CSEP505: Programming Languages Lecture 4: Untyped Lambda-Calculus, Formal Operational Semantics,

Dan Grossman Autumn 2016

Where are we

- To talk about functions more precisely, we need to define them as carefully as we did IMP's constructs
- First try adding functions & local variables to IMP "on the cheap"
 It didn't work [see last week]
- · Now back up and define a language with nothing but functions
 - [started last week]
 - And then *encode* everything else

Lecture 4

CSE P505 August 2016 Dan Grossman

Review

- Cannot properly model local scope via a global heap of integers
 - Functions are not syntactic sugar for assignments to globals
- · So let's build a model of this key concept
 - Or just borrow one from 1930s logic
- · And for now, drop mutation, conditionals, and loops
 - We won't need them!
- The Lambda calculus in BNF

Expressions: $e := x \mid \lambda x. \ e \mid e \ e$ Values: $v := \lambda x. \ e$

Lecture 4

CSE P505 August 2016 Dan Grossman

That's all of it! [More review]

Expressions: $e := x \mid \lambda x. e \mid e e$

Values: $v := \lambda x. e$

A program is an e. To call a function:

substitute the argument for the bound variable

That's the key operation we were missing

Example substitutions:

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

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

$$(\lambda x. \ x \ x) \ (\lambda x. \ x \ x) \ (\lambda x. \ x \ x)$$

Lecture 4

Lecture 4

CSE P505 August 2016 Dan Grossman

Why substitution [More review]

- After substitution, the bound variable is gone
 - So clearly its name didn't matter
 - That was our problem before
- Given substitution we can define a little programming language
 - (correct & precise definition is subtle; we'll come back to it)
 - This microscopic PL turns out to be Turing-complete

Full large-step interpreter

CSE P505 August 2016 Dan Grossman

Lecture 4

Interpreter summarized

- Evaluation produces a value Lam(x,e3) if it terminates
- · Evaluate application (call) by
 - 1. Evaluate left
 - 2. Evaluate right
 - 3. Substitute result of (2) in body of result of (1)
 - 4. Evaluate result of (3)

A different semantics has a different evaluation strategy:

- 1. Evaluate left
- 2. Substitute right in body of result of (1)
- 3. Evaluate result of (2)

Lecture 4

CSE P505 August 2016 Dan Grossman

Another interpreter

Lecture 4

CSE P505 August 2016 Dan Grossman

What have we done

- Syntax and two large-step semantics for the untyped lambda calculus
 - First was "call by value"
 - Second was "call by name"
- · Real implementations don't use substitution
 - They do something equivalent
- · Amazing (?) fact:
 - If call-by-value terminates, then call-by-name terminates
 - (They might both not terminate)

Lecture 4

CSE P505 August 2016 Dan Grossman

What will we do

- · Go back to math metalanguage
 - Notes on concrete syntax (relates to OCaml)
 - Define semantics with inference rules
- Lambda encodings (show our language is mighty)
- · Define substitution precisely
- Environments

Next time??

- Small-step
- Play with continuations ("very fancy" language feature)

Lecture 4

Lecture 4

CSE P505 August 2016 Dan Grossman

Syntax notes

- When in doubt, put in parentheses
- Math (and OCaml) resolve ambiguities as follows:
- 1. λx. e1 e2 is (λx. e1 e2)
 - not (λx. e1) e2

General rule: Function body "starts at the dot" and "ends at the first unmatched right paren"

Example:

(λx. y (λz. z) w) q

Syntax notes

```
2. e1 e2 e3 is (e1 e2) e3
```

- not e1 (e2 e3)

General rule: Application "associates to the left"

So e1 e2 e3 e4 is (((e1 e2) e3) e4)

Lecture 4 CSE P505 August 2016 Dan Grossman

CSE P505 August 2016 Dan Grossman

12

It's just syntax

- · As in IMP, we really care about abstract syntax
 - Here, internal tree nodes labeled "λ" or "apply" (i.e., "call")
- · Previous 2 rules just reduce parens when writing trees as strings
- · Rules may seem strange, but they're the most convenient
 - Based on 70 years experience
 - Especially with currying

Lecture 4

CSE P505 August 2016 Dan Grossman

13

What will we do

- · Go back to math metalanguage
 - Notes on concrete syntax (relates to OCaml)
 - Define semantics with inference rules
- · Lambda encodings (show our language is mighty)
- · Define substitution precisely
- Environments

Next time??

- Small-step
- Play with continuations ("very fancy" language feature)

Lecture 4

CSE P505 August 2016 Dan Grossman

14

Inference rules

- A metalanguage for operational semantics
 - Plus: more concise (& readable?) than OCaml
 - Plus: useful for reading research papers
 - Plus: natural support for nondeterminism
 - Definition allowing observably different implementations
 - Minus: less tool support than OCaml (no compiler)
 - Minus: one more thing to learn
 - Minus: painful in Powerpoint

Lecture 4

CSE P505 August 2016 Dan Grossman

15

Informal idea

Want to know:

what values (0, 1, many?) an expression can evaluate to

So define a relation over pairs (e,v):

- Where e is an expression and v is a value
- Just a subset of all pairs of expressions and values

If the language is deterministic, this *relation* turns out to be a *function* from expressions to values

Metalanguage supports defining relations

- Then prove a relation is a function (if it is)

Lecture 4

CSE P505 August 2016 Dan Grossman

16

Making up metasyntax

Rather than write (\mathbf{e}, \mathbf{v}) , we'll write $e \mathbf{v}$.

- It's just metasyntax (!)
 - Could use interp(e,v) or « v e » if you prefer
- Our metasyntax follows PL convention
 - Colors are not conventional (slides: green = metasyntax)
- And distinguish it from other relations

First step: define the form (arity and metasyntax) of your relation(s):



This is called a judgment

What we need to define

So we can write $e \Psi v$ for any e and v

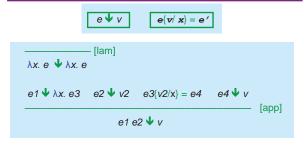
 But we want such a thing to be "true" to mean e can evaluate to v and "false" to mean it cannot

Examples (before the definition):

- $(\lambda x. \lambda y. y x) ((\lambda z. z) (\lambda z. z)) \Psi \lambda y. y (\lambda z. z)$ in the relation
- λy. y Ψ λy. y in the relation
- $(\lambda x. x x) (\lambda x. x x) \Psi \lambda y. y$ not in the relation
- (λx. x x) (λx. x x) ↓ (λx. x x) (λx. x x) metasyntactically bogus

Lecture 4

Inference rules



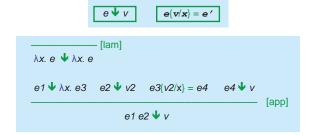
- Using definition of a set of 4-tuples for substitution
 - (exp * value * variable * exp)
 - · Will define substitution later

Lecture 4

CSE P505 August 2016 Dan Grossman

19

Inference rules



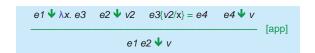
- Rule top: hypotheses (0 or more)
- Rule bottom: conclusion
- · Metasemantics: If all hypotheses hold, then conclusion holds

Lecture 4

CSE P505 August 2016 Dan Grossman

20

Rule schemas



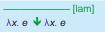
- Each rule is a schema you "instantiate consistently"
- So [app] "works" "for all" x, e1, e2, e3, e4, v2, and v
- But "each" e1 has to be the "same" expression
 - · Replace metavariables with appropriate terms
 - Deep connection to logic programming (e.g., Prolog)

Lecture 4

CSE P505 August 2016 Dan Grossman

21

Instantiating rules



- Two example legitimate instantiations:
 - - x instantiated with z, e instantiated with z
 - λz. λy. y z
 ↓ λz. λy. y z
 - x instantiated with z, e instantiated with λy. y z
- · Two example illegitimate instantiations:
 - λz. z
 ↓ λy. z

Must get your rules "just right" so you don't allow too much or too little

Lecture 4

CSE P505 August 2016 Dan Grossman

22

24

Derivations

- Tuple is "in the relation" if there exists a derivation of it
 - An upside-down (or not?!) tree where each node is an instantiation and leaves are axioms (no hypotheses)
- To show e ♥ v for some e and v, give a derivation
 - But we rarely "hand-evaluate" like this
 - We're just defining a semantics remember
- Let's work through an example derivation for (λx. λy. y x) ((λz. z) (λz. z)) Ψ λy. y (λz. z)

Which relation?

So exactly which relation did we define

- The pairs at the bottom of finite-height derivations

Note: A derivation tree is like the tree of calls in a large-step interpreter

- [when relation is a function]
- Rule being instantiated is branch of the match-expression
- Instantiation is arguments/results of the recursive call

Lecture 4 CSE P505 August 2016 Dan Grossman 23 Lecture 4 CSE P505 August 2016 Dan Grossman

A couple extremes

 This rules are a bad idea because either one adds all pairs to the relation





· This rule is pointless because it adds no pairs to the relation



Lecture 4

CSE P505 August 2016 Dan Grossman

25

Summary so far

- · Define judgment via a collection of inference rules
 - Tuple in the relation ("judgment holds") if a derivation (tree of instantiations ending in axioms) exists

As an interpreter, could be "nondeterministic":

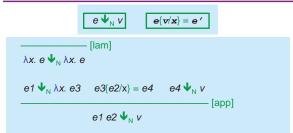
- Multiple derivations, maybe multiple v such that $e \Psi v$
 - Our example language is deterministic
 - In fact, "syntax directed" (≤1 rule per syntax form)
- Still need rules for $e\{v/x\}=e'$
- · Let's do more judgments (i.e., languages) to get the hang of it...

Lecture 4

CSE P505 August 2016 Dan Grossman

26

Call-by-name large-step



- · Easier to see the difference than in OCaml
- · Formal statement of amazing fact:

For all e, if there exists a v such that $e \checkmark v$ then there exists a v2 such that $e \checkmark v2$

(Proof is non-trivial & must reason about substitution)

ecture 4 CSE P505 August 2016 Dan Grossman

IMP

- Two judgments H;e

 i and H;s

 H2
- Assume get(H,x,i) and set(H,x,i,H2) are defined
- · Let's try writing out inference rules for the judgments...

Lecture 4

CSE P505 August 2016 Dan Grossman

28

What will we do

- · Go back to math metalanguage
 - Notes on concrete syntax (relates to OCaml)
 - Define semantics with inference rules
- · Lambda encodings (show our language is mighty)
- · Define substitution precisely
- Environments

Next time??

- · Small-step
- Play with continuations ("very fancy" language feature)

Encoding motivation

- Fairly crazy: we left out integers, conditionals, data structures, ...
- · Turns out we're Turing complete
 - We can encode whatever we need
 - (Just like assembly language can)
- · Motivation for encodings
 - Fun and mind-expanding
 - Shows we are not oversimplifying the model ("numbers are syntactic sugar")
 - Can show languages are too expressive
 Example: C++ template instantiation
- Encodings are also just "(re)definition via translation"

Lecture 4

CSE P505 August 2016 Dan Grossman

Lecture 4

CSE P505 August 2016 Dan Grossman

Encoding booleans

The "Boolean Abstract Data Type (ADT)"

- · There are 2 booleans and 1 conditional expression
 - The conditional takes 3 (curried) arguments
 - If 1st argument is one bool, return 2nd argument
 - · If 1st argument is other bool, return 3rd argument
- Any set of 3 expressions meeting this specification is a proper encoding of booleans
- · Here is one (of many):
 - "true" λx. λy. x
 - "false" λx. λy. y
 - "if" λb . λt . λf . b t f

Lecture 4

CSE P505 August 2016 Dan Grossman

31

Example

- · Given our encoding:
 - "true" **λx**. **λy**. **x**
 - "false" λx. λy. y
 - "if" λb . λt . λf . b t f
- And every "law of booleans" works out
 - And every non-law does not
- · By the way, this is OOP

Lecture 4

CSE P505 August 2016 Dan Grossman

32

But...

- Evaluation order matters!
 - With ♥, our "if" is not YFL's if

"if" "true" (\(\lambda x \cdot x \cdot x)\) doesn't terminate but

"if" "true" ($\lambda x.x$) ($\lambda z.$ ($\lambda x.xx$) ($\lambda x.xx$) ($\lambda x.xx$) z) terminates

Lecture 4

CSE P505 August 2016 Dan Grossman

33

Encoding pairs

- · The "Pair ADT"
 - There is 1 constructor and 2 selectors
 - 1st selector returns 1st argument passed to the constructor
 - 2nd selector returns 2nd argument passed to the constructor
- This does the trick:

- "make_pair" λx . λy . λz . z x y

- "first" $\lambda p. p (\lambda x. \lambda y. x)$

- "second" $\lambda p. p (\lambda x. \lambda y. y)$

Example:

Lecture 4

CSE P505 August 2016 Dan Grossman

34

Reusing Lambda

- Is it weird that the encodings of Booleans and pairs both used
 (λx. λy. x) and (λx. λy. y) for different purposes?
- Is it weird that the same bit-pattern in binary code can represent an int, a float, an instruction, or a pointer?
- · Von Neumann: Bits can represent (all) code and data
- · Church (?): Lambdas can represent (all) code and data
- · Beware the "Turing tarpit"

Encoding lists

- Why start from scratch? Can build on bools and pairs:
 - "empty-list" is "make pair" "false" "false"
 - "cons" is λħ.λt."make_pair" "true" "make_pair" h t
 - $\ \text{``is-empty''} \quad \text{is } \dots$
 - "head" is …
 - "tail" is ...
- Note
 - Not too far from how lists are implemented
 - Taking "tail" ("tail" "empty") will produce some lambda
 - Just like, without page-protection hardware,

 ${\tt null->tail->tail} \ \ would \ produce \ some \ bit-pattern$

Encoding natural numbers

- · Known as "Church numerals"
 - Will skip in the interest of time
- · The "natural number" ADT is basically:
 - "zero"
 - "successor" (the add-one function)
 - "plus"
 - "is-equal"
- Encoding is correct if "is-equal" agrees with elementary-school arithmetic
- [Don't need "full" recursion, but with "full" recursion, can also just do lists of Booleans...]

Lecture 4

CSE P505 August 2016 Dan Grossman

37

Recursion

· Can we write useful loops? Yes!

To write a recursive function:

- Write a function that takes an f and call f in place of recursion:
 - Example (in enriched language):

done only once. For the curious:

```
\lambda f. \lambda x. \text{if } x=0 \text{ then } 1 \text{ else } (x * f(x-1))
```

"fix" $\lambda f. \lambda x.$ if x=0 then 1 else (x * f(x-1))

• Then apply "fix" to it to get a recursive function

Details, especially in CBV are icky; but it's possible and need be

"fix" is λf . (λx .f (λy . x x y)) (λx .f (λy . x x y))

Lecture 4

CSE P505 August 2016 Dan Grossman

More on "fix"

- · "fix" is also known as the Y-combinator
- · The informal idea:
 - "fix" (λf.e) becomes something like

e{("fix" (\lambda f.e)) / f}

- That's unrolling the recursion once
- Further unrollings are delayed (happen as necessary)
- · Teaser: Most type systems disallow "fix"
 - So later we'll add it as a primitive
 - Example: OCaml can never type-check (x x)

Lecture 4

CSE P505 August 2016 Dan Grossman

39

What will we do

- · Go back to math metalanguage
 - Notes on concrete syntax (relates to OCaml)
 - Define semantics with inference rules
- Lambda encodings (show our language is mighty)
- · Define substitution precisely
- Environments

Next time??

- Small-step
- Play with continuations ("very fancy" language feature)

Lecture 4

CSE P505 August 2016 Dan Grossman

40

38

Our goal

Need to define

$$e1{e2/x} = e3$$

- · Used in [app] rule
- Informally, "replace occurrences of x in e1 with e2"
- Shockingly subtle to get right (in theory or programming)
- (Under call-by-value, only need e2 to be a value, but that doesn't make it much easier, so define the more general thing.)

Try #1[WRONG]

 $e1{e2/x} = e3$ y != x $e1{e2/x} = e3$

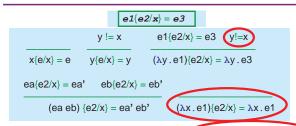
 $x\{e/x\} = e$ $y\{e/x\} = y$ $(\lambda y \cdot e1)\{e2/x\} = \lambda y \cdot e3$

 $ea{e2/x} = ea'$ $eb{e2/x} = eb'$ $(ea eb) {e2/x} = ea' eb'$

- Recursively replace every x leaf with e2
- But the rule for substituting into (nested) functions is wrong: If the function's argument binds the same variable (shadowing), we should not change the function's body
- Example program: (λx.λx.x) 42

Lecture 4

Try #2 [WRONG]



- Recursively replace every x leaf with e2, but respect shadowing
- · Still wrong due to capture [actual technical term]:
 - Example: (\(\lambda\)y.e1)\{y/x\}
 - Example (\(\lambda\)y . e1){(\(\lambda\)z.y/x}
 - In general, if "y appears free in e2"

Lecture 4

CSE P505 August 2016 Dan Grossman

43

More on capture

- Good news: capture can't happen under CBV or CBN
 If program starts with no unbound ("free") variables
- · Bad news: Can still result from "inlining"
- · Bad news: It's still "the wrong definition" in general
 - My experience: The nastiest of bugs in language tools

44

46

Lecture 4 CSE P505 August 2016 Dan Grossman

Try #3 [Almost Correct]

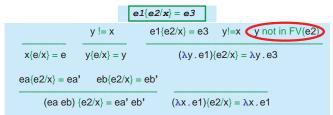
- First define an expression's "free variables" (braces here are set notation)
 - $FV(x) = \{x\}$
 - FV(e1 e2) = FV(e1) U FV(e2)
 - $FV(\lambda y.e) = FV(e) \{y\}$
- · Now require "no capture":

$$\frac{e1\{e2/x\} = e3 \quad y! = x \quad y \text{ not in FV(e2)}}{(\lambda y \cdot e1)\{e2/x\} = \lambda y \cdot e3}$$

Lecture 4

CSE P505 August 2016 Dan Grossman

Try #3 in Full



- · No mistakes with what is here...
- · ... but only a partial definition
 - What if y is in the free-variables of e2

Lecture 4 CSE P505 August 2016 Dan Grossman

Implicit renaming

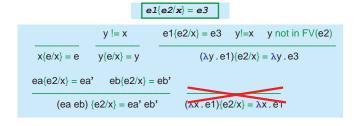
$$\frac{e1\{e2/x\} = e3 \quad y! = x \quad y \text{ not in FV}(e2)}{(\lambda y \cdot e1)\{e2/x\} = \lambda y \cdot e3}$$

- But this is a partial definition due to a "syntactic accident", until...
- · We allow "implicit, systematic renaming" of any term
 - In general, we never distinguish terms that differ only in variable names
 - A key language-design principle
 - Actual variable choices just as "ignored" as parens
 - Means rule above can "always apply" with a lambda
- Called "alpha-equivalence": terms differing only in names of variables are the same term

Try #4 [correct]

Lecture 4

• [Includes systematic renaming and drops an unneeded rule]



ecture 4 CSE P505 August 2016 Dan Grossman

More explicit approach

- · While "everyone in the PL field":
 - Understands the capture problem
 - Avoids it by saying "implicit systematic renaming" you may find that unsatisfying...
 - ... especially if you have to implement substitution while avoiding capture
- So this more explicit version also works ("fresh z for y"):

```
\frac{\text{z not in FV(e1) U FV(e2) U \{x\} e1\{z/y\} = e3 e3\{e2/x\} = e4}}{(\lambda y \cdot e1)\{e2/x\} = \lambda z \cdot e4}
```

 You have to "find an appropriate z", but one always exists and \$\$\tmp\$ appended to a global counter "probably works"

Lecture 4

CSE P505 August 2016 Dan Grossman

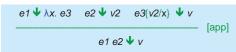
40

Note on metasyntax

- · Substitution often thought of as a metafunction, not a judgment
 - I've seen many nondeterministic languages
 - But never a nondeterministic definition of substitution
- · So instead of writing:

```
e1 \psi \lambda x. e3 e2 \psi v2 e3{v2/x} = e4 e4 \psi v [app]
```

· Just write:



Lecture 4

CSE P505 August 2016 Dan Grossman

What will we do

- · Go back to math metalanguage
 - Notes on concrete syntax (relates to OCaml)
 - Define semantics with inference rules
- · Lambda encodings (show our language is mighty)
- · Define substitution precisely
- Environments

Next time??

- Small-step
- Play with continuations ("very fancy" language feature)

Lecture 4

CSE P505 August 2016 Dan Grossman

51

Where we're going

- · Done: large-step for untyped lambda-calculus
 - CBV and CBN
 - Note: infinite number of other "reduction strategies"
 - Amazing fact: all equivalent if you ignore termination!
- · Now other semantics, all equivalent to CBV:
 - With environments (in OCaml to prep for Homework 3)
 - Basic small-step (easy)
 - Contextual semantics (similar to small-step)
 - · Leads to precise definition of continuations

Lecture 4

CSE P505 August 2016 Dan Grossman

52

50

Slide repeat...

Environments

- Rather than substitute, let's keep a map from variables to values
 - Called an environment
 - Like IMP's heap, but immutable and 1 not enough
- · So a program "state" is now exp and environment
- A function body is evaluated under the environment where it was defined!
 - Use closures to store the environment
 - See also Lecture 1

Lecture 4 CSE P505 August 2016 Dan Grossman

Lecture 4

CSE P505 August 2016 Dan Grossman

No more substitution

```
type exp = Var of string
         | Lam of string * exp
         | Apply of exp * exp
         | Closure of string * exp * env
and env = (string * exp) list
let rec interp env e =
 match e with
  Var s -> List.assoc s env (* do the lookup *)
 | Lam(s,e2) -> Closure(s,e2,env) (* store env! *)
 | Closure _ -> e (* closures are values *)
 | Apply(e1,e2) ->
    let v1 = interp env e1 in
   let v2 = interp env e2 in
    match v1 with
     Closure(s,e3,env2) -> interp((s,v2)::env2) e3
     | _ -> failwith "impossible"
                CSE P505 August 2016 Dan Grossman
```

Worth repeating

- · A closure is a pair of code and environment
 - Implementing higher-order functions is not magic or run-time code generation
- · An okay way to think about OCaml
 - Like thinking about OOP in terms of vtables
- · Need not store whole environment of course
 - See Homework 3

Lecture 4

CSE P505 August 2016 Dan Grossman

56

What will we do

- · Go back to math metalanguage
 - Notes on concrete syntax (relates to OCaml)
 - Define semantics with inference rules
- · Lambda encodings (show our language is mighty)
- · Define substitution precisely
 - And revisit function equivalences
- Environments

Next time??

- · Small-step
- Play with continuations ("very fancy" language feature)

Lecture 4

CSE P505 August 2016 Dan Grossman