



CS 311O

The Environment Model

Prof. Clarkson

Fall 2019

Today's music: Selections from *Doctor Who* soundtracks by Murray Gold

CLICKER QUESTION 1

Review

Previously in 3110:

- Interpreters
- Substitution model

Today:

- Small-step vs. big-step evaluation
- Environment model
- Dynamic vs. static scope

Small-step evaluation

Small (single) step relation: $e \rightarrow e'$

- $(10 + 1) + (5 + 6) \rightarrow 11 + (5 + 6)$
- $11 + (5 + 6) \rightarrow 11 + 11$
- $11 + 11 \rightarrow 22$
- $22 \nrightarrow$

Multistep relation: $e \rightarrow^* e'$

- $(10 + 1) + (5 + 6) \rightarrow^* (10 + 1) + (5 + 6)$
- $(10 + 1) + (5 + 6) \rightarrow^* 11 + (5 + 6)$
- $(10 + 1) + (5 + 6) \rightarrow^* 11 + 11$
- $(10 + 1) + (5 + 6) \rightarrow^* 22$

The final value is what OCaml actually gives us...



big-step relation

the `eval` function we implemented

Big-step evaluation

forget about intermediate steps



$$e \Rightarrow v$$

Big and small should be consistent:

for all expressions e and values v ,

$$e \Rightarrow v \text{ if and only if } e \rightarrow^* v$$

BIG-STEP SEMANTICS

SimPL

$e ::= x \mid i \mid b$
 $\mid e1 \text{ bop } e2$
 $\mid \text{let } x = e1 \text{ in } e2$
 $\mid \text{if } e1 \text{ then } e2 \text{ else } e3$

$\text{bop} ::= + \mid * \mid <=$

Semantics

$v \Rightarrow v$

$e1 + e2 \Rightarrow v$

if $e1 \Rightarrow v1$

and $e2 \Rightarrow v2$

and v *is the result of primitive operation* $v1 + v2$

$x \not\Rightarrow$

Semantics

let $x = e_1$ in $e_2 \Rightarrow v_2$
 if $e_1 \Rightarrow v_1$
 and $e_2\{v_1/x\} \Rightarrow v_2$

if e_1 then e_2 else $e_3 \Rightarrow v_2$
 if $e_1 \Rightarrow \text{true}$
 and $e_2 \Rightarrow v_2$

if e_1 then e_2 else $e_3 \Rightarrow v_3$
 if $e_1 \Rightarrow \text{false}$
 and $e_3 \Rightarrow v_3$

ENVIRONMENT MODEL

Substitution could be slow

```
let x = 0 in 42
```

Imagine a large block of code instead of just [42]

```
let b = ... in
```

```
let x = ... in
```

```
if b then x + 1 else x - 1
```

Imagine large blocks of code instead of just [x+1] and [x-1]

Dictionaries are fast

```
let x = 42 in
```

```
let y = 3110 in
```

```
x + y
```

x	42
y	3110

Dynamic
environment

x

42

Dynamic environment

- Maps variable names to values in current scope
- Implements a kind of **lazy substitution**

```
let x = 0 in 42
```

x	0
---	---

```
let b = ... in
```

```
let x = ... in
```

```
if b then x + 1 else x - 1
```

Doesn't matter how large a block of code

b	?
x	?

machine
configuration

$$\langle env, e \rangle \Rightarrow v$$

environment-model big-step relation

Values

$$\langle \text{env}, v \rangle \Rightarrow v$$

Binary operators

$\langle \text{env}, e1 + e2 \rangle \Rightarrow v$

if $\langle \text{env}, e1 \rangle \Rightarrow v1$

and $\langle \text{env}, e2 \rangle \Rightarrow v2$

and v is $v1 + v2$

If expressions

$\langle \text{env}, \text{if } e1 \text{ then } e2 \text{ else } e3 \rangle \Rightarrow v2$

if $\langle \text{env}, e1 \rangle \Rightarrow \text{true}$

and $\langle \text{env}, e2 \rangle \Rightarrow v2$

$\langle \text{env}, \text{if } e1 \text{ then } e2 \text{ else } e3 \rangle \Rightarrow v3$

if $\langle \text{env}, e1 \rangle \Rightarrow \text{false}$

and $\langle \text{env}, e3 \rangle \Rightarrow v3$

Let expressions

$\langle \text{env}, \text{let } x = e1 \text{ in } e2 \rangle \Rightarrow v2$

if $\langle \text{env}, e1 \rangle \Rightarrow v1$

and $\langle \text{env}[x \mapsto v1], e2 \rangle \Rightarrow v2$

think of this as recording the
substitution in case it is ever needed

$\text{env}[x \mapsto v]$: env with x bound to v

Variables

$$\langle \text{env}, \mathbf{x} \rangle \Rightarrow \text{env}(\mathbf{x})$$

env (x) : the value to which **env** binds **x**

FUNCTIONS

SimPL + functions

```
e ::= x | i | b
    | e1 bop e2
    | let x = e1 in e2
    | if e1 then e2 else e3
    | fun x -> e
    | e1 e2
```

```
bop ::= + | * | <=
```

Function values v1.0

Since functions are values:

$$\langle \text{env}, \text{fun } x \rightarrow e \rangle \Rightarrow \text{fun } x \rightarrow e$$

Function application rule v1.0

$\langle \text{env}, e1 \ e2 \rangle \Rightarrow v$

if $\langle \text{env}, e1 \rangle \Rightarrow \text{fun } x \rightarrow e$

and $\langle \text{env}, e2 \rangle \Rightarrow v2$

and $\langle \text{env}[x \mapsto v2], e \rangle \Rightarrow v$

CLICKER QUESTION 2

Scope: OCaml

What does OCaml say this evaluates to?

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

Scope: our semantics

What does our semantics say?

```
let x = 1 in
```

```
{x:1} let f = fun y -> x in
```

```
{x:1, f: (fun y -> x)} let x = 2 in
```

```
{x:2, f: (fun y -> x)} f 0
```

$\langle \{x:2, f: (\text{fun } y \rightarrow x) \}, f \ 0 \rangle \Rightarrow ???$

1. Evaluate **f** to a value: **fun y -> x**
2. Evaluate **0** to a value: **0**
3. Extend environment to map parameter:
{x:2, f: (fun y -> x), y:0}
4. Evaluate body **x** in that environment
5. Return **2**

2 <> 1

Why different answers?

Two different rules for variable scope:

- Rule of **dynamic scope** (our semantics so far)
- Rule of **lexical scope** (OCaml)

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.

- Causes our semantics to 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**

Lexical scope

Rule of
evaluation
exists
the current
called.

- Cause
- Thus



is
that
, not
is

Implementing time travel

Q: How can functions be evaluated in old environments?

A: The language implementation keeps old environments around as necessary

Implementing time travel

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

- The **code**, an expression **e**
- The **environment** **env** that was current when the function was defined
- We'll notate that data structure as $(|e, env|)$

$(|e, env|)$ is like a pair

- But **indivisible**: you cannot write OCaml syntax to access the pieces
- And **inexpressible**: you cannot directly write it in OCaml syntax

Closures in OCaml bytecode compiler

<https://github.com/ocaml/ocaml/search?q=kclosure>

Results in `ocaml/ocaml`

`bytecomp/instruct.ml`

OCaml

Showing the top match Last indexed on Sep 15, 2016

```
63 | Krestart
64 | Kgrab of int (* number of arguments *)
65 | Kclosure of label * int
```

`bytecomp/printinstr.ml`

OCaml

Showing the top match Last indexed on Sep 15, 2016

```
35 | Kgrab n -> fprintf ppf "\tgrab %i" n
36 | Kclosure(lbl, n) ->
37   fprintf ppf "\tclosure L%i, %i" lbl n
```

`bytecomp/instruct.mli`

OCaml

Showing the top match Last indexed on Aug 10

```
84 | Kgrab of int (* number of arguments *)
85 | Kclosure of label * int
86 | Kclosurerec of label list * int
87 | Koffsetclosure of int
```

Function values v2.0

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

$\langle \text{env}, \text{fun } x \rightarrow e \rangle$

$\Rightarrow (| \text{fun } x \rightarrow e \text{ , env} |)$

Function application rule v2.0

$\langle \text{env}, e1\ e2 \rangle \Rightarrow v$

if $\langle \text{env}, e1 \rangle \Rightarrow$

$(| \text{fun } x \rightarrow e \text{ , } \text{defenv} |)$

and $\langle \text{env}, e2 \rangle \Rightarrow v2$

and $\langle \text{defenv}[x \mapsto v2], e \rangle \Rightarrow v$

Lexical vs. dynamic scope

- Consensus after decades of programming language design is that **lexical scope is the right choice**
 - it supports the Principle of Name Irrelevance: name of variable shouldn't matter to meaning of program
 - programmers free to change names of local variables
 - type checker can prevent more run-time errors
- Dynamic scope is useful 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
- 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

Upcoming events

- [last night]: R8 due (last reflection!)
- [Tue/Wed]: MS1 demos in section
- [Thur]: MS1 due in CMS

This is closure.

THIS IS 3110