Lambda Calculus

COS 441 Slides 12

read: 3.4, 5.1, 5.2, 3.5 Pierce

the lambda calculus

- Originally, the lambda calculus was developed as a logic by Alonzo Church in 1932 at Princeton
  - Church says: “There may, indeed, be other applications of the system than its use as a logic.”
  - Dave says: “There sure are!”
- The lambda calculus is a language of pure functions
- It serves as the semantic basis for languages like Haskell that are based around functions, but also pretty much every other language that includes some notion of function
- It is just as powerful as a Turing Machine (lambda terms can compute anything a Turing Machine can) and provides an alternate basis for understanding computation

- Pierce Text, Chap 3, 5

Operational Semantics

- Denotational semantics for a language provides a function that translates from program syntax into mathematical objects like sets, functions, lists or even some other programming language
  - a denotational semantics acts like a compiler
- Operational semantics works by rewriting or executing programs step-by-step
  - it uses only one program syntax to explain how a program runs
- As languages become more complicated, it is often easier to define operational semantics than denotational semantics
  - it requires less math to do so
  - but you might not be able to prove particularly strong theorems using the semantics
- Starting with the lambda calculus, we will look at operational semantics

Operational Rules

- Operational rules typically look like this:

$$\frac{\text{condition1 .... conditionk} \qquad \text{subprogram --> subprogram'}}{\text{prog --> prog'}}$$

- Read prog --> prog' as prog "steps to" prog'
- prog --> prog' is a new kind of judgement (aka property/assertion/claim)

Operational Rules

- Operational rules typically look like this:

$$\frac{\text{condition1 .... conditionk} \qquad \text{subprogram --> subprogram'}}{\text{prog --> prog'}}$$

- Read prog --> prog' as prog "steps to" prog'
- prog --> prog' is a new kind of judgement (aka property/assertion/claim)
- An example, defining evaluation of if statements:

$$\frac{\text{e --> e'}}{\text{if e then c1 else c2 --> if e' then c1 else c2}}$$

$$\frac{}{\text{if True then c1 else c2 --> c1}} \qquad \frac{}{\text{if False then c1 else c2 --> c2}}$$

LAMBDA CALCULUS

syntax

```
e ::= x            (a variable)
    | \x.e         (a function; in Haskell: \x -> e)
    | e e          (function application)
```

[“\” will be written “λ” in a nice font and pronounced "lambda"]

syntax

- the identity function:
    - \x.x
- 2 notational conventions:
    - applications associate to the left (like in Haskell):
    - “y z x”   is   “(y z) x”
    - the body of a lambda extends as far as possible to the right:
    - “\x.x \z.x z x”   is   “\x.(x \z.(x z x))”

terminology

\x.x x

the scope of x is the entire body of the function
(ie: the x's that appear in the body of the function refer to that particular argument)

\x.x y

y is free in the term \x.x y

x is bound
in the term \x.x y

scope again, shadowed names

the scope of the right-most x includes the body of the function; the scope of the left-most x does not

\x.\x. x x

\y.\x. x y

if you wanted to refer to the first x, above, well you can't. You should have chosen a different variable name in your programs

Important note: The names of bound variables don't matter to the semantics of lambda calculus programs, so you can rename bound variables (provided you do so consistently) whenever you want.

\x.x == \y.y == \z.z

\x.\y.x y == \y.\x.y x == \z.\w. z w

Call-by-value operational semantics

- single-step, call-by-value operational semantics:

  e --> e'

- In English, we say "e steps to e'"

- This is a new kind of "judgement", just like a Hoare triple was a judgement and there were rules that allowed us to conclude when it was a valid judgement

Call-by-value operational semantics

- single-step, call-by-value operational semantics: e --> e'
  - values are v ::= \x.e
  - primary rule (beta reduction):

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

call-by-value
since argument is a
value rather than
general expression

  - e [v/x] is the expression in which all free occurrences of x in e are replaced with v
  - this replacement operation is called substitution
  - implementing substitution for the lambda calculus properly is actually tougher than it would seem at first

operational semantics

- beta rule:

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

- is used together with search rules:

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

- notice, because of the rules, evaluation is left to right

- and that's it -- 3 rules -- that is all you need to know about evaluating expressions in the lambda calculus!

Example

- Program:

  ((\x.\y. x y) (\w.w)) (\z.z)

- Proof that it can take a step:

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

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

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

Example

- Program:

  ((\x.\y. x y) (\w.w)) (\z.z)

- Proof that it can take a step:

$$\dfrac{\dfrac{}{(\backslash x.\backslash y.\ x\ y)\ (\backslash w.w) \text{ --> } \backslash y.\ (\backslash w.w)\ y}\ \text{(beta)}}{\underbrace{((\backslash x.\backslash y.\ x\ y)\ \ (\backslash w.w))}_{e1}\ \ \underbrace{(\backslash z.z)}_{e2} \text{ --> } \underbrace{(\backslash y.\ (\backslash w.w)\ y)}_{e1'}\ \underbrace{(\backslash z.z)}_{e2}}\ \text{(app1)}$$

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

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

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

Example

- Program:

  ((\x.\y. x y) (\w.w)) (\z.z)

- Proof that it can take a step:

$$\dfrac{\dfrac{}{(\backslash x.\backslash y.\ x\ y)\ (\backslash w.w) \;\text{-->}\; \backslash y.\ (\backslash w.w)\ y}\ \text{(beta)}}{((\backslash x.\backslash y.\ x\ y)\ \ (\backslash w.w))\ \ (\backslash z.z) \;\text{-->}\; (\backslash y.\ (\backslash w.w)\ y)\ (\backslash z.z)}\ \text{(app1)}$$

- Proof it can take a second step:

$$\dfrac{}{(\backslash y.\ (\backslash w.w)\ y)\ \ (\backslash z.z) \;\text{-->}\; (\backslash w.w)\ (\backslash z.z)}\ \text{(beta)}$$

- So we typically write (without explicit proofs):

  ((\x.\y. x y) (\w.w)) (\z.z) --> (\y. (\w.w) y) (\z.z) --> (\w.w) (\z.z)

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

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

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

Example

(\x.x x) (\y.y)

Example

(\x.x x) (\y.y)

--> x x [\y.y / x]

Example

```
    (\x.x x) (\y.y)
--> x x [\y.y / x]
==  (\y.y) (\y.y)
```

Example

(\x.x x) (\y.y)

--> x x [\y.y / x]

== (\y.y) (\y.y)

--> y [\y.y / y]

Example

```
    (\x.x x) (\y.y)
--> x x [\y.y / x]
== (\y.y) (\y.y)
--> y [\y.y / y]
== \y.y
```

A Non-Example

- Given:

((\x.x) (\y.y)) ((\w.w) (\z.z))

- One might think that:

((\x.x) (\y.y)) ((\w.w) (\z.z)) --> ((\x.x) (\y.y)) (\z.z)

- Since: (\w.w) (\z.z) --> (\z.z)

- But that would require the presence of this rule:

$$\frac{e2\ \text{-->}\ e2'}{e1\ e2\ \text{-->}\ e1\ e2'}\ \text{(app3)}$$

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

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

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

Another example

(\x.x x) (\x.x x)

Another example

(\x.x x) (\x.x x)

--> x x [\x.x x/x]

Another example

(\x.x x) (\x.x x)

--> x x [\x.x x/x]

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

- In other words, it is simple to write non-terminating computations in the lambda calculus

- So, what else can we do with the lambda calculus?

We can do everything

- The lambda calculus can be used as an “assembly language”
- We can show how to compile useful, high-level operations and language features into the lambda calculus
  - Result = adding high-level operations is convenient for programmers, but not a computational necessity
  - Result = make your compiler intermediate language simpler

- Translations that show how to implement various useful programming features in the lambda calculus are typically called "Church encodings" after Alonzo Church

Aside

- Single-step reduction, one by one, gets pretty tedious, so we can make up a new notation for multi-step evaluation (and give the new notation a formal definition!)
- To say a program takes 0, 1 or many steps, we write:

e -->* e'

- Rules:

$$\frac{}{\text{e -->* e}}\ \text{(reflexivity)} \qquad \frac{\text{e1 --> e2} \qquad \text{e2 -->* e3}}{\text{e1 -->* e3}}\ \text{(transitivity)}$$

Aside

$$\frac{}{\texttt{e -->* e}} \quad \text{(reflexivity)}$$

$$\frac{\texttt{e1 --> e2} \qquad \texttt{e2 -->* e3}}{\texttt{e1 -->* e3}} \quad \text{(transitivity)}$$

- A multi-step proof:

$$\frac{\texttt{a --> b} \qquad \texttt{b -->* e}}{\texttt{a -->* e}}$$

Aside

$$\frac{}{\text{e -->* e}}\ \text{(reflexivity)}$$

$$\frac{\text{e1 --> e2} \qquad \text{e2 -->* e3}}{\text{e1 -->* e3}}\ \text{(transitivity)}$$

- A multi-step proof:

$$\frac{\text{a --> b} \qquad \dfrac{\text{b --> c} \qquad \text{c -->* e}}{\text{b -->* e}}}{\text{a -->* e}}$$

Aside

$$\frac{}{e \dashrightarrow^{*} e}\ \text{(reflexivity)}$$

$$\frac{e1 \text{ --> } e2 \qquad e2 \text{ -->* } e3}{e1 \text{ -->* } e3}\ \text{(transitivity)}$$

- A multi-step proof:

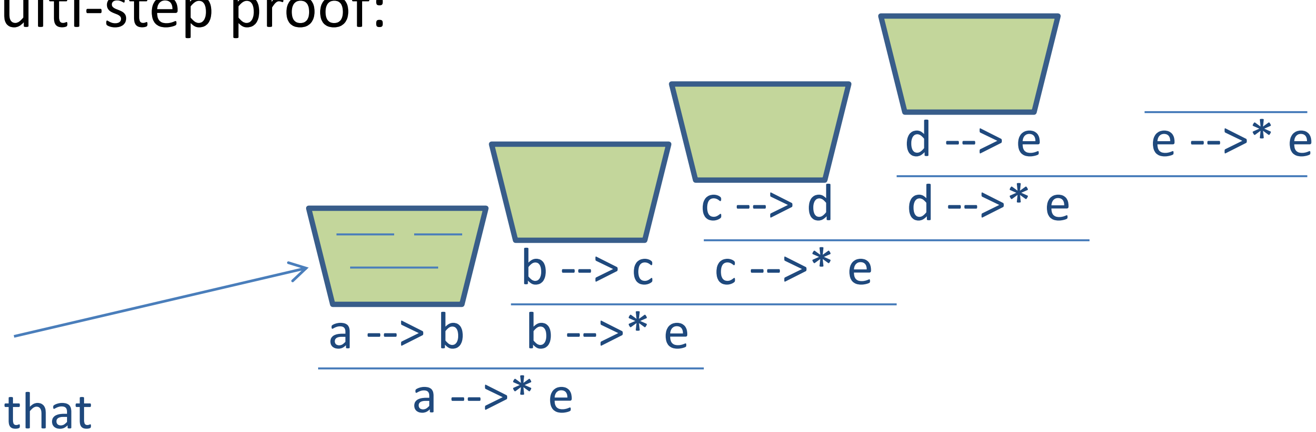
$$\dfrac{a \text{ --> } b \qquad \dfrac{b \text{ --> } c \qquad \dfrac{c \text{ --> } d \qquad \dfrac{d \text{ --> } e \qquad \dfrac{}{e \text{ -->* } e}}{d \text{ -->* } e}}{c \text{ -->* } e}}{b \text{ -->* } e}}{a \text{ -->* } e}$$

Aside

$$\frac{}{e \dashrightarrow^{*} e} \quad \text{(reflexivity)}$$

$$\frac{e1 \text{ --> } e2 \qquad e2 \text{ -->* } e3}{e1 \text{ -->* } e3} \quad \text{(transitivity)}$$

- A multi-step proof:



proof that
a --> b

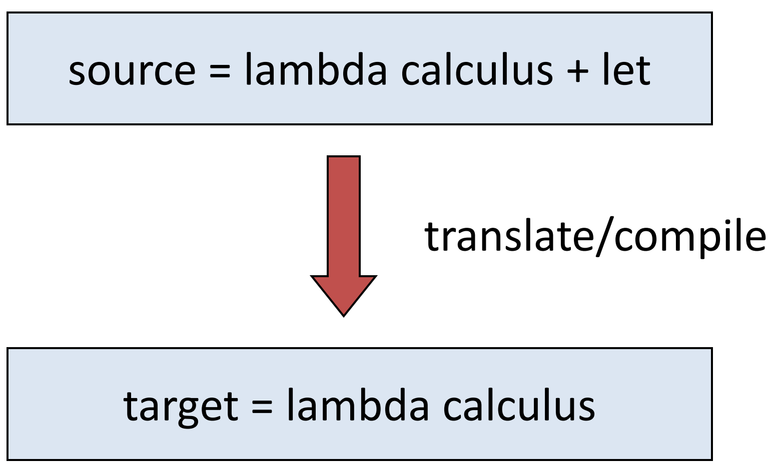
CHURCH ENCODINGS

Let Expressions

- It is useful to bind intermediate results of computations to variables:

  let x = e1 in e2

- Question: can we implement this idea in the lambda calculus?



Let Expressions

- It is useful to bind intermediate results of computations to variables:

  let x = e1 in e2

- Question: can we implement this idea in the lambda calculus?

  translate (let x = e1 in e2) =

Let Expressions

- It is useful to bind intermediate results of computations to variables:

  let x = e1 in e2

- Question: can we implement this idea in the lambda calculus?

  translate (let x = e1 in e2) =

  (\x. translate e2) (translate e1)

Let Expressions

- It is useful to bind intermediate results of computations to variables:

  let x = e1 in e2

- Question: can we implement this idea in the lambda calculus?

  translate (let x = e1 in e2) =

  (\x. translate e2) (translate e1)

  translate (x) = x

  translate (\x.e) = \x.translate e

  translate (e1 e2) = (translate e1) (translate e2)

ENCODING BOOLEANS

booleans

- we can encode booleans
  - we will represent "true" and "false" as functions named "tru" and "fls"
  - how do we define these functions?
  - think about how "true" and "false" can be used
  - they can be used by a testing function:
    - "test b then else" returns "then" if b is true and returns "else" if b is false
    - the only thing the implementation of test is going to be able to do with b is to apply it
    - the functions "tru" and "fls" must distinguish themselves when they are applied

booleans

- the encoding:

tru = \t.\f. t

fls = \t.\f. f

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

booleans

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

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

eg:

test tru a b

booleans

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

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

eg:

test tru a b

== (\x.\then.\else. x then else) (\t.\f.t) a b

booleans

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

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

eg:

test tru a b

== (\x.\then.\else. x then else) (\t.\f.t) a b

-->* (\t.\f. t) a b

booleans

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

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

eg:

test tru a b

== (\x.\then.\else. x then else) (\t.\f.t) a b

-->* (\t.\f. t) a b

-->* a

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

SUMMARY

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