Lambda Calculus 2

COS 441 Slides 14

read: 3.4, 5.1, 5.2, 3.5 Pierce
The lambda calculus is a language of pure functions

- expressions: \[ e ::= x \mid \lambda x.e \mid e_1 e_2 \]
- values: \[ v ::= \lambda x.e \]
- call-by-value operational semantics:

\[
\frac{\lambda\lambda x.e \, v \rightarrow e[v/x]}{(\beta)}
\]

\[
\frac{e_1 \rightarrow e_1'}{e_1 \, e_2 \rightarrow e_1' \, e_2} \quad \frac{e_2 \rightarrow e_2'}{v \, e_2 \rightarrow v \, e_2'}
\]

- example execution: \((\lambda x.x \, x) \, (\lambda y.y) \rightarrow (\lambda y.y) \, (\lambda y.y) \rightarrow \lambda y.y\)
ENCODING BOOLEANS
booleans

- the encoding:

\[\text{tru} = \lambda t. \lambda f. t\]

\[\text{fls} = \lambda t. \lambda f. f\]

\[\text{test} = \lambda x. \lambda \text{then}. \lambda \text{else}. x \text{ then else}\]
create a function "and" in the lambda calculus that mimics conjunction. It should have the following properties.

and tru tru -->* tru
and fls tru -->* fls
and tru fls -->* fls
and fls fls -->* fls
booleans

\text{tru} = \lambda t. f. t \quad \text{fls} = \lambda t. f. f

\text{and} = \lambda b. c. b \ c \ \text{fls}

\text{and} \ \text{tru} \ \text{tru}

\Rightarrow \ * \ \text{tru} \ \text{tru} \ \text{fls}

\Rightarrow \ * \ \text{tru}
booleans

tru = \t.\f. t  
fls = \t.\f. f

and = \b.\c. b c fls

and fls tru

-->* fls tru fls

-->* fls
booleans

tru = \t.\f. t
fls = \t.\f. f

and = \b.\c. b c fls

and fls tru
-->* fls tru fls
-->* fls

challenge: try to figure out how to implement "or" and "xor"
ENCODING PAIRS
pairs

• would like to encode the operations
  – create e1 e2
  – fst p
  – sec p

• pairs will be functions
  – when the function is used in the fst or sec operation it should reveal its first or second component respectively
pairs

create = \(x\).\(y\).\(b\) . \(b\)\(x\)\(y\)

fst = \(p\). \(p\) \(tru\)              \(tru\) = \(x\).\(y\).\(x\)

sec = \(p\). \(p\) \(fls\)              \(fls\) = \(x\).\(y\).\(y\)
create = \x.\y.\b.\ b \times \ y

fst = \p.\ p \ tru \quad tru = \x.\y.\ \ x
sec = \p.\ p \ fls \quad fls = \x.\y.\ y

fst (create \ tru \ fls)
= fst ((\x.\y.\b.\ b \times \ y) \ tru \ fls)
pairs

create = \x.\y.\b. b x y
fst = \p. p tru    tru = \x.\y.x
sec = \p. p fls    fls = \x.\y.y

fst (create tru fls)
= fst ((\x.\y.\b. b x y) tru fls)
-->* fst (\b. b tru fls)
pairs

create = \(x.\ y.\ b.\ b\ x\ y\)
fst = \(p.\ p\ \text{tru}\) \hspace{0.5cm} \text{tru} = \(x.\ y.\ x\)
sec = \(p.\ p\ \text{fls}\) \hspace{0.5cm} \text{fls} = \(x.\ y.\ y\)

\[
\text{fst (create tru fls)}
\]
\[
= \text{fst} ((x.\ y.\ b.\ b\ x\ y)\ \text{tru}\ \text{fls})
\]
\[
\rightarrow \ast \quad \text{fst} (b.\ b\ \text{tru}\ \text{fls})
\]
\[
= (p.\ p\ \text{tru}) (b.\ b\ \text{tru}\ \text{fls})
\]
pairs

create = \x.\y.\b. b x y
fst = \p. p tru
sec = \p. p fls
tru = \x.\y.x
fls = \x.\y.y

fst (create tru fls)
= fst ((\x.\y.\b. b x y) tru fls)
-->* fst (\b. b tru fls)
= (\p.p tru) (\b. b tru fls)
---> (\b. b tru fls) tru
create = \(x.\ y.\ b\ b\ x\ y\)

\(\text{fst} = p.\ p\ \text{tru} \quad \text{tru} = x.\ y.x\)

sec = \(p.\ p\ \text{fls} \quad \text{fls} = x.\ y.y\)

\(\text{fst}\ (\text{create}\ \text{tru}\ \text{fls})\)

= \(\text{fst}\ ((x.\ y.\ b\ b\ x\ y)\ \text{tru}\ \text{fls})\)

-->* \(\text{fst}\ (b.\ b\ \text{tru}\ \text{fls})\)

= \((p.p\ \text{tru})\ (b.\ b\ \text{tru}\ \text{fls})\)

--> \((b.\ b\ \text{tru}\ \text{fls})\ \text{tru}\)

--> \(\text{tru}\ \text{tru}\ \text{fls}\)

= \((x.\ y.x)\ \text{tru}\ \text{fls}\)

--> \((y.\text{tru})\ \text{fls}\)

--> \(\text{tru}\)
NUMBERS
Encoding Numbers

zero = \s.\z.z
one  = \s.\z.s z
two  = \s.\z.s (s z)
...
n     = \s.\z.s (s (s (.... z)))
      \overbrace{n of them}
Encoding Numbers

zero = \s.\z.z
one  = \s.\z.s z
two  = \s.\z.s (s z)
...
\n= \s.\z.s (s (s (... z)))

addone = \n.\s.\z.s (n s z)
Encoding Numbers

zero = \s.\z.z
one  = \s.\z.s z
two  = \s.\z.s (s z)
...

\( n = \s.\z.s (s (s (\ldots z))) \)

\( \text{n of them} \)

addone = \n.\s.\z.s (n s z)

\text{addone zero}

\[= (\n.\s.\z.s (n s z)) (\s.\z.z)\]

\[\rightarrow \s.\z.s ((\s.\z.z) s z)\]
Encoding Numbers

\(\text{zero} = \lambda s. \lambda z. z\)
\(\text{one} = \lambda s. \lambda z. s\ z\)
\(\text{two} = \lambda s. \lambda z. s\ (s\ z)\)
...
\(n = \lambda s. \lambda z. s\ (s\ (\ldots\ z))\)

\(\text{addone} = \lambda n. \lambda s. \lambda z. s\ (n\ s\ z)\)

\(\text{addone}\ \text{zero}\)
\[= (\lambda n. \lambda s. \lambda z. s\ (n\ s\ z))\ (\lambda s. \lambda z. z)\]
\[\rightarrow \lambda s. \lambda z. s\ ((\lambda s. \lambda z. z)\ s\ z)\]
\[= \lambda s. \lambda z. s\ ((\lambda z. z)\ z)\]
\[= \lambda s. \lambda z. s\ z\]
\[= \text{one}\]

Evaluating underneath the lambda in the body of the expression yields semantically equivalent values, like in Haskell.
Encoding Numbers

zero = \( s.z.z \)

one = \( s.z.s \ z \)

two = \( s.z.s \ (s \ z) \)

...

\( n = \underbrace{s.z.s \ (s \ (s \ (\ldots \ z))))}_{n \ of \ them} \)

addone = \( n.s.z.s \ (n \ s \ z) \)

can we code addition?
Encoding Numbers

zero = \s.\z.z
one  = \s.\z.s z
two  = \s.\z.s (s z)
...

n       = \s.\z.s (s (s (.... z)))
          \underline{n of them}

addone = \n.\s.\z.s (n s z)

can we code addition? we need to basically "stack" the s from the
two numbers:
two == \s.\z.s (s z)    three == \s.\z.s (s (s z))
five == \s.\z. s (s (s (s (s z))))

core of three in place of
two's z
Encoding Numbers

zero = \s.\z.z
one  = \s.\z.s \ z
two  = \s.\z.s (s \ z)
...
\ n = \s.\z.s (s (s (.... \ z)))
      \ n of them

addone = \n.\s.\z.s (n \ s \ z)

can we code addition?

\n.\m. ...
Encoding Numbers

zero = \s.\z.z
one  = \s.\z.s z
two  = \s.\z.s (s z)
...
n    = \s.\z.s (s (s (.... z)))))

addone = \n.\s.\z.s (n s z)

can we code addition?

\n.\m.(\s.\z. ... )
Encoding Numbers

zero = \s.\z.z
one  = \s.\z.s z
two  = \s.\z.s (s z)
...
n   = \s.\z.s (s (s (.... z)))
    \underbrace{\text{n of them}}
addone = \n.\s.\z.s (n s z)

can we code addition?

\n.\m.(\s.\z. n s m)
Encoding Numbers

zero = \s.\z.z
one  = \s.\z.s z
two  = \s.\z.s (s z)
...

\begin{align*}
n & = \s.\z.s (s (s (\ldots \ z))) \\
& \quad \text{n of them} \\
\end{align*}

addone = \n.\s.\z.s (n s z)

can we code addition?

\begin{align*}
\n.\m.&(\s.\z.n s m) \text{ two three} \\
& \quad \rightarrow^* \s.\z. \text{ two s three} \\
& \quad = \s.\z. s (s \text{ three}) \\
& \quad = \s.\z. s (s (\s.\z.s (s (s z))))
\end{align*}
Encoding Numbers

zero = \s.\z.z
one  = \s.\z.s z
two  = \s.\z.s (s z)
...
n     = \s.\z.s (s (s (.... z)))

addone = \n.\s.\z.s (n s z)

can we code addition?

\n.\m.(\s.\z. n s (m s z))

n of them
Encoding Numbers

• try multiplication, subtraction (harder!) on your own
OTHER OPERATIONAL SEMANTICS
Other Operational Semantics

• We have seen one way to evaluate lambda terms
  – left-to-right, call-by-value operational semantics:

\[
\begin{align*}
\frac{e_1 \rightarrow e_1'}{e_1 e_2 \rightarrow e_1' e_2} & \quad \text{(app1)} \\
\frac{(\lambda x. e) \, v \rightarrow e[v/x]}{v} & \quad \text{(beta)} \\
\frac{e_2 \rightarrow e_2'}{v \, e_2 \rightarrow v \, e_2'} & \quad \text{(app2)}
\end{align*}
\]
Other Operational Semantics

- We have seen one way to evaluate lambda terms:
  - left-to-right, call-by-value operational semantics:
    
    $\text{(app1)} \quad \frac{e_1 \rightarrow e_1'}{\frac{e_1 \rightarrow e_1'}{e_1 \cdot e_2 \rightarrow e_1' \cdot e_2}}$  
    
    $\text{(beta)} \quad \frac{(\lambda x. e) \cdot v \rightarrow e[v/x]}{e_2 \rightarrow e_2'}$  
    
    $\text{(app2)} \quad \frac{v \cdot e_2 \rightarrow v \cdot e_2'}{e_2 \rightarrow e_2'}$

- right-to-left, call-by-value operational semantics:
    
    $\text{(app1')} \quad \frac{e_2 \rightarrow e_2'}{\frac{e_2 \rightarrow e_2'}{e_1 \cdot e_2 \rightarrow e_1 \cdot e_2'}}$

    $\text{(beta)} \quad \frac{(\lambda x. e) \cdot v \rightarrow e[v/x]}{e_1 \rightarrow e_1'}$  

    $\text{(app2')} \quad \frac{e_1 \cdot v \rightarrow e_1' \cdot v}{e_1 \rightarrow e_1'}$
Other Operational Semantics

- We have seen one way to evaluate lambda terms
  - left-to-right, call-by-value operational semantics:

\[
\begin{align*}
\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2} & \quad \text{(app1)} \\
\frac{(\lambda x. e) v \rightarrow e [v/x]}{} & \quad \text{(beta)} \\
\frac{e_2 \rightarrow e'_2}{v e_2 \rightarrow v e'_2} & \quad \text{(app2)}
\end{align*}
\]

- call-by-name operational semantics (more similar to Haskell):

\[
\begin{align*}
\frac{(\lambda x. e) e_1 \rightarrow e [e_1/x]}{} & \quad \text{(beta-name)} \\
\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2} & \quad \text{(app1)}
\end{align*}
\]
Call-by-Name vs. Call-by-Value

• An example:

\[
\text{loop} = (\lambda x.x) (\lambda x.x)
\]

(\lambda y.y) \text{ loop}

• Under call-by-value:

(\lambda y.y) \text{ loop} \rightarrow (\lambda y.y) \text{ loop} \rightarrow (\lambda y.y) \text{ loop} \rightarrow (\lambda y.y) \text{ loop}

• Under call-by-name:

(\lambda y.y) \text{ loop} \rightarrow y

• Call-by-name terminates strictly more often
Full beta reduction will evaluate any function application anywhere within an expression, even inside a function body before the function has been called:

\[ (\lambda x. e) e_1 \rightarrow e [e_1/x] \]  (beta)

\[ e_1 \rightarrow e_1' \]
\[ e_1 e_2 \rightarrow e_1' e_2 \]  (app1)

\[ e_2 \rightarrow e_2' \]
\[ e_1 e_2 \rightarrow e_1 e_2' \]  (app2)

\[ e \rightarrow e' \]
\[ \lambda x. e \rightarrow \lambda x. e' \]  (fun)

Full beta is useful not for computing but for reasoning about which programs are equivalent to which other ones.
Full Beta Reduction

• Full beta reduction will evaluate any function application anywhere within an expression, even inside a function body before the function has been called:

\[
(\lambda x.e) e \rightarrow e[\lambda x]/x
\]

\[
\begin{align*}
\frac{e \rightarrow e'}{e1 \rightarrow e1'} \quad \text{(app1)} & \quad \frac{e2 \rightarrow e2'}{e1 e2 \rightarrow e1 e2'} \quad \text{(app2)} \\
\frac{\lambda x.e \rightarrow \lambda x.e'}{\cdot}
\end{align*}
\]

• Full beta is useful not for computing but for reasoning about which programs are equivalent to which other ones

• Full beta is highly non-deterministic -- lots of different reductions could apply at any point
Recall reasoning about the church encoding of numbers. We used full beta to reason about equivalence:

\[
\lambda s. \lambda z. s ((\lambda s. \lambda z. s) z) \rightarrow \lambda s. \lambda z. s ((\lambda z. z) z) \rightarrow \lambda s. \lambda z. s z \equiv \text{one}
\]
We can encode many objects

- loops
- if statements
- booleans
- pairs
- numbers
- and many more:
  - lists, trees and datatypes
  - exceptions, loops, ...
  - ...
- the general trick:
  - values (true, false, pairs) will be functions
  - construct these functions so that they return the appropriate information when called by an operation
Summary

• The Lambda Calculus involves just 3 things:
  – variables x, y, z
  – function definitions \( \lambda x.e \)
  – function application \( e_1 e_2 \)

• Despite its simplicity, despite the apparent lack of if statements or loops or any data structures other than functions, it is Turing complete

• Church encodings are translations that show how to encode various data types or linguistic features in the lambda calculus