Extending the λ-Calculus

October 2, 2001

CS 131: Programming Languages

Where We Are So Far

- Encodings of data (booleans, natural numbers, pairs, ...)
- Operations using these encodings

```
iszero := λx.(x (λy.ff) tt)
```

- Hopefully you should be convinced that in principle we can represent any sort of data structure and any program that operates with such structures.
  - Integers? Characters?
  - Lists? Strings? Trees?
  - Unicalc?

What's the Point of Conversion?

- The β-reduction step $(\lambda x.M)N \rightarrow_\beta M[x \rightarrow N]$ is intuitively a computation step; applying a function.
  - But what about β-expansion, $M[x \rightarrow N] \rightarrow_\beta (\lambda x.M)N$?
  - Or the conversion relation $\leftrightarrow_\beta^*$?

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

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

A Property of λ-Calculus

Theorem [Confluence]:

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 *normal* (or to be a *normal form*) if it cannot be further reduced.
    - e.g., `λn.n` and `x(λy.z)` are normal with respect to β-reduction
  - If $M \to_\beta^* N$ and `N` is normal then we say that `N` is *a 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 $M_1 \leftrightarrow_\beta^* M_2$

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.

```
(λx.⌜0⌝)((λx.xx)(λx.xx)) →β ⌜0⌝

(λx.⌜0⌝)((λx.xx)(λx.xx))
              →β (λx.⌜0⌝)((λx.xx)(λx.xx))
              →β (λx.⌜0⌝)((λx.xx)(λx.xx))
              →β …
```

- Theorem: If you always reduce the application whose λ is leftmost, you're guaranteed to reach a normal form as long as one exists. [*Normal order*]

λ-Calculus as a PL

- We can view the λ-calculus as a programming language, where we "execute" programs by applying β-*reductions*.
- Nearly every language makes some choice about the order in which computations occur.
  - Are function arguments evaluated before the function call?
  - Are subexpressions evaluated in a certain order?
  - Are subexpressions even guaranteed to be evaluated at all?
- Because this is a critical aspect of the language, I want to give a formal description.

Contexts

- We can divide up any piece of code into a (arbitrary) subexpression and everything else.

`(λx.xxx)((λy.y)(λz.z))   (λx.xxx)((λy.y)(λz.z))`

  - We call the "everything else" the *context* of the subexpression
  - We can write down a context by itself as a piece of code with a single "hole"

`(λx.xxx)((λy.•)(λz.z))   (λx.• x)((λy.y)(λz.z))`

Defining Evaluation

- One way to specify legal evaluation orders is to specify the contexts in which reduction is allowed
- These are called *evaluation contexts*

Call-By-Name

- Basic steps:

$\boxed{(\lambda x.M)N \;\rightarrow\; M[x \rightarrow N]}$

- Evaluation Contexts:

```
E ::=  •
    |  E M
```

Example CBN Evaluations

$(\lambda x.xxx)((\lambda y.y)(\lambda z.z)) \rightarrow$

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

Call-By-Value

- Basic steps: $\boxed{(\lambda x.M)V \;\rightarrow\; M[x \rightarrow V]}$
- Evaluation Contexts:

```
E ::=  •
    |  E M
    |  V E
```

where $V ::= \lambda x.M$

represents a value (expression which cannot be further reduced.

Example CBV Evaluations

$(\lambda x.xxx)((\lambda y.y)(\lambda z.z)) \rightarrow$

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

So What?

- It is incredibly cool that this tiny language is sufficient for any sort of data processing
  - In the same way that TM's are sufficient.
  - Combinatory logic is even simpler, BTW
- But it's not very practical as it stands
  - Recall the definition of predecessor

Extending the λ-Calculus

- Key observation
  - All we care about **tt**, **ff**, and **if** is that they have the "right" behavior.

if tt M N $\rightarrow_\beta^*$ M
if ff M N $\rightarrow_\beta^*$ N

  - Same with natural numbers:

succ ⌜*n*⌝ $\rightarrow_\beta^*$ ⌜*n+1*⌝
pred ⌜*n+1*⌝ $\rightarrow_\beta^*$ ⌜*n*⌝
iszero ⌜0⌝ $\rightarrow_\beta^*$ **tt**
iszero ⌜*n+1*⌝ $\rightarrow_\beta^*$ **ff**

Extending the λ-Calculus

- Same with pairs:

fst <M,N> $\rightarrow_\beta^*$ M
snd <M,N> $\rightarrow_\beta^*$ N

- Same with lists:

hd (**cons** M N) $\rightarrow_\beta^*$ M
tl (**cons** M N) $\rightarrow_\beta^*$ N
isnil nil $\rightarrow_\beta^*$ **tt**
isnil (**cons** M N) $\rightarrow_\beta^*$ **ff**

Extending the λ-Calculus

- So, why not just throw all these (definable) things in as new primitives?
  - Start with the unityped λ-Calculus
  - Throw in integers, booleans, pairs, lists, …
    - Formally redundant, but we can implement these primitives much more efficiently than using encodings.
    - We also need to throw in the appropriate computation steps, in addition to β-reduction
- When we do this, we get a *functional programming language*.

Adding Basic Arithmetic

- Syntax

```
M,N ::=  x, y, z, ...          variables
      |  λx.M                  functions
      |  M N                   applications
      |  0, 1, -1, ...         integer constants
      |  M + N                 additions
```

- Computation axioms

```
(λx.M)N  →  M[x→N]
  n + m  →  n⊕m
```

Sample Computations

(λx.x+x)(2) →

3+((λx.x+4)(-2)) →

(1+2)+(3+4) →

(λx.x+x)(2+2) →

Digression: Order of Evaluation

- Does it matter how we start this program?
  (λx.x+x)(2+2)
- No, because...
- Yes, because...

Call-By-Name with Arithmetic

- Basic steps:

```
(λx.M)N  →  M[x→N]
  n + m  →  n⊕m
```

- Evaluation Contexts:

```
E ::=  •
    |  E M
    |  E + M
    |  M + E
```

Call-By-Value with Arithmetic

- Basic steps:

```
(λx.M)N  →  M[x→N]
  n + m  →  n⊕m
```

- Evaluation Contexts:

```
E ::=  •
    |  E M
    |  V E
    |  E + M
    |  M + E
```

Reduction Order in PLs

- Call-by-value
  - Function arguments are evaluated before the function is called.
  - FORTRAN, LISP, C, Java, ML, …
- Call-by-name
  - Function arguments are evaluated only when they are used by the function (and every time they are used)
  - Algol 60
- Call-by-need
  - Function arguments are evaluated only when they are used by the function (and the result is cached)
  - Miranda, Gofer, Haskell

Mini-Scheme *Abstract* Syntax

```
M,N ::=  x, y, z, ...                  variables
      |  λx.M                          functions
      |  M N                           applications
      |  0, 1, -1, ...                 integer constants
      |  M + N                         additions
      |  tt | ff                       booleans
      |  iszero M                      zero-test
      |  if M then N_1 else N_2        conditionals
      |  <M,N>                         pairs
      |  fst M | snd M                 projections
      |  nil                           empty list
      |  isnil M                       empty-test
```

Mini-Scheme Values

```
V ::=
```

Mini-Scheme Primitive Steps

Mini-Scheme Evaluation Contexts