(\lambda x.F(x\ x)))[[nill]])$
$\rightarrow_\beta^2 [[succ]]([[succ]]\ ([[null?]][[nil]])\ [[0]]\ ([[succ]]\ ((\lambda x.F(x\ x))(\lambda x.F(x\ x))))\ ([[cdr]][[nil]]))$
$\rightarrow_\beta^* [[succ]]([[succ]]\ [[true]]\ [[0]]\ ([[succ]]\ ((\lambda x.F(x\ x))(\lambda x.F(x\ x))))\ ([[cdr]][[nil]]))$
$\rightarrow_\beta^2 [[succ]]([[succ]]\ 0)$
$\rightarrow_\beta^2 [[2]]$

- Works under NOR, but not under AOR.

For recursion under eager evaluation, need 3 things:

1) Modified reduction strategy - Applicative Order Evaluation (AOE)
   - choose the leftmost, innermost redex that is not within the body of an abstraction
2) Modified Y combinator
   - $Y' = \lambda f.(\lambda x.f(\lambda y.x\ x\ y))(\lambda x.f(\lambda y.x\ x\ y))$
     Note: $Y' \rightarrow_\eta^2 Y$
3) "Short-circuit if-then-else"

   [[if B then T else F]] [[B]] $(\lambda x.[[T]])$ $(\lambda x.[[F]])$ x

1.3 Scheme & Functional Programming

See notes, Chapter 3

- "pure" functional programming - No side effects, no mutation, no mutation, no state, no I/O

$\Rightarrow$ referential transparency - output of a function is completely determined by its input

Consequence - "equals can be substituted for equals"

e.g. (let ((z (f 3))) $\cdots$) - anywhere z occurs, can substitute (f 3), and vice versa

- Not possible in the presence of side-effects.

Higher-Order Functions

Functions are **first-class values** - can be

1) passed as parameters
2) returned by functions
3) stored as data

Functions that take or return other functions are called **higher-order functions**

e.g. map, foldl, foldr

Consider: $\lambda x.\lambda y.x$ $(lambdax.\lambda y.x)\ a\ b \rightarrow^2_\beta a$

- takes 2 arguments and returns the first

In Scheme: (lambda (x) (lambda (y) x))

(((lambda (x) (lambda (y) x)) 1) 2)

This form (lambda (x) (lambda (y) x)) called **curried**.

- simulate multi-argument functions with functions that return functions

Advantage - partial application

(define plus (lambda (x) (lambda (y) (+ x y))))

(define f (plus 5)) - save for later use - f is a function that adds 5

(f 1) $\Rightarrow$ 6

(f 8) $\Rightarrow$ 13

Implemented
- first-class functions implemented as a **closure**

- a pair [function code — env]

- pointer to function code, pointer to the environment in which the function was defined.

This is how f knows what x was when (plus 5) was called.

2 Type Theory

Consider [[true id]]

$$\begin{aligned}[[true]][[id]] &= (\lambda x.\lambda y.x)(\lambda z.z) \\ &\rightarrow_\beta \lambda y.\lambda z.z \\ &= [[false]]\end{aligned}$$

(true id) produces false. Makes sense? NO!

Unrestricted combinations ⇒ unexpected results. - only do things that make sense

How? Introduce a system of **types**

- set of values an expression can have

- interpretation of raw data

Type System - set of types and set of rules for assigning types

An expression is **well-typed** if a type is derivable for it from the type rules - else it is **ill-typed**

Strong vs weak typing - how strictly are type rules enforced?

Static vs Dynamic Typing

- When are types determined
  - static-at-compile-time (C)
  - dynamic-at-run-time (Scheme)

Monomorphic vs. Polymorphic typing

monomorphic- entities have a unique type

polymorphic- entities can have multiple types

We study strong,static, monomorphic (for now) typing.

2.1 The Simply-Typed λ-Calculus (monomorphic

Syntax:

$$\begin{aligned}
< expr > &::=< var > \;|\; < abs > \;|\; < app > \\
< var > &::= a \mid b \mid c \mid \cdots \\
< abs > &::= \lambda < var > : < type > . < expr > \\
< app > &::=< expr >< expr > \\
< type > &::=< primitive > \;|\; < constructed > \\
< primitive > &::= t_1 | t_2 | \cdots \\
< constructed > &::=< type > \rightarrow < type >
\end{aligned}$$

Primitive types - "built-in", e.g. int, bool, t_1, t_2, ...

Constructed types

- built from other types
- component types and type constructor

e.g. int $\rightarrow$ bool - "function taking int and returning bool"

- the $\rightarrow$ is type constructor for functions

$\rightarrow$: right-to-left associative

$t_1 \rightarrow t_2 \rightarrow t_3 = t_1 \rightarrow (t_2 \rightarrow t_3) \neq (t_1 \rightarrow t_2) \rightarrow t_3$

2.2 Type Checking/Inference

Type theory $\cong$ Intuitionist Implicational Logic (subset of propositional logic)

- called the Curry-Howard Isomorphism

Rules expressed as a formal inference system.

$\frac{premises}{conclusion}$

$\frac{P_1 \cdots P_n}{C}$ = "If we can construct proofs of $P_1 \cdots P_n$, we have a proof of C.

No premises: $\frac{}{conclusion}$ - axioms - conclusion always holds

Eg.(logic): "modus ponens" $\frac{p \rightarrow q \; p}{q}$

$\frac{a\ b}{a \wedge b}$ ($\wedge$ introduction)

$\frac{a \wedge b}{a}$ $\frac{a \wedge b}{b}$ ($\wedge$ elimination)

2.3 Type rules for Simply Typed λ-Calculus

type environment

- map from identifiers to types

- list of ¡name,type¿ pairs - ”symbol table”

- denoted A (”assumptions”) or Γ

A(x) = ”look up x in A” - if $< x, \tau > \in$ A, A(x) = τ

Type judgement - statement of the form $A \vdash E : \tau$

- $\vdash$ turnstile $\equiv$ derivability

Variables - look up in environment

The course notes says $\frac{A(x)=\tau}{A \vdash x:\tau}$ $[var]$

We are going with $\frac{}{A \vdash x:A(x)}$ $[var]$ to save some writing

Abstractions - assume the param has the given type, then type the body

$$\frac{A+ < x_1, \tau_1 > \vdash E : \tau_2}{A \vdash (\lambda x : \tau_1 \ . \ E) : \tau_1 \to \tau_2} \ [abs]$$

Applications

- type the rator and rand

- type of the rand must match the param type of the rator

- type of expr is the result type of the rator

$$\frac{A \vdash M : \tau_1 \to \tau_2 \ \ A \vdash N : \tau_1}{A \vdash M \ N : \tau_2} \ [app]$$

$$\frac{\tau_1 \to \tau_2 \ \ \tau_1}{\tau_2}$$

Ex. find the type of $\lambda x : t_1 .x$

$$\dfrac{\dfrac{}{\{<x,t_1>\}\vdash x:t_1}\,[var]}{\{\}\vdash(\lambda x:t_1.x):t_1\to t_1}\,[abs]$$

Eg. $\lambda x:t_1.\lambda y:t_1\to t_2.\ y\ x$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{<x,t_1>,<y,t_1\to t_2\}\vdash y:t_1\to t_2}\,[var]\quad\dfrac{}{\{<x,t_1>,<y,t_1\to t_2\}\vdash x:t_1}\,[var]}{\{<x,t_1>,<y,t_1\to t_2>\}\vdash y\ x:t_2}\,[app]}{\{<x,t_1>\}\vdash(\lambda y:t_1\to t_2.\ y\ x):(t_1\to t_2)\to t_2}\,[abs]}{\{\}\vdash(\lambda x:t_1.\lambda y:t_1\to t_2.\ y\ x):t_1\to(t_1\to t_2)\to t_2}\,[abs]$$

2.4 Evaluating Type Systems

- How do we know these type rules work?

2 Theorems: **Progress** and **Preservation**

Theorem (progress): Let E be a closed, well-typed term in the Simply Typed λ Calculus, i.e. for some A, τ, $A\vdash E:\tau$. Then either E (E is a value) is in Weak Normal Form or there is an expression E' such that $E\to_\beta E'$.

Proof: Induction on the length of the type derivation for E.

Case 1: E is a variable - impossible if E is closed.

Case 2: E is an abstraction $\Rightarrow$ E is in Weak Normal Form, done.

Case 3: E is an application, $E=M\ N$. Then the type derivation for E looks like

$$\dfrac{\dfrac{\dots}{A\vdash M:\tau_1\to\tau_2}\quad\dfrac{\dots}{A\vdash N:\tau_1}}{A\vdash E:\tau}$$

By induction, M is in Weak Normal Form or reducible

By induction, N is in Weak Normal Form or reducible

If M or N is reducible, then so is E since $E=M\ N$

If neither is reducible, then both are values

In that case, M is a value, $M=\lambda x.M'$, By app rule, M's type derivation must look like

$$\dfrac{\dots}{A\vdash(\lambda x.\tau_1.M'):\tau_1\to\tau}\,[abs]$$

So $E=M\ N=(\lambda x:\tau_1.M')\ N$ is reducible. (N is τ_1)

QED.

Simple, but less so when the language is enhanced.

Progress $\Rightarrow$ can't get stuck, e.g. can't derive $(\lambda x : t_1.A)\ B$ where B is t_2

Theorem (Preservation, aka Subject Reduction Theorem): Let A be a type environment, P, Q be expressions such that $P \to^*_{\beta\eta} Q$ (i.e. P reduces to Q in 0 or more β and/or η reductions). Suppose $A \vdash P : \tau$ for some τ. Then $A \vdash Q : \tau$.

Proof: see notes, page 57-58

Preservation: Well-typed terms remain well-typed after reduction

Progress and Preservation $\Rightarrow$ Safety.

Star with E which is well typed, Progress says E is reducible or E is WNF (done). Then if E is not done $E \to_\beta E'$, from this preservation says E' is well-typed, and then progress says E' is reducible or E' is done. This means $E' \to_\beta E''$, and then Preservation says E'' is well typed, etc.

$\therefore$ Well-typed terms either reduce forever, or make a "sensible" value.

$\therefore$ Our type rules "make sense"

Strong Normalization Theorem: The set of well-typed terms in the Simply Typed λ-Calculus is **strong normalizing** - that is, infinite reductions are impossible.

Proof: Appendix B.

$\Rightarrow$ Simply typed λ-Calculus not Turing-complete. (Can't simulate no terminating turing machine)

Intuition: Consider $(\lambda x.x\ x)(\lambda x.x\ x)$

What type can we give $\lambda x.x\ x$ - How do we type $x\ x$

If the 2nd x has some type τ_1, then 1st x must have type $\tau_1 \to \tau_2$ for some τ_2

No way τ_1 and $\tau_1 \to \tau_2$ can represent the same type. $\therefore$ no type for x, $\lambda x.x\ x$, $(\lambda x.x\ x)(\lambda x.x\ x)$

Can't type Y either.

2.5 Polymorphism

-exprs may have multiple types

Cardelli and Wegner (computing Surveys 1985) - polymorphism hierarchy

Polymorphism

- universal
    - parametric
    - inclusion
- adhoc
    - Overloading
    - (implicit) coercion

Universal

- generally, one implementation with many types
- unbounded number of specializations
- works on unknown (not yet created types
- Parametric
    - "type parameter" (often implicit)

  e.g. function id (x:t) : t

  return x;

  ≈ generic programming ML, Haskell
- Inclusion
    - "subtyping" hierarchy of types
    - can use a subtype in any context that calls for a supertype (OOP)

Adhoc

- generally, several implementations
- bounded number of specializations
- works on only known (existing) types
- Overloading
    - "name sharing"
    - multiple functions with the same name in the same scope
    - compiler chooses the correct instance based on
        1) Number and type of arguments
        2) (possibly) return type

- this is called "overload resolution"

e.g. Ada

- Coercion
  - Implicit
    - expression automatically converted to a different type

    e.g. C, 3.5 + 4 = 7.5, 4 is an implicit conversion to double
  - Explicit
    - conversion invoked by the programmer
    - "casting"

For now: parametric polymorphism

2.6 System F (or 2nd order λ calculus

- discovered independently by Girard ("System F"), and Reynolds ("polymorphic λ-calculus")

- models parametric polymorphism
e.g.

$\lambda x : int.x$ - id for ints

$\lambda x : bool.x$ - id for bools

$\lambda x : int \to int.x$ - id for int functions

- same implementation, different annotations

idea; make the type a parameter to the function

Syntax::

$$\begin{aligned}
< expr > &::=< var > \mid < abs > \mid < app > \mid < t - abs > \mid < t - app > \\
< var > &::= a|b|c| \cdots \\
< abs > &::= \lambda < var >:< type > . < expr > \\
< app > &::=< expr >< expr > \\
< t - abs > &::= \wedge < t - var > . < expr > \\
< t - app > &::=< expr > \{< type >\} \\
< type > &::=< prim > \mid < t - var > \mid < type >\to< type > |\forall < t - var > . < type > \\
< prim > &::= t_1|t_2| \cdots \\
< t - var > &::= \alpha|\beta|\gamma| \cdots
\end{aligned}$$

variables,app - as before

t-abs,t-app - "type abstraction", "type application"

type abstraction - function parameterized by type

t-var - "type variable" - placeholder for types

$\forall < t - var > . < type >$ - quantified type - the type of a type abstraction

Ex. $\lambda x : int.x$ int $\rightarrow$ int, and $\lambda x : bool.x$ bool $\rightarrow$ bool - Now replace int,bool by a type variable

$(\lambda x : \alpha.x) : \alpha \rightarrow \alpha$

α - a free type variable - meaning depends on external context

Now abstract over α:

$$id = (\wedge\alpha.\lambda x : \alpha.x) : \forall\alpha.\alpha \rightarrow \alpha$$

To apply id:

$$\begin{aligned} id\ \{int\}\ 3 &= (\wedge\alpha.\lambda x : \alpha.x)\ \{int\}\ 3 \\ &\rightarrow_\beta (\lambda x : int.x)\ 3 \\ &\rightarrow_\beta 3 \end{aligned}$$

$$\begin{aligned} id\ \{bool\}\ true &= (\wedge\alpha.\lambda x : \alpha.x)\ \{bool\}\ true \\ &\rightarrow_\beta (\lambda x : bool.x)\ true \\ &\rightarrow_\beta true \end{aligned}$$

Type rules:

$$\frac{}{A \vdash x : A(x)}\ [var]$$

$$\frac{< x, \tau_1 > + A \vdash E : \tau_2}{A \vdash (\lambda x : \tau_1.E) : \tau_1 \rightarrow \tau_2}\ [abs]$$

$$\frac{A \vdash M : \tau_1 \rightarrow \tau_2 \quad A \vdash N : \tau_1}{A \vdash (MN) : \tau_2}\ [app]$$

$$\frac{A \vdash E : \tau}{A \vdash (\wedge\alpha.E) : \forall\alpha.\tau}\ [t - abs](\alpha \text{ not free in A})$$

$$\frac{A \vdash E : \forall\alpha.\tau_1}{A \vdash (E\{\tau_2\}) : \tau_1[\tau_2/\alpha]}[t - app]$$

Type $\forall\alpha.\alpha \to \alpha \equiv$ for any type τ, the function can be given type $\tau \to \tau$.

$\forall\alpha$ is the binding occurrence

The αs afterwards are the bound occurrences

- same substitution, α-equivalence rules for types as for λ terms.

Nested quantifiers:

Say $f : (\forall\alpha.\alpha \to \alpha) \to int$

parameter type : $\forall\alpha.\alpha \to \alpha$

result type: int

$\therefore$ f **requires** a polymorphic function as argument, then returns int

$g : ((\forall\alpha.\alpha \to \alpha) \to int) \to (\forall\beta.\beta \to (\beta \to \beta))$

g requires (a function requiring a polymorphic argument) as its argument, returns a polymorphic result

$h : int \to (\forall\alpha.\alpha \to \alpha)$

- h takes an int and returns a polymorphic function

$k : \forall\alpha.int \to (\alpha \to \alpha)$

- polymorphic - for any τ, k takes int and returns $\tau \to \tau$

System F types more terms than the simply typed λ calculus

Can now do self-application

Before: $(\lambda x : int \to int.x)(\lambda x : int.x) \to_\beta (\lambda x : intx)$ (they are not the same)

Now: $id = \wedge\alpha.\lambda x : \alpha.x \quad \forall\alpha.\alpha \to \alpha$

Can code id id as

$$
\begin{aligned}
& id\ \ \{\forall\alpha.\alpha \to \alpha\}\ \ id \\
& = (\wedge\alpha.\lambda x : \alpha.x)\{\forall\alpha.\alpha \to \alpha\}(\wedge\alpha.\lambda x : \alpha.x) \\
& \to_\beta (\lambda x : (\forall\alpha.\alpha \to \alpha).x)(\wedge\alpha.\lambda x : \alpha.x) \\
& \to_\beta (\wedge\alpha.\lambda x : \alpha.x) \\
& = id
\end{aligned}
$$

Can abstract out the self-application:

$$id\ id =_\beta (\lambda x.x\ x)\ id$$

So

$$
\begin{aligned}
& (\wedge\alpha.\lambda x : \alpha.x)\{\forall\alpha.\alpha \to \alpha\}(\wedge\alpha.\lambda x : \alpha.x) \\
& = (\lambda x : (\forall\alpha.\alpha \to \alpha).x\{\forall\alpha.\alpha \to \alpha\}\ x)(\wedge\alpha.\lambda x : \alpha.x)
\end{aligned}
$$

So $\lambda x : (\forall\alpha.\alpha \to \alpha).x\{\forall\alpha.\alpha \to \alpha\}\ x$ is a well-typed implementation of $\lambda x.x\ x$

$$
\frac{\dfrac{\dfrac{\dfrac{}{\{<x,\forall\alpha.\alpha\to\alpha>\}\vdash x:\forall\alpha.\alpha\to\alpha}\,[var]}{\{<x,\forall\alpha.\alpha\to\alpha>\}\vdash(x\{\forall\alpha.\alpha\to\alpha\}):(\forall\alpha.\alpha\to\alpha)\to(\forall\alpha.\alpha\to\alpha)}\,[t-app] \quad \dfrac{}{\{<x,\forall\alpha.\alpha\to\alpha>\}\vdash x:\forall\alpha.\alpha\to\alpha}\,[var]}{\{<x,\forall\alpha.\alpha\to\alpha>\}\vdash(x\ \{\forall\alpha.\alpha\to\alpha\}\ x):\forall\alpha.\alpha\to\alpha}\,[app]}{\{\} \vdash (\lambda x : (\forall\alpha.\alpha \to \alpha).x\{\forall\alpha.\alpha \to \alpha\}\ x) : (\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)}\,[abs]
$$