# Closures

Mooly Sagiv

**Michael Clarkson, Cornell CS 3110 Data Structures and Functional Programming**

# Formal Semantics of Functional Programs

- Compile into typed lambda calculus
- Small step operational semantics
  - Environment $\times$ Expression $\Rightarrow$ Environment $\times$ Expression

# Essential OCcaml sublanguage

$$
\begin{aligned}
e ::= \; & c \\
& \mid x \\
& \mid (e_1, \ldots, e_n) \\
& \mid e_1 \; e_2 \\
& \mid \text{fun } x \text{ -> } e \\
& \mid \text{let } x = e_1 \text{ in } e_2 \\
& \mid \text{match } e_0 \text{ with } p_i \text{ -> } e_i
\end{aligned}
$$

# Evaluation of Expression

- Expressions evaluate to values in a dynamic environment
  - env :: e - - > v
- Evaluation is meaningless if expression does not type check
- Values are a *syntactic subset of expressions:*

```
v ::= c |  (v_1, …, v_n)
     | fun x -> e
```

# Dealing with Functions as Values

- Anonymous functions fun x-> e are values
  - **env :: (fun x -> e) --> (fun x -> e)**

# Evaluating "let expressions"

- To evaluate **let x = $e_1$ in $e_2$** in environment **env**:
  1. **Evaluate** the binding expression **$e_1$** to a value **$v_1$** in environment **env**
     **env :: $e_1$ --> v**
  2. **Extend** the environment to bind **x** to **$v_1$**
     **env' = env [x $\mapsto$ $v_1$]**
     (newer bindings temporarily *shadow older bindings)*
  3. **Evaluate** the body expression **$e_2$** to a value **$v_2$** in environment **env**'
     **env' :: $e_2$ --> $v_2$**
  4. **Return $v_2$**

# Evaluating Function Application take 1

- To evaluate $\mathbf{e_1}$ $\mathbf{e_2}$ in environment **env**
  1. **Evaluate** $\mathbf{e_2}$ to a value $\mathbf{v_2}$ in environment **env**
     **env :: $\mathbf{e_2}$ --> $\mathbf{v_2}$**
     *Note: right to left order, like tuples, which matters in the presence of side effects*
  2. **Evaluate** $\mathbf{e_1}$ to a value $v_1$ in environment **env**
     **env :: $\mathbf{e_1}$ --> $\mathbf{v_1}$**
     *Note that $\mathbf{v_1}$ must be a function value **fun x -> e***
  3. **Extend** environment to bind formal parameter **x** to actual value $\mathbf{v_2}$
     **env' = env [x↦$\mathbf{v_2}$]**
  4. **Evaluate** body **e** to a value v in environment **env'**
     **env' :: e --> v**
  5. **Return v**

# Evaluating Function Application take 1

if **env :: e2 --> v2**
and **env :: e1 --> (fun x -> e)**
and **env[x↦v2] :: e --> v**
then **env :: e1 e2 --> v**

# Evaluating Function Application Simple Example

let f = fun x -> x in f 0

**$env_0$ =[]**

1. Evaluate binding expression **fun x->x** to a value in empty environment **$env_0$**
2. Extend environment to bind **f** to **fun x->x**

   **$env_1$=$env_0$[f ↦ fun x -> x ] = [f ↦ fun x -> x]**

3. Evaluate let-body expression **f 0** in environment **$env_1$**

   **$env_1$ :: f 0 --> v1**

   1. Evaluate **0** to a value 0 in environment $env_1$
   2. Evaluate **f** to ***fun x -> x***
   3. Extend environment to bind formal parameter x to actual value 0

      **$env_2$= $env_1$[x ↦0] = [f ↦ …, x ↦0]**

   4. Evaluate the function body **x** in environment **$env_2$**
      **$env_2$ :: x--> 0**

      **$env_2$ :: x--> 0**

4. Return **0**

# Hard Example

```
let x = 1 in
  let f = fun y -> x in
    let x = 2 in
      f 0
```

1. What is the result of the expression?
2. What does OCaml say?
3. What do you say?

# Hard Example Ocaml

```
let x = 1 in
   let f = fun y -> x in
     let x = 2 in
         f 0
warning 26: x unused variable
:- int 1
```

# Hard Example "C"

```
{
  int x = 1
  { int f(int y)
     {
       return x ;
     }
     {
        int x = 2;
        printf("%d", f(0)) ;
     }
}
```

# Why different answers?

- Two different rules for variable scope
  - Rule of dynamic scope (lisp)
  - Rule of lexical (static) scope (Ocaml, Javascript, Scheme, …)

# Dynamic Scope

- **Rule of dynamic scope:** The body of a function is evaluated in the current dynamic environment at the time the function is called, not the old dynamic environment that existed at the time the function was defined
- Use latest binding of **x**
- Thus return 2

# Lexical Scope

- **Rule of lexical scope:** The body of a function is evaluated in the old dynamic environment that existed at the time the function was **defined**, not the current environment when the function is called
- Causes OCaml to use earlier binding of **x**
- Thus return 1

# Scope

- **Rule of dynamic scope**: The body of a function is evaluated in the current dynamic environment at the time the function is called, not the old dynamic environment that existed at the time the function was defined
- **Rule of lexical scope:** The body of a function is evaluated in the old dynamic environment that existed at the time the function was **defined,** not the current environment when the function is called
- *In both, environment is extended to map formal parameter to actual value*
- Why would you want one vs. the other?

# Lexical vs. dynamic scope

- Consensus after decades of programming language design is that **lexical scope is the right choice**
- Dynamic scope is convenient in some situations
- Some languages use it as the norm (e.g., Emacs LISP, LaTeX)
- Some languages have special ways to do it (e.g., Perl, Racket)
- But most languages just don’t have it

# Why Lexical Scope (1)

- Programmer can freely change names of local variables

```
(* 1 *) let x = 1
(* 2 *) let f y =
            let x = y + 1 in
               fun z -> x+y+z
(* 3 *) let x = 3
(* 4 *) let w = (f 4) 6
```

```
(* 1 *) let x = 0
(* 2 *) let f y =
            let q = y + 1 in
               fun z -> q+y+z
(* 3 *) let x = 3
(* 4 *) let w = (f 4) 6
```

# Why Lexical Scope (2)

- Type checker can prevent run-time errors

```
(* 1 *) let x = 1
(* 2 *) let f y =
          let x = y + 1 in
            fun z -> x+y+z
(* 3 *) let x = 3
(* 4 *) let w = (f 4) 6
```

```
(* 1 *) let x = 0
(* 2 *) let f y =
          let x = y + 1 in
            fun z -> x+y+z
(* 3 *) let x = “hi”
(* 4 *) let w = (f 4) 6
```

# Exception Handling

- Resembles dynamic scope:
- **raise e** transfers control to the “most recent” exception handler
- like how dynamic scope uses “most recent” binding of variable

# Where is an exception caught?

- Dynamic scoping of handlers
  - Throw to most recent catch on run-time stack
- Dynamic scoping is not an accident
  - User knows how to handler error
  - Author of library function does not

# Implementing time travel (lexical)

Q How can functions be evaluated in old environments?

A The language implementation keeps them around as necessary

A function value is really a data structure that has two parts:

1. The code
2. The environment that was current when the function was defined

1. Gives meaning to all the *free variables of the function body*

– Like a “pair”

• But you cannot access the pieces, or directly write one down in the language syntax

• All you can do is call it

– This data structure is called a ***function closure***

A function application:

– evaluates the code part of the closure

– in the environment part of the closure extended to bind the function argument

# Hard Example Revisited

```
[1] let x = 1 in
[2]  let f = fun y -> x in
[3]    let x = 2 in
[4]       let z = f 0 in z
```

With lexical scope:

- Line 2 creates a closure and binds **f** to it:

– Code: **fun y -> x**

– Environment: **$[x \mapsto 1]$**

- Line 4 calls that closure with **0** as argument

– In function body, **y** maps to **0** and **x** maps to **1**

- So **z** is bound to **1**

# Another Example

```
[1] let x = 1 in
[2]  let f  y  = x + y in
[3]    let x = 3 in
[4]      let y = 4 in
[5]        let z = f (x + y) in z
```

With lexical scope:

1. Creates a closure and binds **f** to it:

– Code: **fun y -> x + y**

– Environment: **[x↦1]**

2. Line 5 env =[x ↦3, y ↦ 4]

3. Line 5 calls that closure with **x+y=7** as argument

– In function body, **x** maps to **1**

• So **z** is bound to **8**

# Another Example

```
[1] let x = 1 in
[2]  let f  y  = x + y in
[3]    let x = 3 in
[4]      let y = 4 in
[5]        let z = f (x + y) in z
```

With dynamic scope:

1. Line 5 env =[x ↦3, y ↦ 4]
2. Line 5 calls that closure with **x+y=7** as argument

– In function body, **x** maps to **3**, so **x+y** maps to **10**
   Note that argument **y** shadows **y** from line 4

• So **z** is bound to **10**

# Closure Notation

**<<code, environment>>**

**<<fun y -> x+y, [x↦1>>**

*With lexical scoping, well-typed programs are guaranteed never to have any variables in the code body other than function argument and variables bound by closure environment*

# Evaluating Function Application take 2

- To evaluate $\mathbf{e_1}$ $\mathbf{e_2}$ in environment **env**
  1. **Evaluate** $\mathbf{e_2}$ to a value $v_2$ in environment env
     **env :: $\mathbf{e_2}$ --> $\mathbf{v_2}$**
     *Note: right to left order, like tuples, which matters in the presence of side effects*
  2. **Evaluate** $\mathbf{e_1}$ to a value $v_1$ in environment env
     **env :: $\mathbf{e_1}$ --> $\mathbf{v_1}$**
     *Note that **$\mathbf{v_1}$** must be a closure with function value **fun x -> e** and environment **env'***
  3. **Extend** environment to bind formal parameter **x** to actual value $\mathbf{v_2}$
     **env'' = env' [x↦$\mathbf{v_2}$]**
  4. **Evaluate** body **e** to a value v in environment **env''**
     **env'' :: e --> v**
  5. **Return v**

# Evaluating Function Application take 2

if **env :: e2 --> v2**
and **env :: e1 -->**
**<<fun x -> e, env’>>**
and **env’[x↦v2] :: e -->** v
then **env :: e1 e2 --> v**

# Evaluating Anonymous Function Application take 2

Anonymous functions **fun x->** e are closures

**env :: (fun x -> e) -->**

**<<fun x -> e, env>>**

# Why are Closure useful?

- Hides states in an elegant way
- Useful for
  - Implementing objects
  - Web programming
  - Operated system programming
  - Emulating control flow
  - …

# Simple Example

```
let startAt  x =
   let incrementBy y = x + y
      in incrementBy
val startAt : int -> int -> int = <fun>
```

```
let closure1 = startAt 3
val closure1 : int -> int = <fun>
```

```
let closure2 = startAt 5
val closure2 : int -> int = <fun>
```

```
closure1 7
:- int =10
```

```
closure2 9
:- int =14
```

# Another Example

```
let derivative f dx =
  fun x -> f (x + dx) – f x / dx
val derivative : (int -> int) -> int -> int -> int = <fun>
```

# Implementation Notes

- Duration of closure can be long
  - Usually implemented with garbage collection
- It is possible to support lexical scopes without closure (using stack) if one of the following is forbidden:
  - Nested scopes (C, Java)
  - Returning a function (Algol, Pascal)

# Essential OCcaml sublanguage

```
e ::= c
    | x
    | e1 e2
    | fun x -> e
    | let x = e1 in e2
    | match e0 with pi -> ei
```

# Essential OCcaml sublanguage+rec

**e ::= c**
**| x**
**| ($e_1$, …, $e_n$)**
**| $e_1$ $e_2$**
**| fun x -> e**
**| let x = $e_1$ in $e_2$**
**| match $e_0$ with $p_i$ -> $e_i$**
**| let rec f x = e1 in e2**

# **let rec** Evaluation

- How to handle
  **let rec f x =** $\mathbf{e_1}$ **in** $\mathbf{e_2}$

# **let rec** Evaluation

- To evaluate **let rec f x = $e_1$ in $e_2$** in environment **env**
  - *don't evaluate the binding expression* $e_1$

1. Extend the environment to bind f to a *recursive closure*
   **env' = env [f ↦<<f, fun x -> $e_1$, env>>]**
2. Evaluate the body expression $e_2$ to a value $v_2$ in environment **env'**
   **env' :: $e_2$ --> $v_2$**
3. Return $v_2$

# Closure in OCaml

- *Closure conversion is an important phase of* compiling many functional languages
- Expands on ideas we've seen here
- Many optimizations possible
- Especially, better handling of recursive functions

# Closures in Java

- Nested classes can simulate closures
- Used everywhere for Swing GUI!
- http://docs.oracle.com/javase/tutorial/uiswing/events/
- generalrules.html#innerClasses
- Java 8 adds higher-order functions and closures
- Can even think of OCaml closures as resembling Java objects:
  - closure has a single method, the code part, that can be Invoked
  - closure has many fields, the environment part, that can be accessed

# Closures in C

- In C, a *function pointer is just a code pointer, period,* No environment
- To simulate closures, a common **idiom:**
  - Define function pointers to take an extra, explicit environment argument
  - But without generics, no good choice for type of list elements or the environment
- Use **void* and various type casts…**
- From Linux kernel:
  - http://lxr.free-electrons.com/source/include/linux/kthread.h#L13

# Summary

- Lexical scoping is natural
- Permit general programming style
  - Works well with higher order functions
- Well understood
- Implemented with closures
  - But requires long lived objects
- Integrated into many programming languages
- Some surprises (javascript)

# Summary (Ocaml)

- Functional programs provide concise coding
- Compiled code compares with C code
- Successfully used in some commercial applications
  - F#, ERLANG, Jane Street
- Ideas used in imperative programs
- Good conceptual tool
- Less popular than imperative programs