Extending the $\lambda$-Calculus

October 2, 2001
CS 131: Programming Languages

Where We Are So Far

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

\[
iszero := \lambda x. (x \ (\lambda y ff) \ tt)
\]

• 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 $\beta$-reduction step \((\lambda x. M) N \to_{\beta} M[x \to N]\) is intuitively a computation step; applying a function.
  - But what about $\beta$-expansion, \(M[x \to N] \to_{\beta} (\lambda x. M) N\) ?
  - Or the conversion relation $\leftrightarrow_{\beta}^*$ ?

<table>
<thead>
<tr>
<th>$M \leftrightarrow_{\beta}^* M$</th>
<th>$M_1 \to_{\beta}^* M_2$</th>
<th>$M_2 \leftrightarrow_{\beta}^* M_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_1 \leftrightarrow_{\beta}^* M_2$</td>
<td>$M_1 \leftrightarrow_{\beta}^* M_2$</td>
<td>$M_1 \leftrightarrow_{\beta}^* M_2$</td>
</tr>
</tbody>
</table>

A Property of $\lambda$-Calculus

Theorem [Confluence]:
If $M \to_{\beta}^* M_1$ and $M \to_{\beta}^* M_2$ then there exists $N$ such that $M_1 \to_{\beta}^* N$ and $M_2 \to_{\beta}^* N$.

\[
\begin{array}{c}
\quad \quad M \\
\quad M_1 \quad \quad M_2 \\
\quad M_1 \quad \quad M_2 \\
\quad M_1 \quad \quad M_2 \\
\quad M_1 \quad \quad M_2 \\
\end{array}
\]
**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., $\lambda n.n$ and $x(\lambda y.x)$ are normal with respect to $\beta$-reduction
  - If $M \rightarrow^* N$ and $N$ is normal then we say that $N$ is a normal form of $M$.
  - Not every term has a normal form: $(\lambda x.xx)(\lambda x.xx)$

- **Lemma:**
  - A term has at most one $\beta$-normal form.
  - Proof?

**Church-Rosser Property**

**Theorem**

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

*If:* By definition of conversion.

*Only if:* By induction on the proof that $M_1 \leftrightarrow \beta^* M_2$.

**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.

\[
\begin{align*}
\lambda x.0 \,(\lambda x.xx) \,(\lambda x.xx) & \rightarrow_\eta 0 \\
\lambda x.0 \,(\lambda x.xx) \,(\lambda x.xx) & \rightarrow_\eta (\lambda x.0) \,(\lambda x.xx) \,(\lambda x.xx) \\
& \rightarrow_\eta (\lambda x.0) \,(\lambda x.xx) \,(\lambda x.xx) \\
& \rightarrow_\eta \ldots
\end{align*}
\]

• Theorem: If you always reduce the application whose \(\lambda\) is leftmost, you're guaranteed to reach a normal form as long as one exists. [Normal order]

\[\lambda\text{-Calculus as a PL}\]

• We can view the \(\lambda\)-calculus as a programming language, where we "execute" programs by applying \(\beta\)-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.

\[\text{Contexts}\]

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

\[
\begin{align*}
(\lambda x.xxx) \,(\lambda y.y) \,(\lambda z.z) & \quad \quad (\lambda x.xxx) \,(\lambda y.y) \,(\lambda z.z) \\
\end{align*}
\]

  - 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"

\[
\begin{align*}
(\lambda x.xxx) \,(\lambda y.y) \,(\lambda z.z) & \quad \quad (\lambda x.xxx) \,(\lambda y.y) \,(\lambda z.z) \\
\end{align*}
\]

\[\text{Defining Evaluation}\]

• 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:

\[(\lambda x. M) N \rightarrow M[x\rightarrow N]\]

• Evaluation Contexts:

\[E ::= \bullet \mid E \ M\]

Example CBN Evaluations

\[(\lambda x.xxx)((\lambda y)(\lambda z)) \rightarrow\]

\[\lambda x.((\lambda y)(\lambda z)) \rightarrow\]

Call-By-Value

• Basic steps:

\[(\lambda x. M) V \rightarrow M[x\rightarrow V]\]

• Evaluation Contexts:

\[E ::= \bullet \mid E \ M \mid V \ E\]

where \[V ::= \lambda x. M\]
represents a value (expression which cannot be further reduced.

Example CBV Evaluations

\[(\lambda x.xxx)((\lambda y)(\lambda z)) \rightarrow\]

\[\lambda x.((\lambda y)(\lambda z)) \rightarrow\]
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.

\[
\begin{align*}
\text{if } tt \ M \ N & \rightarrow \alpha \ M \\
\text{if } ff \ M \ N & \rightarrow \alpha \ N
\end{align*}
\]

- Same with natural numbers:

\[
\begin{align*}
\text{succ } n & \rightarrow \alpha \ n+1 \\
\text{pred } n+1 & \rightarrow \alpha \ n \\
\text{iszero } 0 & \rightarrow \alpha \ tt \\
\text{iszero } n+1 & \rightarrow \alpha \ ff
\end{align*}
\]

Extending the λ-Calculus

- So, why not just throw all these (definable) things in as new primitives?
  - Start with the untyped 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 \( \beta \)-reduction
- When we do this, we get a functional programming language.
Adding Basic Arithmetic

• Syntax

\[ M, N ::= x, y, z, \ldots \] variables

\[ \lambda x. M \] functions

\[ M N \] applications

\[ 0, 1, -1, \ldots \] integer constants

\[ M + N \] additions

• Computation axioms

\[ (\lambda x. M) N \rightarrow M[x\rightarrow N] \]

\[ n + m \rightarrow n\oplus m \]

Sample Computations

\[ (\lambda x. x+x) (2) \rightarrow \]

\[ 3 + ((\lambda x. x+4) (-2)) \rightarrow \]

\[ (1+2) + (3+4) \rightarrow \]

\[ (\lambda x. x+x) (2+2) \rightarrow \]

Digression: Order of Evaluation

• Does it matter how we start this program?
  \[ (\lambda x. x+x) (2+2) \]

• No, because...

• Yes, because...
Call-By-Name with Arithmetic

• Basic steps:

\[(\lambda x.M) N \rightarrow M[x \rightarrow N]\]

\[n + m \rightarrow n \oplus m\]

• Evaluation Contexts:

\[
E ::= \\
| E \ M \\
| E + M \\
| M + E \\
\]

Call-By-Value with Arithmetic

• Basic steps:

\[(\lambda x.M) N \rightarrow M[x \rightarrow N]\]

\[n + m \rightarrow n \oplus m\]

• 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 N₁ else N₂ 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