Computer Science 131
Programming Languages

November 14, 2000

b-reduction and Combinators

Syntax

```
M,N ::=  x              variables
      |  λx.M           functions
      |  M N            applications
```

- Application associates leftward

```
xyzw   =   ((xy)z)w
```

- Function bodies as as large as possible.

```
λx.λy.fyx = λx.(λy.fyx) = λx.(λy.(fyx))
          = λx.(λy.((fy)x))
λx.λy.f(yx) = λx.(λy.(f(yx)))
λx.(λy.fy)x = λx.((λy.(fy))x)
```

Review

- The relation $\rightarrow_b$ is defined by:

$$\frac{}{\mathtt{(l x.M)N} \;\rightarrow_b\; \mathtt{M[x \rightarrow N]}}$$

$$\frac{\mathtt{M} \rightarrow_b \mathtt{M'}}{\mathtt{M\ N} \rightarrow_b \mathtt{M'\ N}} \qquad \frac{\mathtt{N} \rightarrow_b \mathtt{N'}}{\mathtt{M\ N} \rightarrow_b \mathtt{M\ N'}}$$

$$\frac{\mathtt{M} \rightarrow_b \mathtt{M'}}{\mathtt{l x.M} \rightarrow_b \mathtt{l x.M'}}$$

Review

$$\underline{(\lambda b.\lambda x.\lambda y.b\ y\ x)(\lambda w.\lambda z.w)} \qquad \textit{(not tt)}$$

$$\to_\beta\ \lambda x.\lambda y.\underline{(\lambda w.\lambda z.w)\ y}\ x$$

$$\to_\beta\ \lambda x.\lambda y.\underline{(\lambda z.y)\ x}$$

$$\to_\beta\ \lambda x.\lambda y.y$$

$$\underline{(\lambda b.(\lambda x.(\lambda y.((b\ y)\ x))))(\lambda w.(\lambda z.w))}$$

$$\to_\beta \lambda x.(\lambda y.((\underline{(\lambda w.(\lambda z.w))\ y})\ x))$$

$$\to_\beta \lambda x.(\lambda y.(\underline{(\lambda z.y)\ x}))$$

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

Factorial Revisited

FACT := lf.ln.(iszero n) <u>1</u>
(times n (f (pred n)))

f_0 := ln.<u>0</u>

f_1 := FACT(f_0)

$\leftrightarrow_b^*$ **ln.(iszero n) <u>1</u>**
(times n (f_0 (pred n)))

f_2 := FACT(f_1)

$\leftrightarrow_b^*$ **ln.(iszero n) <u>1</u>**
(times n (f_1 (pred n)))

Factorial Revisited

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

Factorial Revisited

```
FACT :=  lf.ln.(iszero n) 1
                      (times n (f (pred n)))
fact := Y(FACT)

fact(N)
  = (Y(FACT))(N)
```

$\leftrightarrow_b^*$ **FACT(Y(FACT))(N)**

= FACT(fact)(N)

$\leftrightarrow_b^*$ **(iszero N) <u>1</u>**
(times N (fact (pred N)))

Fibonacci

```
FIB :=  lf.ln.(iszero n) 0
               (iszero (pred n) 1
                 (plus (f (pred n))
                       (f (pred (pred n)))))
fib := Y(FIB)
```

Fixed Points

- Every term has at least one fixed point.
  - Even **succ** and **not**.
- Some terms have many fixed points
  - For example, l**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 $\mathbf{M} \to_b^* \mathbf{M_1}$ and $\mathbf{M} \to_b^* \mathbf{M_2}$ then there exists $\mathbf{N}$ such that $\mathbf{M_1} \to_b^* \mathbf{N}$ and $\mathbf{M_2} \to_b^* \mathbf{N}$.

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

β-Normal Forms

Definitions

A term **M** is said to be *b-normal* (or to be a β-normal form) if there is no **N** such that **M** $\rightarrow_b$ **N**.

For example, l**n.n** or **x(**l**y.z)** are β-normal.

If **M** $\rightarrow_b^*$ **N** and **N** is β-normal then we say that **N** is a *b-normal form of* **M**.

Not every term has a normal form: **(**l**x.xx)(**l**x.xx)**

β-Normal Forms

Corollary

A term has at most one β-normal form.

Proof?

Church-Rosser Property

Theorem

$\mathbf{M_1} \leftrightarrow_b^* \mathbf{M_2}$ if and only if there exists $\mathbf{N}$ such that $\mathbf{M_1} \rightarrow_b^* \mathbf{N}$ and $\mathbf{M_2} \rightarrow_b^* \mathbf{N}$.

If: By definition of conversion.

Only if: By induction on the proof that $\mathbf{M_1} \leftrightarrow_b^* \mathbf{M_2}$

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.

$$\mathbf{(l x.\underline{0})((l x.xx)(l x.xx))} \to_b \mathbf{\underline{0}}$$

$$\begin{aligned} &\mathbf{(l x.\underline{0})((l x.xx)(l x.xx))} \\ &\quad \to_b \mathbf{(l x.\underline{0})((l x.xx)(l x.xx))} \\ &\quad \to_b \mathbf{(l x.\underline{0})((l x.xx)(l x.xx))} \\ &\quad \to_b \ldots \end{aligned}$$

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.

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

- Guaranteed to find a normal form if one exists.

Example

- Let **`I := `**λ**`x.x`**
- Reduce **`(`**λ**`y.yyy)(II)`** via call-by-name

Call-by-Value

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

(λx.0)((λx.xx)(λx.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 := 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!
  - Typical term: **K(SKK)KS**

One-step Reduction

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

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

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

$$\frac{\mathtt{a} \rightarrow_{CL} \mathtt{a'}}{\mathtt{a\ b} \rightarrow_{CL} \mathtt{a'\ b}} \qquad \frac{\mathtt{b} \rightarrow_{CL} \mathtt{b'}}{\mathtt{a\ b} \rightarrow_{CL} \mathtt{a\ b'}}$$

One-step Reduction

- Using the left-associativity of application

$$\frac{}{\mathbf{K\ a\ b} \rightarrow_{\mathrm{CL}} \mathbf{a}}$$

$$\frac{}{\mathbf{S\ a\ b\ c} \rightarrow_{\mathrm{CL}} \mathbf{(a\ c)(b\ c)}}$$

$$\frac{\mathbf{a} \rightarrow_{\mathrm{CL}} \mathbf{a'}}{\mathbf{a\ b} \rightarrow_{\mathrm{CL}} \mathbf{a'\ b}} \qquad \frac{\mathbf{b} \rightarrow_{\mathrm{CL}} \mathbf{b'}}{\mathbf{a\ b} \rightarrow_{\mathrm{CL}} \mathbf{a\ b'}}$$

Correspondence with l-Calculus

$$\frac{}{\mathtt{K\ a\ b} \rightarrow_{\mathrm{CL}} \mathtt{a}}$$

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

K ≈ l**x.**l**y.x**

= l**x.(**l**y.x)**

S ≈ l**x.**l**y.**l**z.(xz)(yz)**

= l**x.(**l**y.(**l**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 l-term, there are terms in combinatory logic with the "same meaning"
  - For example, **SKK** acts like the identity function:

$$\mathbf{SKKa} \rightarrow_{CL}^{*} \mathbf{a}$$

  - **SII = S(SKK)(SKK)** acts like l**x.xx**

$$\mathbf{(S(SKK)(SKK))a} \rightarrow_{CL}^{*} \mathbf{aa}$$

- Thus combinatory logic is as powerful as the l-calculus

This slide intentionally left blank

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. $\mathbf{([x]a)b} \to_{CL}^{*} \mathbf{a[x \to b]}$

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

Bracket Abstraction

[x]K =

[x]S =

[x]x =

[x]y = **(x**^{1} **y)**

[x](ab) =

Examples

- **`[x](xx) =`**

- **`[x](SKx) =`**

Combinatory Completeness

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

$$
\begin{array}{lcl}
\mathbf{CL(x)} & := & \mathbf{x} \\
\mathbf{CL(\lambda x.e)} & := & \mathbf{[x](CL(e))} \\
\mathbf{CL(e_1 e_2)} & := & \mathbf{(CL(e_1))(CL(e_2))}
\end{array}
$$

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

Examples

CL(λx.λy.x) =

CL(λx.λy.y) =