β-Reduction and Combinatory Logic

April 9, 2001

CS 131: Programming Languages

Review: Conversion

- Notion of program equivalence

$$\frac{M_1 \to_\beta M_2}{M_1 \leftrightarrow_\beta^* M_2}$$

$$\frac{}{M \leftrightarrow_\beta^* M} \qquad \frac{M_2 \leftrightarrow_\beta^* M_1}{M_1 \leftrightarrow_\beta^* M_2}$$

$$\frac{M_1 \leftrightarrow_\beta^* M_2 \qquad M_2 \leftrightarrow_\beta^* M_3}{M_1 \leftrightarrow_\beta^* M_3}$$

Review: Fixed Points

1. For *every* λ-calculus term `M`, there exists `N` such that

   $$M(N) \leftrightarrow_\beta^* N$$

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

   $$M(\mathbf{Y}(M)) \leftrightarrow_\beta^* \mathbf{Y}(M)$$

   Namely,

   $$\mathbf{Y} := \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Review: Fixed Points

```
F    :=  λf.λn.(iszero n) ⌜1⌝
                   (times n (f (pred n)))
fact := Y F

fact(⌜n⌝)
  ↔β* (F(fact))(⌜n⌝)
  ↔β* (iszero ⌜n⌝) ⌜1⌝
          (times ⌜n⌝ (fact (pred ⌜n⌝)))
```

Review: Factorial

F := λf.λn.(**iszero** n) ⌜1⌝
(**times** n (f (**pred** n)))

f_0 := *...whatever you want...*

f_1 := **F**(f_0)

$\leftrightarrow_\beta^*$ λn.(**iszero** n) ⌜1⌝
(**times** n (f_0 (**pred** n)))

f_2 := **F**(f_1)

$\leftrightarrow_\beta^*$ λn.(**iszero** n) ⌜1⌝
(**times** n (f_1 (**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**(**F**(...(**F** f_0)...)))) ⌜n⌝

  where there are n+1 applications of **F**; that is,

  ($\mathbf{F}^{n+1}$(f_0))(⌜n⌝)
- But

  fact = **Y F** $\leftrightarrow_\beta^*$ **F**(**Y F**) $\leftrightarrow_\beta^*$ **F**(**F**(**Y F**))
  $\leftrightarrow_\beta^*$ ... $\leftrightarrow_\beta^*$ ($\mathbf{F}^{n+1}$(**Y F**)).

Confluence

Theorem:

If M $\rightarrow_\beta^*$ M_1 and M $\rightarrow_\beta^*$ M_2 then there exists N such that $M_1 \rightarrow_\beta^*$ N and $M_2 \rightarrow_\beta^*$ N.

$$\begin{tikzcd}
 & M \arrow[dl, "*"'] \arrow[dr, "*"] & \\
M_1 \arrow[dr, dotted, "*"'] & & M_2 \arrow[dl, dotted, "*"] \\
 & N &
\end{tikzcd}$$

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, λn.n or x(λy.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

  (λx.xx)(λx.xx)
- Lemma:
  - A term has at most one β-normal form.
  - Proof?

Church-Rosser Property

Theorem

$M_1 \leftrightarrow_\beta^* M_2$ if and only if there exists N such that $M_1 \to_\beta^* N$ and $M_2 \to_\beta^* N$.

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 **`tt`** or **`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.

$$(\lambda x.\ulcorner 0 \urcorner)((\lambda x.xx)(\lambda x.xx)) \to_\beta \ulcorner 0 \urcorner$$

$$\begin{aligned}
&(\lambda x.\ulcorner 0 \urcorner)((\lambda x.xx)(\lambda x.xx)) \\
&\quad \to_\beta (\lambda x.\ulcorner 0 \urcorner)((\lambda x.xx)(\lambda x.xx)) \\
&\quad \to_\beta (\lambda x.\ulcorner 0 \urcorner)((\lambda x.xx)(\lambda x.xx)) \\
&\quad \to_\beta \ldots
\end{aligned}$$

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.

$$\underbrace{(\lambda x.\ulcorner 0 \urcorner)(\underline{(\lambda x.xx)(\lambda x.xx)})}_{\text{(leftmost)}}$$

- Guaranteed to find a normal form if one exists.

Example

- Let `I := λx.x`
- Reduce `(λy.yyy)(II)` via call-by-name

Call-by-Value

- Also known as
  - Applicative reduction
- Rule: apply β only when argument is a value.

$$\underline{(\lambda x.\ulcorner 0 \urcorner)(\underbrace{(\lambda x.xx)(\lambda x.xx)}_{\text{(non-value argument)}})}$$

- 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 := λx.x`
- Reduce `(λy.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 ::=  K          a constant
          |  S          another constant
          |  a b        application
```

- That's it!
  - Random term: `K(SKK)KS`
    `= ((K((SK)K))K)S`

One-step Reduction

- The relation $\to_{CL}$ is defined by:

$$\frac{}{(\mathtt{K}\ a)\ b \to_{CL} a}$$

$$\frac{}{((\mathtt{S}\ a)\ b)\ c \to_{CL} (a\ c)(b\ c)}$$

$$\frac{a \to_{CL} a'}{a\ b \to_{CL} a'\ b} \qquad \frac{b \to_{CL} b'}{a\ b \to_{CL} a\ b'}$$

One-step Reduction

- Using the left-associativity of application

$$\frac{}{\mathtt{K}\ a\ b \to_{CL} a}$$

$$\frac{}{\mathtt{S}\ a\ b\ c \to_{CL} (a\ c)(b\ c)}$$

$$\frac{a \to_{CL} a'}{a\ b \to_{CL} a'\ b} \qquad \frac{b \to_{CL} b'}{a\ b \to_{CL} a\ b'}$$

Correspondence with λ-Calculus

$$\frac{}{\mathtt{K}\ a\ b \to_{CL} a}$$

$$\frac{}{\mathtt{S}\ a\ b\ c \to_{CL} (a\ c)(b\ c)}$$

$$\mathtt{K} \approx \lambda x.\lambda y.x$$
$$= \lambda x.(\lambda y.x)$$

$$\mathtt{S} \approx \lambda x.\lambda y.\lambda z.(xz)(yz)$$
$$= \lambda x.(\lambda y.(\lambda z.((xz)(yz))))$$

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 λ-term, there are terms in combinatory logic with the "same meaning"
  - For example, `SKK` acts like the identity function:

    $\texttt{SKKa} \rightarrow_{CL}^{*} \texttt{a}$
  - `SII = S(SKK)(SKK)` acts like `λx.xx`

    $\texttt{(S(SKK)(SKK))a} \rightarrow_{CL}^{*} \texttt{aa}$
- Thus combinatory logic is as powerful as the λ-calculus...even though there are no variables!

Extending CL with variables

`a,b,c,d ::=`	`x | y | ...`	*variables*
`|`	`K`	*a constant*
`|`	`S`	*another constant*
`|`	`a 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. $\texttt{([x]a)b} \rightarrow_{CL}^{*} \texttt{a[x}\rightarrow\texttt{b]}$

- For example, $\texttt{([x]xx)(SK)} \rightarrow_{CL}^{*} \texttt{(SK)(SK)}$

Bracket Abstraction

[x]K =
[x]S =
[x]x =
[x]y = (x≠y)
[x](ab) =

Examples

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

Combinatory Completeness

- We can then translate every λ-term into an equivalent extended CL-term.

CL(x) := x
CL(λx.e) := [x](CL(e))
CL(e_1e_2) := (CL(e_1))(CL(e_2))

- Every *closed* λ-term translates into a variable-free CL-term.

Examples

CL(λx.λy.x) =

CL(λx.λy.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.

```
fact = S (S (S (K cond) (S (S (K eq) (K 0)) I))
         (K 1)) (S (S (K times) I) (S (K fact)
         (S (S (K 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(λx.λy.λz.(xz)(yz)) =`

```
S(S(KS)(S(S(KS)(S(KK)(KS)))(S(S(KS)(S(S(KS)(S(K
K)(KS)))(S(S(KS)(S(KK)(KK)))(S(KK)(SKK)))))(S(K
K)(K(SKK))))))(S(S(KS)(S(S(KS)(S(KK)(KS)))(S(S(
KS)(S(KK)(KK)))(K(SKK)))))(S(KK)(K(SKK))))
```

- A better translation would be?
- In general, translation can cause exponential blowup.

Improvements

1. Add new combinator constants

$$\begin{aligned} \mathtt{I\ a} &\rightarrow_{CL} \mathtt{a} \\ \mathtt{B\ a\ b\ c} &\rightarrow_{CL} \mathtt{a\ (b\ c)} \\ \mathtt{C\ a\ b\ c} &\rightarrow_{CL} \mathtt{(a\ c)\ b} \end{aligned}$$

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