Syntax

\[ \textit{M, N ::= } \begin{align*} & x \quad \text{variables} \\ & \mid \lambda x. M \quad \text{functions} \\ & \mid M \ N \quad \text{applications} \end{align*} \]

- Application associates leftward
  \[ xyzw = ((xy)z)w \]

- Function bodies as as large as possible.
  \[
  \begin{align*}
  \lambda x. \lambda y. fyx &= \lambda x. (\lambda y. fyx) = \lambda x. (\lambda y. (fyx)) \\
  &= \lambda x. (\lambda y. ((fy)x)) \\
  \lambda x. \lambda y. f(yx) &= \lambda x. (\lambda y. (f(yx))) \\
  \lambda x. (\lambda y. fy)x &= \lambda x. (((\lambda y. (fy))x))
  \end{align*}
  \]
• The relation $\rightarrow_\beta$ is defined by:

\[
\begin{align*}
(\lambda x. M) N &\rightarrow_\beta M[x \rightarrow N] \\
M \rightarrow_\beta M' &\frac{M N \rightarrow_\beta M' N'}{M N \rightarrow_\beta M' N'} \\
N \rightarrow_\beta N' &\frac{M N \rightarrow_\beta M N'}{M N \rightarrow_\beta M N'} \\
\end{align*}
\]
Review

\[(\lambda b. \lambda x. \lambda y. b \ y \ x)(\lambda w. \lambda z. w)\] (not tt)
\[
\rightarrow_{\beta} \lambda x. \lambda y. (\lambda w. \lambda z. w) \ y \ x
\]
\[
\rightarrow_{\beta} \lambda x. \lambda y. (\lambda z. y) \ x
\]
\[
\rightarrow_{\beta} \lambda x. \lambda y. y
\]

\[(\lambda b. (\lambda x. (\lambda y. ((b \ y) \ x))))(\lambda w. (\lambda z. w))\]
\[
\rightarrow_{\beta} \lambda x. (\lambda y. ((\lambda w. (\lambda z. w)) \ y) \ x))
\]
\[
\rightarrow_{\beta} \lambda x. (\lambda y. ((\lambda z. y) \ x))
\]
\[
\rightarrow_{\beta} \lambda x. (\lambda y. y)
\]
Factorial Revisited

FACT := \( \lambda f.\lambda n. \left(\begin{array}{l}
(iszero n) \ 1 \\
\text{ times } n \ (f \ (pred \ n))
\end{array}\right) \)

\( f_0 := \lambda n. 0 \)

\( f_1 := FACT(f_0) \)

\( \leftrightarrow_\beta^* \ \lambda n. \left(\begin{array}{l}
(iszero n) \ 1 \\
\text{ times } n \ (f_0 \ (pred \ n))
\end{array}\right) \)

\( f_2 := FACT(f_1) \)

\( \leftrightarrow_\beta^* \ \lambda n. \left(\begin{array}{l}
(iszero n) \ 1 \\
\text{ times } n \ (f_1 \ (pred \ n))
\end{array}\right) \)
Factorial Revisited

1. Every time we apply \texttt{FACT}, we get a better approximation to the factorial function.
2. If the argument to \texttt{FACT} was already the factorial function, we'd get the same thing back.
3. Thus the factorial function is a fixed point of \texttt{FACT}.
4. We have a $\lambda$-term $Y$ such that $Y(\texttt{FACT})$ is a fixed point of \texttt{FACT}.
5. Hence $Y(\texttt{FACT})$ is the factorial function. (!?)
Factorial Revisited

\[ \text{FACT} := \lambda f. \lambda n. (\text{iszero } n) 1 \quad (\text{times } n (f (\text{pred } n))) \]

\[ \text{fact} := \text{Y}(\text{FACT}) \]

\[ \text{fact}(N) \]
\[ = (\text{Y}(\text{FACT}))(N) \]
\[ \leftrightarrow^* \text{FACT} (\text{Y}(\text{FACT}))(N) \]
\[ = \text{FACT}(\text{fact})(N) \]
\[ \leftrightarrow^* (\text{iszero } N) 1 \]
\[ (\text{times } N (\text{fact} (\text{pred } N))) \]
Fibonacci

\[
FIB := \lambda f. \lambda n. \begin{cases} 
    0 & \text{if } \text{iszero } n \\
    1 & \text{if } \text{iszero } (\text{pred } n) \\
    \text{plus } (f \ (\text{pred } n)) & \text{otherwise} \\
\end{cases}
\]

\[
fib := Y(FIB)
\]
Fixed Points

• Every term has at least one fixed point.
  - Even `succe` and `not`.

• Some terms have many fixed points
  - For example, `\lambda n. n`
  - Or, `FIB`
  • Recall the `fib` and `intfib` functions from Assignment 1!
  - `y` picks out the unique least fixed point.
    • Which turns out to be the one we expect.
Confluence

**Theorem**

If $M \rightarrow^* M_1$ and $M \rightarrow^* M_2$ then there exists $N$ such that $M_1 \rightarrow^* N$ and $M_2 \rightarrow^* N$. 
β-Normal Forms

Definitions

A term $M$ is said to be β-normal (or to be a β-normal form) if there is no $N$ such that $M \rightarrow_{\beta} N$.

For example, $\lambda n.n$ or $x(\lambda y.z)$ are β-normal.

If $M \rightarrow_{\beta}^* N$ and $N$ is β-normal then we say that $N$ is a β-normal form of $M$.

Not every term has a normal form: $(\lambda x.xx) (\lambda x.xx)$
Corollary
A term has at most one $\beta$-normal form.

Proof?
Church-Rosser Property

**Theorem**

\[ M_1 \leftrightarrow_{\beta^*} M_2 \text{ if and only if there exists } N \text{ such that } M_1 \rightarrow_{\beta^*} N \text{ and } M_2 \rightarrow_{\beta^*} N. \]

If: By definition of conversion.

Only if: By induction on the proof that \( M_1 \leftrightarrow_{\beta^*} M_2 \)
Consistency

Corollary

• There are terms that are not convertible.
• A term might be convertible to $tt$ or $ff$ but not both.
• A term is convertible to at most one Church numeral.
This slide intentionally left blank
Reduction Strategies

• Depending on choice of reductions, may or may not reach a normal form.

\[(\lambda x.0)\ ( (\lambda x.xx)\ (\lambda x.xx) ) \rightarrow_\beta\ 0\]

\[(\lambda x.0)\ ( (\lambda x.xx)\ (\lambda x.xx) )\]

\[\rightarrow_\beta\ (\lambda x.0)\ ( (\lambda x.xx)\ (\lambda x.xx) )\]

\[\rightarrow_\beta\ (\lambda x.0)\ ( (\lambda x.xx)\ (\lambda x.xx) )\]

\[\rightarrow_\beta\ ...\]
Reduction Strategies

• In general, undecidable whether a term has a normal form.
• However, there is a semi-decision procedure
  – Method which (eventually) finds a normal form if one exists.
  – Never terminates otherwise.
Call-by-Name

• Also known as
  - Normal-order reduction
  - Leftmost reduction.

• Rule: always reduce "leftmost" application.

\[
(\lambda x.0) \ ((\lambda x.xx) \ (\lambda x.xx))
\]

______________________________ (leftmost)

• Guaranteed to find a normal form if one exists.
Example

• Let $I := \lambda x.x$

• Reduce $(\lambda y. yyy)(II)$ via call-by-name
Call-by-Value

• Also known as
  - Applicative reduction

• Rule: apply $\beta$ only when argument is a value.
  \[
  (\lambda x.0) \ ((\lambda x.xx) (\lambda x.xx)) \\
  \]  
  (non-value argument)

• Not guaranteed to find normal forms
  - But often more efficient than call-by-name
  - If normal form reached, same as call-by-name.
Example

• Let \( I := \lambda x.x \)
• Reduce \( (\lambda y. yyy)(II) \) via call-by-value
Call-by-Need

• Also known as
  - Lazy reduction
• Cannot be formalized in the framework we have, but easy to describe:
  - Like Call-by-Name (don't evaluate function arguments until actually used by the function.)
  - But, remember the argument's result and re-use.
  - If there are no side-effects, indistinguishable from call-by-name.
Reduction Order in PLs

• Call-by-value
  - FORTRAN, LISP, C, Java, ML, ...

• Call-by-name
  - Algol 60

• Call-by-need
  - Miranda, Gofer, Haskell
Part 2

Combinatory Logic
Syntax

• Pure Combinatory Logic

\[
a, b, c, d ::= \begin{align*}
K & \quad \text{a constant} \\
S & \quad \text{another constant} \\
a \ b & \quad \text{application}
\end{align*}
\]

• That's it!
  - Typical term: \( K (S K K) K S \)
One-step Reduction

• The relation $\rightarrow_{CL}$ is defined by:

\[
\begin{align*}
(K \ a) \ b & \rightarrow_{CL} a \\
((S \ a) \ b) \ c & \rightarrow_{CL} (a \ c) \ (b \ c)
\end{align*}
\]
One-step Reduction

• Using the left-associativity of application

\[ \text{K a b} \rightarrow_{\text{CL}} a \]

\[ \text{S a b c} \rightarrow_{\text{CL}} (a c) (b c) \]

\[ \text{a} \rightarrow_{\text{CL}} a' \]

\[ \text{a b} \rightarrow_{\text{CL}} a' b \]

\[ \text{b} \rightarrow_{\text{CL}} b' \]

\[ \text{a b} \rightarrow_{\text{CL}} a b' \]
Correspondence with $\lambda$-Calculus

\[
\begin{align*}
K \ a \ b & \rightarrow_{CL} a \\
S \ a \ b \ c & \rightarrow_{CL} (a \ c) \ (b \ c)
\end{align*}
\]

\[
K \ \approx \ \lambda x. \lambda y. \ x \\
= \ \lambda x. \ (\lambda y. \ x)
\]

\[
S \ \approx \ \lambda x. \lambda y. \lambda z. \ (xz) \ (yz) \\
= \ \lambda x. \ (\lambda y. \ (\lambda z. \ ((xz) \ (yz))))
\]
Exercises

1. What does \texttt{skks} reduce to?

2. And \texttt{s(kk)s}?

3. How about \texttt{skka}?

4. Put \texttt{i := skk}. How does \texttt{sii(sii)} reduce?
Combinary Completeness

- Claim: For every $\lambda$-term, there are terms in combinatory logic with the "same meaning"
  - For example, $\text{SKK}$ acts like the identity function:
    \[
    \text{SKKa} \rightarrow_{\text{CL}}^* a
    \]
    \[
    \text{SII} = S(SKK)(SKK) \text{ acts like } \lambda x.xx
    \]
    \[
    (S(SKK)(SKK))a \rightarrow_{\text{CL}}^* aa
    \]
- Thus combinatory logic is as powerful as the $\lambda$-calculus
Extending CL with variables

\[ a, b, c, d ::= \ x \ | \ y \ | \ldots \ \text{variables} \]
\[ \ | \ K \ \text{a constant} \]
\[ \ | \ S \ \text{another constant} \]
\[ \ | \ a \ b \ \text{application} \]

- Typical term: \[ K(\text{SK}xK)K\text{y}S \]
- No bound variables
  - All variables are free
  - Substitution is really easy
- Evaluation rules unchanged.
Bracket Abstraction

• For every extended-CL term \( a \) and every variable \( x \), there is an extended-CL term \( [x]a \) such that

1. \( x \) is not free in \( [x]a \).

2. \( ([x]a)b \rightarrow^{CL}* a[x\rightarrow b] \)

• For example, \( ([x]xx)(SK) \rightarrow^{CL}* (SK)(SK) \)
Bracket Abstraction

\[[x]K = \]
\[[x]S = \]
\[[x]x = \]
\[[x]y = \]
\[[x](ab) = \] \quad (x \neq y)
Examples

• \([x](xx) =\]

• \([x](SKx) =\]
Combinatory Completeness

• We can then translate every $\lambda$-term into an equivalent extended CL-term.

\[
    \begin{align*}
    \text{CL}(x) & := x \\
    \text{CL}(\lambda x. e) & := [x](\text{CL}(e)) \\
    \text{CL}(e_1 e_2) & := (\text{CL}(e_1))(\text{CL}(e_2))
    \end{align*}
\]

• Every closed $\lambda$-term translates into a variable-free CL-term.
Examples

\[ CL(\lambda x. \lambda y. x) = \]

\[ CL(\lambda x. \lambda y. y) = \]