DD2552 Seminar 1: Functional languages, abstract syntax trees, variable binding, and inductive definitions

Karl Palmskog

KTH

Wednesday August 30, 2023

KTH Research Preparation course

Module D: PL Semantics

Pablo Buiras

Course material

- PFPL chapter 1 and chapter 2
- https://lawrencecpaulson.github.io/papers/Aczel-Inductive-Defs.pdf
- https://www.cs.cmu.edu/~rwh/pfpl/supplements/ulc.pdf

Object language vs. meta language

- in this course, we define syntax and semantics of some (small) languages
- the language being defined is usually called the **object language**
- the language we are using to define object languages is called the **meta language**
- our typical meta language is "mathematical English", but could also be some foundational formalism like ZF set theory or constructive type theory
- the "ML" in Standard ML stands for meta language

Grammars and derivations

Consider the following grammar:

$$\begin{aligned}\text{Expr} \to \text{Num} \mid \text{Expr} + \text{Expr} \mid \text{Expr} - \text{Expr} \\ \mid \text{Expr} * \text{Expr} \mid \text{Expr} \,/\, \text{Expr} \mid (\text{Expr})\end{aligned}$$

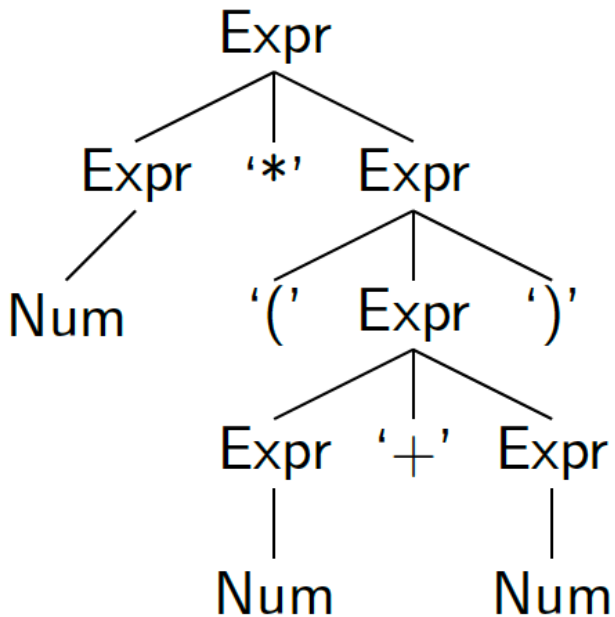
We can derive `"4*(3+5)"`:

$$\begin{aligned}\text{Expr} \to \text{Expr} * \text{Expr} \to \text{Num} * \text{Expr} \to \text{Num} * (\text{Expr}) \to \\ \text{Num} * (\text{Expr} + \text{Expr}) \to \text{Num} * (\text{Num} + \text{Expr}) \to \\ \text{Num} * (\text{Num} + \text{Num})\end{aligned}$$

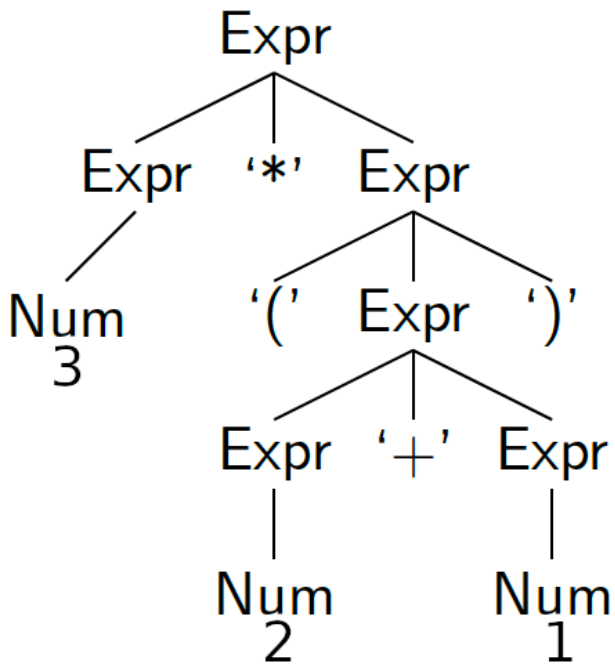
Abstract Syntax Trees

An abstract syntax tree (AST) can represent a set of derivations.

$$\underline{\text{Expr}} \to \underline{\text{Expr}} * \text{Expr} \to \text{Num} * \underline{\text{Expr}} \to \text{Num} * (\underline{\text{Expr}}) \to$$
$$\text{Num} * (\underline{\text{Expr}} + \text{Expr}) \to \text{Num} * (\text{Num} + \underline{\text{Expr}}) \to$$
$$\text{Num} * (\text{Num} + \text{Num})$$



Adding data to ASTs



Variables and substitution

Consider a grammar with variables:

$$\text{Expr} \rightarrow \text{Num} \mid \text{Expr} + \text{Expr} \mid Var$$

If we have an expression e, we can *substitute* a variable x for a number n:

- $[b/x]x = b$ and $[b/x]y = y$ if $x \neq y$
- $[b/x]o(a_1, \ldots, a_n) = o([b/x]a_1, \ldots, [b/x]a_n)$

Variables binding in grammars

$$\text{Expr} \to \text{Num} \mid \text{Expr} + \text{Expr} \mid \text{Var} \mid \text{Let Var} = \text{Expr In Expr}$$

Now the second substitution rule won't work well anymore:

- $[b/x]x = b$ and $[b/x]y = y$ if $x \neq y$
- $[b/x]o(a_1, \ldots, a_n) = o([b/x]a_1, \ldots, [b/x]a_n)$

Solution: rename variables to avoid capture (see PFPL for details)

ASTs and structural induction

Consider a more limited grammar:

$$\text{Expr} \to \text{Num} \mid \text{Expr} + \text{Expr}$$

We want to prove a property P holds for all ASTs in this grammar. It then suffices to:

- prove $P(n)$ for all numbers n
- assume $P(e)$ and $P(e')$ and prove $P(e + e')$

Why does this work? We cover all ways of forming strings according to grammar.

Inductive definitions ("judgments")

$$\begin{array}{c} J_1 \\ \dots \\ J_n \\ \hline J \end{array}$$

or

$$\frac{J_1 \dots J_n}{J}$$

- define relation or mathematical structure via rules
- relation name can occur in J_i ("recursive call")
- rules can only be applied finite number of time

Inductive definition examples

$$\frac{}{n \Downarrow n}\ \textsc{os\_\_eval\_\_num}$$

$$\frac{\begin{array}{l} e_1 \Downarrow n_1 \\ e_2 \Downarrow n_2 \\ n_1 + n_2 = n \end{array}}{e_1 + e_2 \Downarrow n}\ \textsc{os\_\_eval\_\_plus}$$

Inductive derivations

- derive a judgment by reducing it to other judgments via rules
- all reductions must terminate using rules without (inductive) premises
- derivations are **trees** with the desired judgment (conclusion) as root

Ott

- tool for writing grammars and inductive rules
- exportable to LaTeX (also Coq, HOL4, Isabelle/HOL)
- https://github.com/ott-lang/ott

Ott grammar example

```
grammar
e :: e_ ::=
  | x :: :: var {{ com variable }}
  | n :: :: num {{ com number }}
  | e + e' :: :: plus {{ com plus }}
  | e * e' :: :: times {{ com times }}
  | let x := e in e' :: :: def (+ bind x in e' +)
    {{ com let }}
  | e [ e' / x ] :: M :: subst
    {{ com substitution }}
    {{ coq (subst_e [[e']] [[x]] [[e]]) }}
  | ( e ) :: S :: parentheses
    {{ coq ([[e]]) }}
```

Ott grammar using generated LaTeX

e	$::=$			
	$\mid$	x		variable
	$\mid$	n		number
	$\mid$	$e + e'$		plus
	$\mid$	$e * e'$		times
	$\mid$	$\mathbf{let}\ x := e\ \mathbf{in}\ e'$	bind x in e'	let
	$\mid$	$e[e'/x]$	M	substitution
	$\mid$	(e)	S	

Ott rules example

```
defn
  e -> e' :: :: red :: red_
  {{ com reduction step }} by

  n1 + n2 = n
  ------------ :: plus
  n1 + n2 -> n

  e1 -> e'1
  -------------------- :: plus_l
  e1 + e2 -> e'1 + e2
```

Ott rules using generated LaTeX

$$\frac{n_1 + n_2 = n}{n_1 + n_2 \to n} \quad \text{OS\_\_RED\_\_PLUS}$$

$$\frac{e_1 \to e_1'}{e_1 + e_2 \to e_1' + e_2} \quad \text{OS\_\_RED\_\_PLUS\_\_L}$$

Semantics of expressions using rules

$$\frac{e \to e'}{n + e \to n + e'} \quad \text{OS\_\_RED\_\_PLUS\_\_R}$$

$$\frac{n_1 * n_2 = n}{n_1 * n_2 \to n} \quad \text{OS\_\_RED\_\_TIMES}$$

$$\frac{e_1 \to e_1'}{e_1 * e_2 \to e_1' * e_2} \quad \text{OS\_\_RED\_\_TIMES\_\_L}$$

$$\frac{e \to e'}{n * e \to n * e'} \quad \text{OS\_\_RED\_\_TIMES\_\_R}$$

$$\frac{e_1 \to e_1'}{\mathbf{let}\, x := e_1 \,\mathbf{in}\, e_2 \to \mathbf{let}\, x := e_1' \,\mathbf{in}\, e_2} \quad \text{OS\_\_RED\_\_LET}$$

$$\frac{}{\mathbf{let}\, x := n \,\mathbf{in}\, e_2 \to e_2[n/x]} \quad \text{OS\_\_RED\_\_BIND}$$

Using the reduction relation

- we are given some expression AST e
- consider reflexive-transitive closure of $\rightarrow$ on expressions
- if $e \rightarrow^* n$, then n is the result of evaluating e
- to find and prove $e \rightarrow^* n$, we may have to do a lot of deriving

Alternative inductive relation

$$\frac{}{n \Downarrow n}\ \textsc{os\_\_eval\_\_num}$$

$$\frac{\begin{array}{c} e_1 \Downarrow n_1 \\ e_2 \Downarrow n_2 \\ n_1 + n_2 = n \end{array}}{e_1 + e_2 \Downarrow n}\ \textsc{os\_\_eval\_\_plus}$$

$$\frac{\begin{array}{c} e_1 \Downarrow n_1 \\ e_2 \Downarrow n_2 \\ n_1 * n_2 = n \end{array}}{e_1 * e_2 \Downarrow n}\ \textsc{os\_\_eval\_\_times}$$

$$\frac{\begin{array}{c} e_1 \Downarrow n_1 \\ e_2[n_1/x] \Downarrow n_2 \end{array}}{\textbf{let}\ x := e_1\ \textbf{in}\ e_2 \Downarrow n_2}\ \textsc{os\_\_eval\_\_let}$$

How is this related to $\rightarrow$?

DD2552 Seminar 2: Untyped Lambda Calculus

Karl Palmskog

KTH

Thursday August 31, 2023

Three basic ingredients

- Primitive elements
  - e.g. variables, numbers, references, etc.
- Means of combination
  - e.g. pairs, function calls, operators, sequencing, etc.
- Means of abstraction
  - e.g. procedures, blocks, functions, classes, etc.

Syntax

t	$::=$			term
	$\mid$	x		variable
	$\mid$	$\lambda x.t$	bind x in t	lambda
	$\mid$	$t\,t'$		app
	$\mid$	(t)	S	
	$\mid$	$[t/x]t'$	M	
v	$::=$			value
	$\mid$	$\lambda x.t$		lambda

Semantics

$$\frac{}{(\lambda x.t_{12})\ v_2 \longrightarrow [v_2/x]t_{12}}\ \textsc{ax\_\_app}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t \longrightarrow t_1'\ t}\ \textsc{ctx\_\_app\_\_fun}$$

$$\frac{t_1 \longrightarrow t_1'}{v\,t_1 \longrightarrow v\,t_1'}\ \textsc{ctx\_\_app\_\_arg}$$

λ-calculus syntax

x, y, z, ... variables

M, N ::= x (Variable)
| M N (Application)
| λx.M (Abstraction)

λx.x (λx.x) z (x y) z

λx.λy.y λx.y x (y z)

Naive substitution

$$(\lambda y.\lambda x.y)\ z \rightarrow \lambda x.z$$ constant function

$$(\lambda y.\lambda x.y)\ x \rightarrow \lambda x.x$$ identity function

- Meaning has changed, variable x has been *captured*
- When doing substitutions, we need to be careful not to change the meaning of the term, so ideally

$$(\lambda y.\lambda x.y)\ x \rightarrow \lambda w.x$$

Free variables

$$\frac{}{x \in \mathrm{FV}(x)} \quad \textsc{var}$$

$$\frac{x \in \mathrm{FV}(t_1)}{x \in \mathrm{FV}(t_1\ t_2)} \quad \textsc{app\_l}$$

$$\frac{x \in \mathrm{FV}(t_2)}{x \in \mathrm{FV}(t_1\ t_2)} \quad \textsc{app\_r}$$

$$\frac{\begin{array}{l} x \in \mathrm{FV}(t) \\ x \neq y \end{array}}{x \in \mathrm{FV}(\lambda y.t)} \quad \textsc{lam}$$

α-equivalence

- we can equate terms that differ only in bound variable names
- such terms are called α-**equivalent**

$$\frac{}{t \equiv_\alpha t}\ \text{AEQ\_\_ID} \qquad \frac{t \equiv_\alpha t'}{t' \equiv_\alpha t}\ \text{AEQ\_\_SYM}$$

$$\frac{t \equiv_\alpha t' \quad t' \equiv_\alpha t''}{t \equiv_\alpha t''}\ \text{AEQ\_\_TRANS} \qquad \frac{t_1 \equiv_\alpha t_1' \quad t_2 \equiv_\alpha t_2'}{t_1\ t_2 \equiv_\alpha t_1'\ t_2'}\ \text{AEQ\_\_APP}$$

$$\frac{t \equiv_\alpha t'}{\lambda x.t \equiv_\alpha \lambda x.t'}\ \text{AEQ\_\_LAM} \qquad \frac{x' \notin FV(t)}{\lambda x.t \equiv_\alpha \lambda x'.[x'/x]t}\ \text{AEQ\_\_SUBST}$$

λ-calculus semantics

Conversion / Reduction

$$\frac{}{(\lambda x.M)N \rightarrow_\beta M[N/x]}\ \beta$$

Context closure

$$\frac{M \rightarrow_\beta M'}{M\ N \rightarrow_\beta M'\ N} \qquad \frac{N \rightarrow_\beta N'}{M\ N \rightarrow_\beta M\ N'} \qquad \frac{M \rightarrow_\beta M'}{\lambda x.M \rightarrow_\beta \lambda x.M'}$$

Reduction examples

```
   (λx.x y) (λz.x)
→ (λz.x) y
→ x
```

```
(λx.λy.x) (λy.y)
```

```
z y (λx.(λy.x) z)
```

Properties of reduction

- order of applying reduction rules could vary
- do we still get “same” results?
- is there a point where no reduction rule applies?

β-equivalence

$$\frac{}{t \equiv_\beta t}\ \textsc{beq\_\_id} \qquad \frac{t \equiv_\beta t'}{t' \equiv_\beta t}\ \textsc{beq\_\_sym}$$

$$\frac{t \equiv_\beta t' \quad t' \equiv_\beta t''}{t \equiv_\beta t''}\ \textsc{beq\_\_trans} \qquad \frac{t_1 \equiv_\beta t_1' \quad t_2 \equiv_\beta t_2'}{t_1\ t_2 \equiv_\beta t_1'\ t_2'}\ \textsc{beq\_\_app}$$

$$\frac{t \equiv_\beta t'}{\lambda x.t \equiv_\beta \lambda x.t'}\ \textsc{beq\_\_lam} \qquad \frac{}{(\lambda x.t)\ t' \equiv_\beta [t'/x]t}\ \textsc{beq\_\_subst}$$

Normal forms

- A term N is a **normal form** if there is no Z such that $N \to Z$
  - e.g. x, x y, $(\lambda x.x)$
  - in other words, there is no redex
- A **normal form of** M is a term N such that $M \to^* N$ and N is a normal form
- Does every term have a normal form?

β-equivalence

- $M =_\beta N$ iff M can be transformed into N by zero or more reduction steps and/or inverse reduction steps
  - Reflexive symmetric transitive closure of →
  - The smallest equivalence relation containing →

$(\lambda x.\lambda y.x)\ (\lambda z.z)\ z =_\beta \lambda z.z$

$(\lambda z.z)\ x\ y =_\beta x\ ((\lambda z.y)\ x)$

$x\ y =_\beta x\ ((\lambda z.y)\ x)$

λ-calculus semantics

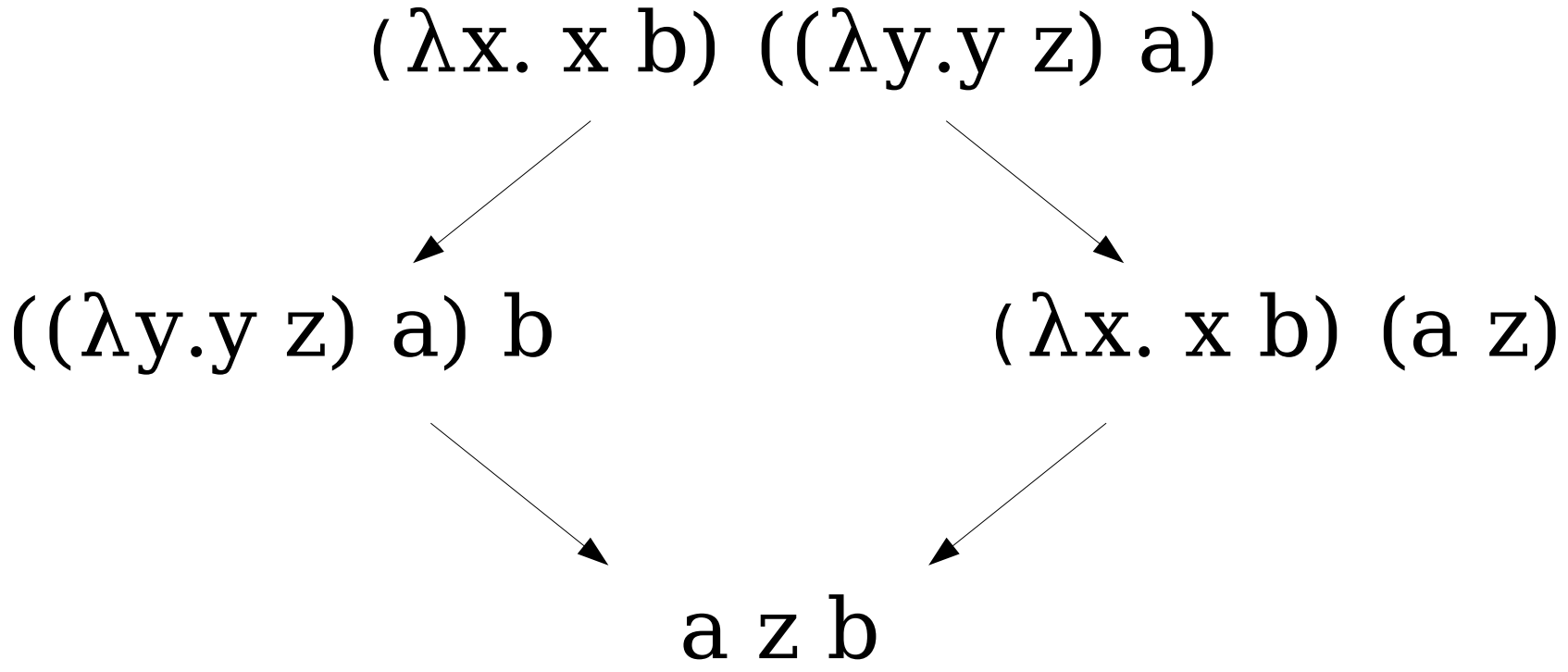
- Small-step operational semantics
  - $\rightarrow \subseteq \Lambda \times \Lambda$ (conversion or reduction)
  - $\rightarrow^*$ is the reflexive-transitive closure

- The meaning of a term is determined by succesive conversion ($\rightarrow^*$) until a normal form is reached, if it exists

β-equivalence as a rewriting system

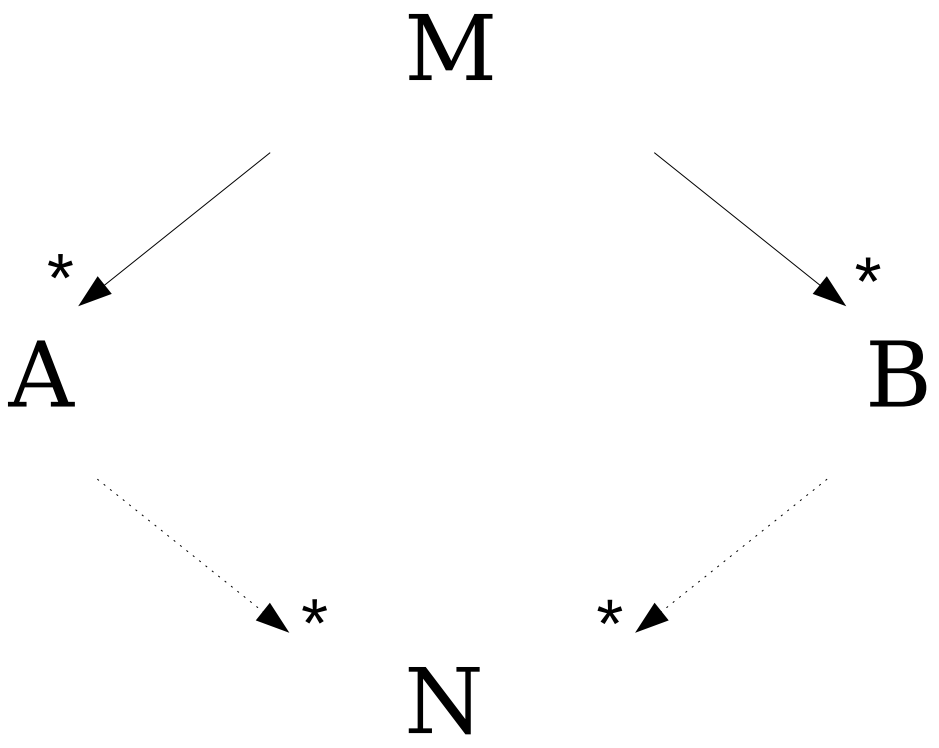
- does order of β rewriting matter?
  - no, by Church-Rosser theorem / confluence
- do we always reach "normal form"?
  - no, and no way to decide when we do

Multiple redexes



Church-Rosser property

For all M
s.t.



exists N
s.t.

Proof. See Tait and Martin-Löf

NFs and Church-Rosser

- Normal forms are unique
- If $M =_\beta N$ and M, N are normal forms, then $M =_\alpha N$
- If $M =_\beta N$, then either both have a normal form or both do not.
- If M is a normal form and $M =_\beta N$, then $M \rightarrow^* N$

Programming in λ-calculus

Booleans

- true ≡(λx y.x) = (λx.λy.x)
- false≡(λx y.y)
- if-then-else = (λb.b)

if-then-else true M N $=_\beta$ M

if-then-else false M N $=_\beta$ N

Programming in λ-calculus

Natural numbers

- $Z \equiv (\lambda fz.z)$
- $S \equiv (\lambda nfz.f\ (n\ f\ z))$
- This repr. can be used to iterate over numbers

$plus \equiv \lambda nm.\ n\ S\ m$

$plus\ Z\ M =_\beta M$

$plus\ (S\ N)\ M =_\beta S\ (plus\ N\ M)$

$isZero \equiv \lambda n.\ n\ (\lambda m.false)\ true$

Programming in λ-calculus

Natural numbers – part 2

- $\overline{n} \equiv S^n\ Z = (\lambda fz.f^n\ z)$ (can be proved by ind.)

$\mathsf{mult} \equiv \lambda nmf.n\ (m\ f)$

$\mathsf{mult}\ Z\ M =_\beta Z$

$\mathsf{mult}\ (S\ N)\ M =_\beta$

$\mathsf{plus}\ (\mathsf{mult}\ N\ M)\ M$

$\mathsf{mult}\ \overline{n}\ \overline{m} =_\beta \overline{n*m}$

$\mathsf{pred} \equiv ?$ such that $\mathsf{pred}\ (S\ N) =_\beta N$

Programming in the λ-calculus

Pairs, lists

$\langle M,N \rangle \equiv \lambda z.z\ M\ N$ $\quad \pi_1 \equiv \lambda p.p\ (\lambda xy.x)$ $\quad \pi_2 \equiv \lambda p.p\ (\lambda xy.y)$

$\pi_1 \langle M,N \rangle =_\beta M$ $\quad \pi_2 \langle M,N \rangle =_\beta N$

$nil \equiv \lambda fz.z$

$cons \equiv \lambda htfz.f\ h\ (t\ f\ z)$

$isEmpty \equiv \lambda l.l\ (\lambda xr.false)\ true$

$length \equiv \lambda l.l\ (\lambda xr.S\ r)\ Z$

$isEmpty\ nil =_\beta true$

$isEmpty\ (cons\ M\ N) =_\beta false$

$length\ nil =_\beta 0$

$length\ (cons\ h\ t) =_\beta plus\ 1\ (length\ t)$

Programming in the λ-calculus

let-expressions

$(\text{let } x = M \text{ in } N) \equiv (\lambda x.N)\ M$

$(\text{let } x = M \text{ in } N) =_\beta M[N/x]$

$(\text{let } x = \overline{2} \text{ in plus } x\ x)$
$=_\beta (\lambda x.\text{plus } x\ x)\ \overline{2}$
$=_\beta \text{plus}\ \ \overline{2}\ \overline{2}$

Evaluation order

- Applicative order (rightmost, innermost)

  $(\lambda x.M)\ N \rightarrow (\lambda x.M)\ N' \rightarrow \ldots \rightarrow (\lambda x.M)\ Z \rightarrow M[Z/x]$ (Z normal form) $\rightarrow \ldots$

- Normal order (leftmost, outermost)

  $(\lambda x.M)\ N \rightarrow M[N/x] \rightarrow \ldots$

Evaluation order examples

Applicative

```
(λx.x x) ((λxy.y) z)
→ (λx.x x) (λy.y)
→ (λy.y) (λy.y)
→ (λy.y)
```

Normal

```
(λx.x x) ((λxy.y) z)
→ ((λxy.y) z) ((λxy.y) z)
→ (λy.y) ((λxy.y) z)
→ ((λxy.y) z)
→ (λy.y)
```

Best strategy?

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

Applicative

$$(\lambda x.x)\ \Omega$$
$$= (\lambda x.x)\ ((\lambda x.x\ x)\ (\lambda x.x\ x))$$
$$\rightarrow (\lambda x.x)\ ((\lambda x.x\ x)\ (\lambda x.x\ x))$$
$$\rightarrow \ldots$$

Normal

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

Theorem. If a term has a normal form, normal order reduction will find it.

λ-calculus with evaluation order

CBV

Call-by-value (CBV)
λ-calculus

(applicative order)

x, y, z, ... variables

M, N ::= x terms
| M N
| λx.M

v ::= λx.M values

$$\frac{}{(\lambda x.M)v \rightarrow_\beta M[v/x]}\ \beta$$

$$\frac{M \rightarrow_\beta M'}{M\ N \rightarrow_\beta M'\ N}$$

$$\frac{N \rightarrow_\beta N'}{v\ N \rightarrow_\beta v\ N'}$$

λ-calculus with evaluation order

CBN

Call-by-name (CBN)
λ-calculus

(normal order)

x, y, z, ... variables

M, N ::= x terms
| M N
| λx.M

v ::= λx.M values

$$\frac{}{(\lambda x.M)N \rightarrow_\beta M[N/x]}\ \beta$$

$$\frac{M \rightarrow_\beta M'}{M\ N \rightarrow_\beta M'\ N}$$

Fixed points

- N is a fixed point of F if $F\ N =_\beta N$
- **Theorem.** In the untyped λ-calculus, every term T has a fixed point.
  - $Y = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$
  - $Y\ T =_\beta (\lambda x.\ T\ (x\ x))\ (\lambda x.\ T\ (x\ x))$
    $=_\beta T\ ((\lambda x.\ T\ (x\ x))\ (\lambda x.\ T\ (x\ x)))$
    $=_\beta T\ (Y\ T)$
  - Y is known as *Curry's paradoxical combinator*

Programming in λ-calculus

Recursion

fact n = if-then-else (isZero n) $\overline{1}$ (mult n (fact (pred n)))

fact =
(λf.λn.if-then-else (isZero n) $\overline{1}$ (mult n (f (pred n)))) fact

fact = F fact

fact = Y F

Recursion, example

$$
\begin{aligned}
\text{fact } \overline{2} &=_\beta F\ (\text{fact } \overline{2}) \\
&=_\beta \text{if-then-else } (\text{isZero } \overline{2})\ \overline{1}\ (\text{mult } \overline{2}\ (\text{fact } (\text{pred } \overline{2}))) \\
&=_\beta \text{if-then-else false } \overline{1}\ (\text{mult } \overline{2}\ (\text{fact } (\text{pred } \overline{2}))) \\
&=_\beta \text{mult } \overline{2}\ (\text{fact } (\text{pred } \overline{2})) \\
&=_\beta \text{mult } \overline{2}\ (\text{fact } \overline{1}) \\
&=_\beta \text{mult } \overline{2}\ (F\ (\text{fact } \overline{1})) \\
&=_\beta \ldots \\
&=_\beta \text{mult } \overline{2}\ (\text{mult } \overline{1}\ (\text{fact } \overline{0}))
\end{aligned}
$$

Recursion, example (pt 2)

$$
\begin{aligned}
&=_\beta \text{mult } \overline{2}\ (\text{mult } \overline{1}\ (\text{fact } \overline{0})) \\
&=_\beta \text{mult } \overline{2}\ (\text{mult } \overline{1}\ (\text{if-then-else } (\text{isZero } \overline{0})\ \overline{1}\ (\ldots))) \\
&=_\beta \text{mult } \overline{2}\ (\text{mult } \overline{1}\ (\text{if-then-else true } \overline{1}\ (\ldots))) \\
&=_\beta \text{mult } \overline{2}\ (\text{mult } \overline{1}\ \overline{1}) \\
&=_\beta \overline{2}
\end{aligned}
$$

Fixed point intuition

	0	1	2	...	n-1	
$\perp$	$\perp$	$\perp$	$\perp$	...	$\perp$	λn . blah
$F^1 \perp$	1	$\perp$	$\perp$	...	$\perp$	λn . if-then-else (isZero n) $\overline{1}$ (mult n ($\perp$ (pred n)))
$F^2 \perp$	1	1	$\perp$	...	$\perp$	λn . if-then-else (isZero n) $\overline{1}$ (mult n ((λn . if-then-else (isZero n) $\overline{1}$ (mult n ($\perp$ (pred n)))) (pred n)))
...						
$F^n \perp$	1	1	2	...	(n-1)!	...
Y F						<infinite number of unfoldings>

Other λ-calculus applications

- Computability
  - Turing-completeness
- Compilers
  - Many functional languages can be internally translated into λ-terms
- Logic
  - Proofs as mathematical objects