**β-Reduction and Combinatory Logic**

April 9, 2001
CS 131: Programming Languages

---

**Review: Conversion**

- Notion of program equivalence

\[ \frac{M_1 \rightarrow^\beta M_2}{M_1 \leftrightarrow^* M_2} \]

\[ \frac{M_2 \leftrightarrow^* M_1}{M_1 \leftrightarrow^* M_2} \]

---

**Review: Fixed Points**

1. For every λ-calculus term \( M \), there exists \( N \) such that \( M(N) \leftrightarrow^* N \)

2. A term \( N \) with this property is called a fixed point of \( M \).

3. Fixed points can be found uniformly. That is, there is a term \( Y \) such that \( M(Y(M)) \leftrightarrow^* Y(M) \)

Namely, \( Y := \lambda f. (\lambda x. f(x)) (\lambda x. f(x)) \)

---

**Review: Fixed Points**

\[
\begin{align*}
F & := \lambda f. \lambda n. (\text{iszero } n) '\text{1}' \\
& \quad \text{(times } n \ (f \ (\text{pred } n))) \\
\text{fact} & := Y \ F \\
\text{fact} ('n') & := \rightarrow^* (F(\text{fact})) ('n') \\
& \quad \rightarrow^* (\text{iszero } 'n') '\text{1}' \\
& \quad \text{(times } 'n' \ (\text{fact } (\text{pred } 'n'))) 
\end{align*}
\]
Review: Factorial

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

\[ f_0 := \text{...whatever you want...} \]

\[ f_1 := F(f_0) \leftrightarrow \beta^* \lambda n. (\text{iszero } n) \cdot 1 \cdot (\text{times } n \ (f_0 \ (\text{pred } n))) \]

\[ f_2 := F(f_1) \leftrightarrow \beta^* \lambda n. (\text{iszero } n) \cdot 1 \cdot (\text{times } n \ (f_1 \ (\text{pred } n))) \]

Review: Factorial

- Every time we apply \( F \), we get a better approximation to the factorial function.
- So to find \( n! \) all we need to do is compute
  \[ F \ (F \ (F \ (\ldots \ (F \ f_0) \ldots)))) \]
  where there are \( n \) applications of \( F \); that is,
  \[ (F^n(f_0))(n!) \]
- But
  \[ \text{fact} = Y \ F \leftrightarrow \beta^* \ F \ (Y \ F) \leftrightarrow \beta^* \ F \ (F \ (Y \ F)) \leftrightarrow \beta^* \ldots \leftrightarrow \beta^* \ (F^{n+1}(Y \ F)). \]

Confluence

Theorem:

If \( M \rightarrow^* M_1 \) and \( M \rightarrow^* M_2 \) then there exists \( N \) such that \( M_1 \rightarrow^* N \) and \( M_2 \rightarrow^* N \).

\[ \begin{array}{c}
M \\
\downarrow^* \\
M_1 \\
\downarrow^* \\
N \\
\end{array} \]

\[ \begin{array}{c}
M \\
\downarrow^* \\
M_2 \\
\downarrow^* \\
N \\
\end{array} \]

\[ \begin{array}{c}
M \rightarrow^* \\
M_1 \rightarrow^* \\
\bullet \rightarrow^* \\
M_2 \rightarrow^* \\
N \rightarrow^* \\
\end{array} \]

\beta\text{-Normal Forms}

- Definitions
  - A term \( M \) is said to be \( \beta\text{-normal} \) (or to be a \( \beta\text{-normal form} \) if there is no \( N \) such that \( M \rightarrow N \).
    - For example, \( \lambda n. n \) or \( \lambda x. x \) are \( \beta\text{-normal} \).
    - If \( M \rightarrow^* N \) and \( N \) is \( \beta\text{-normal} \) then we say that \( N \) is a \( \beta\text{-normal form} \) of \( M \).
- Not every term has a normal form
  \[ (\lambda x. xx) \ (\lambda x. xx) \]
- Lemma:
  - A term has at most one \( \beta\text{-normal form} \).
  - 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

Corollaries:

1. There are terms that are not convertible.
2. A term might be convertible to \( \texttt{tt} \) or \( \texttt{ff} \) but not both.
3. A term is convertible to at most one Church numeral.

Reduction Strategies

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

\[
(\lambda x.x) (\lambda x.(\lambda x.x) (\lambda x.x)) \rightarrow^\beta x \\
(\lambda x.x) (\lambda x.(\lambda x.x) (\lambda x.x)) \rightarrow^\beta \lambda x.x (\lambda x.x) (\lambda x.x) \rightarrow^\beta \lambda x.x \rightarrow^\beta \ldots
\]
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.

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

\[\quad\quad\quad\quad\quad\quad\text{(leftmost)}\]

- 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.

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

\[\quad\quad\quad\quad\quad\quad\text{(non-value argument)}\]

- 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 \((\lambda y. y y y)(II)\) 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

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

• That's it!
  - Random term: \[ K\ (SKK)\ KS \]

\[ = \ ((K\ ((SK)\ K))\ K)\ S \]

One-step Reduction

• The relation \( \rightarrow_{\text{cl}} \) is defined by:

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

\[ \begin{align*}
 a & \rightarrow_{\text{cl}} a' \\
b & \rightarrow_{\text{cl}} b'
\end{align*} \]

\[ \begin{align*}
 a\ b & \rightarrow_{\text{cl}} a'\ b' \\
 a\ b & \rightarrow_{\text{cl}} a\ b'
\end{align*} \]

One-step Reduction

• Using the left-associativity of application

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

\[ \begin{align*}
 a & \rightarrow_{\text{cl}} a' \\
b & \rightarrow_{\text{cl}} b'
\end{align*} \]

\[ \begin{align*}
 a\ b & \rightarrow_{\text{cl}} a'\ b' \\
 a\ b & \rightarrow_{\text{cl}} a\ b'
\end{align*} \]

Correspondence with \( \lambda \)-Calculus

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

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

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

1. What does $SKKS$ reduce to?
2. And $S(KK)S$ ?
3. How about $SKKa$?
4. Put $I := SKK$. How does $SII(SII)$ reduce?

Combinatory Completeness

- Claim: For every $\lambda$-term, there are terms in combinatory logic with the "same meaning"
  - For example, $SKK$ acts like the identity function:
    $$SKKa \rightarrow_{cl} a$$
  - $SII = S(SKK)(SKK)$ acts like $\lambda x.xx$
    $$(S(SKK)(SKK))a \rightarrow_{cl} aa$$
- Thus combinatory logic is as powerful as the $\lambda$-calculus...even though there are no variables!

Extending CL with variables

- $a, b, c, d ::= x \mid y \mid \ldots$ variables
  - $K$ a constant
  - $S$ another constant
  - $a \mid b$ application
- Typical term: $K(SKxK)KyS$
- 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

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

Examples

\[
[x](xx) = \\
[x](SKx) = 
\]

Combinatory Completeness

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

\[
CL(x) := x \\
CL(\lambda x. e) := [x](CL(e)) \\
CL(e_1 e_2) := (CL(e_1))(CL(e_2))
\]

• Every closed \(\lambda\)-term translates into a variable-free CL-term.

Examples

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

- David Turner (1979):
  - Compile programs into combinatory logic
  - In practice, extend S and K with combinators like + and eq and cond, numeric constants, Y and I, etc.

  \[
  \text{fact} = S(S(K\text{cond}) (S(S(K\text{eq})(K\ 0))\ I)) (K\ 1) (S(S(K\text{times})\ I) (S(K\text{fact}) (S(S(K\text{minus})\ I) (K\ 1))))
  \]

Graph Reduction

- Nice implementation of call-by-need (lazy evaluation)
  - Evaluate each expression at most once

- Represent terms as graphs instead of trees
  - Overwrite sub-graphs with their values
  - Expresses sharing of delayed computations
    - As soon as it's evaluated once, everyone referring to this computation sees the resulting value.

Claimed Advantages

- Resulting program has no variables
  - Don't have to worry about substitution or environments

- Very simple execution strategy
  - Just a handful of combinators

- Could even implement S and K in hardware
  - e.g., SKIM

- Parallel graph reduction easy
  - Processors work on disjoint parts of graphs

Problems

- CL(\lambda x.\lambda y.\lambda z.(xz)(yz)) =

  \[
  S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(KK) (SKK))))))) (S(KK) (K(SKK))))))) (S(S(KS) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KS))) (S(S(KS) (S(KK) (KS))) (S(S(KK) (SKK))))))) (S(KK) (K(SKK)))))))
  \]

- A better translation would be?

- In general, translation can cause exponential blowup.
**Improvements**

1. Add new combinator constants
   - \( I \ a \rightarrow_{\phi} a \)
   - \( B \ a \ b \ c \rightarrow_{\psi} a \ (b \ c) \)
   - \( C \ a \ b \ c \rightarrow_{\chi} (a \ c) \ b \)

2. Improve the translation: \([x]x = I\)

3. Apply optimizations to the output
   - \( S \ (K \ a) \ (K \ b) = K \ (a \ b)\)
   - \( S \ (K \ a) \ I = a\)
   - \( S \ (K \ a) \ b = B \ a \ b\)
   - \( S \ a \ (K \ b) = C \ a \ b\)

**Other Improvements**

- More complex primitive combinators
- Program-specific combinators
  - Any closed lambda term can be made into a new constant
- Avoid graph updates of unshared terms