Lambda Calculus 2

COS 441 Slides 14

read: 3.4, 5.1, 5.2, 3.5 Pierce

Lambda Calculus

- The lambda calculus is a language of pure functions
  - expressions: e ::= x | \x.e | e1 e2
  - values: v ::= \x.e
  - call-by-value operational semantics:

$$\frac{}{(\backslash x.e)\ v \dashrightarrow e\ [v/x]}\ \text{(beta)}$$

$$\frac{e1 \dashrightarrow e1'}{e1\ e2 \dashrightarrow e1'\ e2}\ \text{(app1)} \qquad \frac{e2 \dashrightarrow e2'}{v\ e2 \dashrightarrow v\ e2'}\ \text{(app2)}$$

  - example execution: (\x.x x) (\y.y) --> (\y.y) (\y.y) --> \y.y

ENCODING BOOLEANS

booleans

- the encoding:

tru = \t.\f. t

fls = \t.\f. f

test = \x.\then.\else. x then else

Challenge

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

test = \x.\then.\else. x 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

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

and = \b.\c. b c fls

and tru tru

-->* tru tru fls

-->* 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
```

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)

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 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)

= (\p.p tru) (\b. b 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)

= (\p.p tru) (\b. b tru fls)

--> (\b. b tru fls) tru

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)

= (\p.p tru) (\b. b tru fls)

--> (\b. b tru fls) tru

--> tru tru fls

= (\x.\y.x) tru fls

--> (\y.tru) fls

--> 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)))

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)))

n of them

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 (.... z)))

n of them

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

addone zero
== (\n.\s.\z.s (n s z)) (\s.\z.z)
--> \s.\z.s ((\s.\z.z) 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)

addone zero
== (\n.\s.\z.s (n s z)) (\s.\z.z)
--> \s.\z.s ((\s.\z.z) s z)
== \s.\z.s ((\z.z) z)
== \s.\z.s z
== one

evaluating underneith 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 = \s.\z.s (s (s (.... 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)))

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)))
```

n of them

```
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)))

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)

...

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

n of them

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

can we code addition?

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

(\n.\m.(\s.\z.n s m)) two three
-->* \s.\z. two s three
== \s.\z. s (s three)
== \s.\z. s (s (\s.\z.s (s (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.(\s.\z. n s (m s z))

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:

```
 ______________________________ (beta)
   (\x.e) v --> e [v/x]

      e1 --> e1'                          e2 --> e2'
 ____________________ (app1)         ___________________ (app2)
   e1 e2 --> e1' e2                    v e2 --> v e2'
```

Other Operational Semantics

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

$$\frac{}{\text{(\textbackslash x.e) v --> e [v/x]}}\ \text{(beta)}$$

$$\frac{\text{e1 --> e1'}}{\text{e1 e2 --> e1' e2}}\ \text{(app1)} \qquad \frac{\text{e2 --> e2'}}{\text{v e2 --> v e2'}}\ \text{(app2)}$$

  - right-to-left, call-by-value operational semantics:

$$\frac{}{\text{(\textbackslash x.e) v --> e [v/x]}}\ \text{(beta)}$$

$$\frac{\text{e2 --> e2'}}{\text{e1 e2 --> e1 e2'}}\ \text{(app1')} \qquad \frac{\text{e1 --> e1'}}{\text{e1 v --> e1' v}}\ \text{(app2')}$$

Other Operational Semantics

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

$$\frac{}{\text{(\textbackslash x.e) v --> e [v/x]}}\ \text{(beta)}$$

$$\frac{\text{e1 --> e1'}}{\text{e1 e2 --> e1' e2}}\ \text{(app1)}$$

$$\frac{\text{e2 --> e2'}}{\text{v e2 --> v e2'}}\ \text{(app2)}$$

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

$$\frac{}{\text{(\textbackslash x.e) e1 --> e [e1/x]}}\ \text{(beta-name)}$$

$$\frac{\text{e1 --> e1'}}{\text{e1 e2 --> e1' e2}}\ \text{(app1)}$$

Call-by-Name vs. Call-by-Value

- An example:

loop = (\x.x x) (\x.x x)

(\x.\y.y) loop

- Under call-by-value:

(\x.\y.y) loop --> (\x.\y.y) loop --> (\x.\y.y) loop --> (\x.\y.y) loop

- Under call-by-name:

(\x.\y.y) loop --> \y.y

- Call-by-name terminates strictly more often

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:

```
------------------------------ (beta)
(\x.e) e1 --> e [e1/x]

e1 --> e1’
------------------------------ (app1)
e1 e2 --> e1’ e2

e2 --> e2’
------------------------------ (app2)
e1 e2 --> e1 e2’

e --> e'
------------------------------ (fun)
\x.e --> \x.e'
```

- 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:

$$\frac{}{(\backslash x.e)\ e1 \dashrightarrow e\ [e1/x]}\ \text{(beta)}$$

$$\frac{e1 \dashrightarrow e1'}{e1\ e2 \dashrightarrow e1'\ e2}\ \text{(app1)} \qquad \frac{e2 \dashrightarrow e2'}{e1\ e2 \dashrightarrow e1\ e2'}\ \text{(app2)}$$

$$\frac{e \dashrightarrow e'}{\backslash x.e \dashrightarrow \backslash x.e'}\ \text{(fun)}$$

- 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

Full-Beta Reduction

- Recall reasoning about the church encoding of numbers
- We used full beta to reason about equivalence:

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

SUMMARY

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 \x.e
  - function application e1 e2
- 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