HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 12:
First-class functions

Stephen Chong

https://www.seas.harvard.edu/courses/cs153

Contains content from lecture notes by Steve Zdancewic and Greg Morrisett

Announcements

- Mid-course eval
  - https://forms.gle/zHNzSbyD7zwuVZB76
  - Please fill in by end of Friday Oct 11
- HW3 LLVMlite out
  - Due Tuesday Oct 15
- HW4 Oat v1 out
  - Due Tuesday Oct 29

Today

- Nested functions
  - Substitution semantics
  - Environment semantics and closures

"Functional" languages

- In functional languages, functions are first-class values
  - E.g., ML, Haskell, Scheme, Python, C#, Java 8, Swift
- Functions can be passed as arguments (e.g., `map` or `fold`)
- Functions can be returned as values (e.g., `compose`)
- Functions nest: inner function can refer to variables bound in the outer function

```
let add = fun x -> (fun y -> y+x)
let inc = add 1 (* = fun y -> y + 1 *)
let dec = add -1 (* = fun y -> y + -1 *)

let compose = fun f -> fun g -> fun x -> f(g x)
let id = compose inc dec
 (* = fun x -> inc(dec x) *)
 (* = fun x -> (fun y -> y+1)((fun y -> y-1) x) *)
 (* = fun x -> (fun y -> y+1)(x-1)) *)
 (* = fun x -> (x-1)+1 *)
```

- How do we implement such functions?
  - in an interpreter? in a compiled language?

Making Sense of Nested Functions

- Let's consider what are the right semantics for nested functions
  - We will look at a simple semantics first, and then get to an equivalent semantics that we can implement efficiently

(Untyped) Lambda Calculus

- The lambda calculus is a minimal programming language.
  - Note: we're writing `(fun x -> e)` lambda-calculus notation: $\lambda x.\ e$
- It has variables, functions, and function application.
  - That's it!
  - It's Turing Complete.
  - It's the foundation for a lot of research in programming languages.
  - Basis for "functional" languages like Scheme, ML, Haskell, etc.
- We will add integers and addition to make it a bit more concrete...

(Untyped) Lambda Calculus

- Abstract syntax in OCaml:

```
type exp =
  | Var of var          (* variables              *)
  | Fun of var * exp    (* functions: fun x -> e  *)
  | App of exp * exp    (* function application   *)
  | Int of int          (* integer constants      *)
  | Plus of exp * exp   (* addition               *)
```

- Concrete syntax:

```
exp ::=
      | x                variables
      | fun x -> exp     functions
      | exp₁ exp₂        function application
      | i                integer constants
      | exp₁ + exp₂      addition
      | ( exp )          parentheses
```

Substitution-Based Semantics

```
let rec eval (e:exp) =
  match e with
   | Int i -> Int i
   | Plus(e1,e2) ->
       (match eval e1, eval e2 with
          | Int i,Int j -> Int(i+j))
   | Var x -> error ("Unbound variable "^x)
   | Lambda(x,e) -> Lambda(x,e)
   | App(e1,e2) ->
       (match eval e1, eval e2 with
          (Lambda(x,e),v) ->
    eval (subst v x e)))
```

Replace formal argument **x** with actual argument **v**

Substitution-Based Semantics

```
let rec subst (v:exp) (x:var) (e:exp) =
  match e with
   | Int i -> Int i
   | Plus(e1,e2) -> Plus(subst v x e1, subst v x e2)
   | Var y -> if y = x then v else Var y
   | Lambda(y,e') ->
       if y = x then Lambda(y,e')
                else Lambda(y,subst v x e')
   | App(e1,e2) -> App(subst v x e1, subst v x e2)
```

Slight simplification: assumes that all variable names in program are distinct.

Substitution-Based Semantics

```
let rec subst (v:exp) (x:var) (e:exp) =
  match e with
   | Int i -> Int i
   | Plus(e1,e2) -> Plus(subst v x e1, subst v x e2)
   | Var y -> if y = x then v else Var y
   | Lambda(y,e') ->
       if y = x then Lambda(y,e')
                else Lambda(y,subst v x e')
   | App(e1,e2) -> App(subst v x e1, subst v x e2)
```

- In math: substitution function $e\{v/x\}$

$i\{v/x\}$	$= i$	
$(e1 + e2)\{v/x\}$	$= e1\{v/x\} + e2\{v/x\}$	(substitute everywhere)
$x\{v/x\}$	$= v$	(replace the free x by v)
$y\{v/x\}$	$= y$	(assuming $y \neq x$)
$(\text{fun } x \text{ -> exp})\{v/x\}$	$= (\text{fun } x \text{ -> exp})$	(x is bound in exp)
$(\text{fun } y \text{ -> exp})\{v/x\}$	$= (\text{fun } y \text{ -> exp}\{v/x\})$	(assuming $y \neq x$)
$(e1\ e2)\{v/x\}$	$= (e1\{v/x\}\ e2\{v/x\})$	(substitute everywhere)

Example

```
((fun x -> fun y -> x + y) 3) 4
```

```
eval App(App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3),Int 4)

eval App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3)      eval Int 4

eval Lambda(x,Lambda(y,Plus(Var x,Var y))      eval Int 3
```

Example

```
((fun x -> fun y -> x + y) 3) 4
```

```
eval App(App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3),Int 4)
```

```
eval App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3)        eval Int 4
```

```
Lambda(x,Lambda(y,Plus(Var x,Var y))        Int 3
```

Example

```
((fun x -> fun y -> x + y) 3) 4
```

```
eval App(App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3),Int 4)
```

```
eval App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3)
```

```
eval Int 4
```

```
eval subst x (Int 3) Lambda(y,Plus(Var x,Var y))
```

```
eval  Lambda(y,Plus(Int 3,Var y))
```

Example

```
((fun x -> fun y -> x + y) 3) 4

eval App(App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3),Int 4)

eval App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3)        eval Int 4

Lambda(y,Plus(Int 3,Var y))
```

Example

```
((fun x -> fun y -> x + y) 3) 4
```

```
eval App(App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3),Int 4)
```

```
Lambda(y,Plus(Int 3,Var y))
```

```
eval Int 4
```

Example

```
((fun x -> fun y -> x + y) 3) 4
```

```
eval App(App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3),Int 4)
```

```
Lambda(y,Plus(Int 3,Var y))                    Int 4
```

Example

```
((fun x -> fun y -> x + y) 3) 4

eval App(App(Lambda(x,Lambda(y,Plus(Var x,Var y)),Int 3),Int 4)
        |
  eval subst y (Int 4) Plus(Int 3,Var y)

  eval Plus(Int 3,Int 4)
```

More formally

- Define evaluation relation $e \Downarrow v$
  - Means expression e evaluates to value v
  - E.g.,
    - 35+7 $\Downarrow$ 42
    - (fun x -> x + 7) 35 $\Downarrow$ 42
    - 42 $\Downarrow$ 42
    - (fun f -> fun x -> f (f x)) (fun i -> i+1) $\Downarrow$
      fun x -> (fun i -> i+1) ((fun i -> i+1) x)
- Define using **inference rules**
  - Compact concise way of specifying language properties, analyses, etc.
  - We will see more of these soon...

Inference Rules for Evaluation

Axiom

$$\frac{}{\texttt{i} \Downarrow \texttt{i}}$$

$$\frac{\texttt{e1} \Downarrow \texttt{i1} \qquad \texttt{e2} \Downarrow \texttt{i2} \qquad \texttt{i} = \texttt{i1} + \texttt{i2}}{\texttt{e1 + e2} \Downarrow \texttt{i}}$$

Note: mathematical addition vs. AST for addition

Premises

$$\frac{}{\texttt{fun x -> e} \Downarrow \texttt{fun x -> e}}$$

$$\frac{\texttt{e1} \Downarrow \texttt{fun x -> e} \qquad \texttt{e2} \Downarrow \texttt{v} \qquad \texttt{e\{v/x\}} \Downarrow \texttt{w}}{\texttt{e1 e2} \Downarrow \texttt{w}}$$

Conclusion

- Inference rule
  - If the premises are true, then the conclusion is true
  - An **axiom** is a rule with no premises
  - Inference rules can be **instantiated** by replacing **metavariables** (e, e1, e2, x, i, ...) with expressions, program variables, integers, as appropriate.

Proof tree

$$\frac{}{\text{i} \Downarrow \text{i}} \qquad \frac{\text{e1} \Downarrow \text{i1} \quad \text{e2} \Downarrow \text{i2} \quad \text{i = i1 + i2}}{\text{e1 + e2} \Downarrow \text{i}}$$

$$\frac{}{\text{fun x -> e} \Downarrow \text{fun x -> e}} \qquad \frac{\text{e1} \Downarrow \text{fun x -> e} \quad \text{e2} \Downarrow \text{v} \quad \text{e\{v/x\}} \Downarrow \text{w}}{\text{e1 e2} \Downarrow \text{w}}$$

- Instantiated rules can be combined into **proof trees**
  - e ⇓ v holds if and only if there is a finite proof tree constructed from correctly instantiated rules, and leaves of the tree are axioms

$$\frac{\dfrac{}{\texttt{(fun x->x+35)} \Downarrow \texttt{(fun x->x+35)}} \quad \dfrac{\dfrac{}{\texttt{4} \Downarrow \texttt{4}} \quad \dfrac{}{\texttt{3} \Downarrow \texttt{3}}}{\texttt{(4+3)} \Downarrow \texttt{7}} \quad \dfrac{\dfrac{}{\texttt{7} \Downarrow \texttt{7}} \quad \dfrac{}{\texttt{35} \Downarrow \texttt{35}}}{\texttt{(7+35)} \Downarrow \texttt{42}}}{\texttt{(fun x -> x + 35) (4+3)} \Downarrow \texttt{42}}$$

Problems with Substitution Semantics

- `subst` crawls over expression and replaces variable with value
- Then `eval` crawls over expression
- So `eval (subst v x e)` is not very efficient
- Why not do substitution at the same time as we do evaluation?
- Modify `eval` to use an **environment**: a map from variables to the values

First Attempt

```
type value = Int_v of int
type env = (string * value) list

let rec eval (e:exp) (env:env) : value =
  match e with
   | Int i -> Int_v i
   | Var x -> lookup env x
   | Lambda(x,e) -> Lambda(x,e)
   | App(e1,e2) ->
       (match eval e1 env, eval e2 env with
         | Lambda(x,e'), v -> eval e' ((x,v)::env))
```

- Doesn't handle nested functions correctly!
- E.g., `(fun x -> fun y -> y+x) 1` evaluates to `fun y -> y+x`
- Don't have binding for `x` when we eventually apply this function!

Second Attempt

```
type value = Int_v of int
type env = (string * value) list

let rec eval (e:exp) (env:env) : value =
  match e with
   | Int i -> Int_v i
   | Var x -> lookup env x
   | Lambda(x,e) -> Lambda(x,subst env e)
   | App(e1,e2) ->
       (match eval e1 env, eval e2 env with
        | Lambda(x,e'), v -> eval e' ((x,v)::env))
```

- Need to replace free variables of nested functions using environment where nested function defined
- But now we are using a version of `subst` again...

Closures

- Instead of doing substitution on nested functions when we reach the lambda, we can instead make a promise to finish the substitution if the nested function is ever applied
- Instead of

```
  | Lambda(x,e’) -> Lambda(x,subst env e’)
```

  we will have, in essence,

```
  | Lambda(x,e’) -> Promise(env, Lambda(x, e’))
```

  - Called a **closure**
- Need to modify rule for application to expect environment

Closure-based Semantics

```
type value = Int_v of int
             | Closure_v of {env:env, body:var*exp}
and  env = (string * value) list

let rec eval (e:exp) (env:env) : value =
  match e with
   | Int i -> Int_v i
   | Var x -> lookup env x
   | Lambda(x,e) -> Closure_v{env=env, body=(x,e)}
   | App(e1,e2) ->
       (match eval e1 env, eval e2 env with
        | Closure_v{env=cenv, body=(x,e')}, v ->
                eval e' ((x,v)::cenv))
```

Inference rules

$$\frac{}{\Gamma \vdash i \Downarrow i}$$

$$\frac{\Gamma(x) = v}{\Gamma \vdash x \Downarrow v}$$

$$\frac{\Gamma \vdash e1 \Downarrow i1 \quad \Gamma \vdash e2 \Downarrow i2 \quad i = i1 + i2}{\Gamma \vdash e1 + e2 \Downarrow i}$$

$$\frac{}{\Gamma \vdash \text{fun } x \text{ -> } e \Downarrow (\Gamma, \text{fun } x \text{ -> } e)}$$

$$\frac{\Gamma \vdash e1 \Downarrow (\Gamma_c, \text{fun } x \text{ -> } e) \quad \Gamma \vdash e2 \Downarrow v \quad \Gamma_c[x \mapsto v] \vdash e \Downarrow w}{\Gamma \vdash e1\ e2 \Downarrow w}$$