CS 3110

Lecture 7: The dynamic environment

Prof. Clarkson

Spring 2015

Today's music: "Down to Earth" by Peter Gabriel from the WALL-E soundtrack

Review

Course so far:

- Syntax and semantics of (most of) OCaml

Today:

- Different **semantics**

Question #1

How much of PS1 have you finished?

A. None
B. About 25%
C. About 50%
D. About 75%
E. I'm done!!!

Semantics

- **Dynamic semantics**
  - How expressions evaluate
  - *Dynamic*: execution is in motion
  - Evaluation rules `e --> e' --> e''`
- **Static semantics**
  - How expressions type check (among other things)
  - *Static*: execution is not yet moving
  - Type checking rules `e : t`

Dynamic semantics

Today: change our *model of evaluation*:

- **Small-step substitution model:** substitute value for variable in body of let expression & in body of function
  - What we've done doing so far
  - Good mental model, not really what OCaml does
- **Big-step environment model:** keep a data structure around that binds variables to values
  - What we'll do now
  - Also a good mental model, much closer to what OCaml really does

The core of OCaml

Essential sublanguage of OCaml:

```
e ::=  v | C e | (e1, ..., en) | e1 + e2
     | x | e1 e2
     | let x = e1 in e2
     | match e0 with pi -> ei
v ::= c | fun x -> e | C v | (v1, ..., vn)
```

Missing, unimportant: records, lists, options, declarations, patterns in function arguments and let bindings, `if`

Missing, important: `rec`

Extraneous: all we *really* need is

```
 e ::= x | e1 e2 | fun x -> e
```

Review: evaluation

- Expressions *step to* new expressions

```
e --> e1 --> e2 --> ...
```

- Long arrow means "steps to"
  - Star means reflexive, transitive closure: 0, 1, or more steps
- Values "have no further computation to do"
  - So they don't take a single step: **`v-/->`**
  - But they could take zero steps: **`v-->*v`**
- *Small-step semantics*: we model each small step the evaluation takes

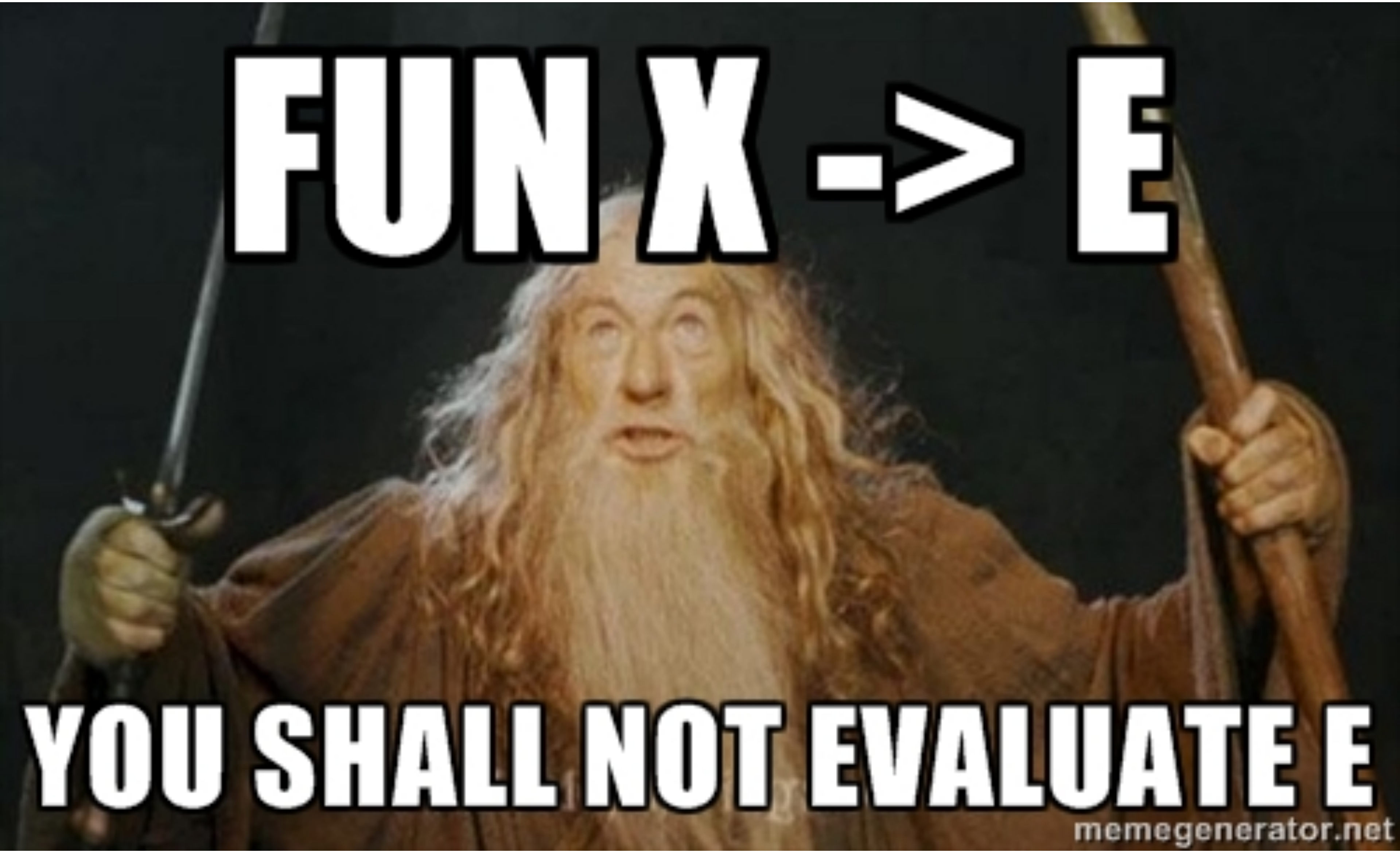
New kind of evaluation

- *Big-step semantics*: we model just the reduction from the original expression to the final value
- Suppose `e --> e' --> ... --> v`
- We'll just record the fact that `e` $\Downarrow$ `v`
  - new notation means `e` *evaluates (down) to* `v`
  - in ASCII: `e || v`

Values

- Values are already done:
  - Evaluation rule: `v || v`

- Constants are already values
  - `42` is already a value
  - `"3110"` is already a value
  - `()` is already a value
- same for `C v` and `(v1, ..., vn)`
- Functions are already values
  - heads-up: we'll reconsider this choice next lecture
  - `fun x -> e` is already a value, no matter what `e` is


FUN X -> E
YOU SHALL NOT EVALUATE E
memegenerator.net

Operator evaluation rule

```
e1 + e2 || v
  if e1 || v1
  and e2 || v2
  and v is the result of primitive
    operation v1 + v2
```

e.g.,

```
 1 + 2 || 3
3.110 *. 1.0 || 3.11
0 < 1 || true
"zar" ^ "doz" || "zardoz"
```

Tuples

To evaluate `(e1,...,en)`,

Evaluate the subexpressions:

```
e1 || v1
...
en || vn
```

Return `(v1,...,vn)`

In which case,

```
(e1,...,en) || (v1,...,vn)
```

Tuple evaluation rule

```
(e1,...,en) || (v1,...,vn)
   if e1 || v1
   and ...
   and en || vn
```

e.g.,

so `(1+1, 2+2) || (2,4)`

because `1+1 || 2` and `2+2 || 4`

Question #2

If we changed evaluation order to be **`en`** first, then then **`e2`**, then **`e1`**, which of the following expressions would evaluate to a different value?

A. **`(0+1,2*3)`**

B. **`(let x = 3 in x, “hi”)`**

C. **`((), (fun x -> x+1) 1)`**

D. All the above

E. None of the above

Question #2

If we changed evaluation order to be **`en`** first, then then **`e2`**, then **`e1`**, which of the following expressions would evaluate to a different value?

A. `(0+1,2*3)`

B. `(let x = 3 in x, “hi”)`

C. `((), (fun x -> x+1) 1)`

D. All the above

E. **None of the above**

Tuple evaluation order

Q: What order are the **`ei`** evaluated in?

A: **It doesn't matter.** Pure programs can't distinguish the order of evaluation.

Pure = no side effects: no printing, no exceptions, ...

A: OCaml language specification says order is unspecified.

A: OCaml compiler on VM does right to left: **`e2`** then **`e1`**.

```
((print_string "left\n"; 0),
 (print_string "right\n"; 1))
```

Constructors

To evaluate `C e`,

Evaluate the subexpression:

`e || v`

Return `C v`

In which case, `C e || C v`

Constructor evaluation rule

```
C e || C v
  if e || v
```

e.g.,

`Some (1+1) || Some 2`

because **`1+1 || 2`**

- Multiple arguments: **`C e1 ... en`**. Rule easily extends.
- Constructors that carry no data behave like constants
  - **`true`** is already a value
  - **`[]`** is already a value

Progress

```
e ::=  v | C e | (e1, ..., en) | e1 + e2
     | x | e1 e2
     | let x = e1 in e2
     | match e0 with pi -> ei
```

Variables

- What does a variable name evaluate to?

```
x || ???
```

- Trick question: we don't have enough information to answer it
- Need to know what value variable was *bound* to

Question #3

What do these evaluate to?

- `let x = 2 in x+1`
- `(fun x -> x+1) 2`
- `match 2 with x -> x+1`

A. 2, 2, and 2

B. 3, 3, and 3

C. 3, 2, and 3

D. 3, 3, and 2

E. 2, 3, and 3

Question #3

What do these evaluate to?

- `let x = 2 in x+1`
- `(fun x -> x+1) 2`
- `match 2 with x -> x+1`

A. 2, 2, and 2

B. 3, 3, and 3

C. 3, 2, and 3

D. 3, 3, and 2

E. 2, 3, and 3

Variables

- What does a variable name evaluate to?

  `x || ???`

- Trick question: we don't have enough information to answer it
- Need to know what value variable was *bound* to
  - e.g., **`let x = 2 in x+1`**
  - e.g., **`(fun x -> x+1) 2`**
  - e.g., **`match 2 with x -> x+1`**
  - All evaluate to **3**, but we reach a point where we need to know binding of **`x`**
- Until now, we've never needed this, because we always substituted before we ever get to a variable name

Variables

- OCaml doesn't actually do substitution
  - `(fun x -> 42) 0`
  - waste of runtime resources to do substitution inside 42
- Instead, OCaml lazily substitutes by maintaining *dynamic environment*

Dynamic environment

- Set of bindings of all current variables
- Changes throughout evaluation:
  - No bindings at $:

```
$ let x = 42 in
  let y = "3110" in
  e
```

  - One binding {x=42} at $:

```
  let x = 42 in
$ let y = "3110" in
  e
```

  - Two bindings {x=42,y="3110"} at $:

```
  let x = 42 in
  let y = "3110" in
$ e
```

Variable evaluation

To evaluate `x` in environment **`env`**

Look up value **`v`** of **`x`** in **`env`**

Return `v`

Type checking guarantees that variable is bound, so we can't ever fail to find a binding in dynamic environment

Variable evaluation rule

```
env :: x || v
  if v = env(x)
```

New notation:

- **`env :: e || v`**
  - meaning: in dynamic environment **`env`**, expression **`e`** evaluates down to value **`v`**
- **`env(x)`**
  - meaning: the value to which **`env`** binds **`x`**

Redo: rules with environment

Values:

```
env :: v || v
```

Operators:

```
env :: e1 + e2 || v
  if env :: e1 || v1
  and env :: e2 || v2
  and v is the result of primitive operation v1+v2
```

Tuples:

```
env :: (e1,...en) || (v1,...vn)
  if env :: e1 || v1
  and ...
  and env :: en || vn
```

Constructors:

```
env :: C e || C v
  if env :: e || v
```

Why the same environment for each component of tuple?

Scope

- Bindings are in effect only in the *scope* (the "block") in which they occur
- Exactly what you're used to from (say) Java
- Bindings inside elements of tuples are not in scope outside that element
  - **`((let x = 1 in x+1), (let y=2 in y+2))`**
  - **`x`** is not in scope in second component
  - **`y`** is not in scope in first component
  - so dynamic environment stays the same from one component to another
    - **`env :: ei || vi`**

Progress

```
e ::=  v | C e | (e1, ..., en) | e1 + e2
     | x | e1 e2
     | let x = e1 in e2
     | match e0 with pi -> ei
```

Let expressions

To evaluate `let x = e1 in e2` in environment **`env`**

Evaluate the binding expression **`e1`** to a value **`v1`** in environment **`env`**

```
env :: e1 || v1
```

Extend the environment to bind **`x`** to **`v1`**

```
env' = env + {x=v1}
```

Evaluate the body expression **`e2`** to a value **`v2`** in environment **`env'`**

```
env' :: e2 || v2
```

Return `v2`

Let expression evaluation rule

```
env :: let x=e1 in e2 || v2
  if env :: e1 || v1
  and env+{x=v1} :: e2 || v2
```

Example:

```
  {} :: let x = 42 in x || 42
```

Why? Because...

- `{} :: 42 || 42`
- and `{}+{x=42} :: x || 42`
  - Why? because if **`env`** is **`{x=42}`** then **`env(x)=42`**

Initial environment

- Can add an entire file's worth of bindings to the dynamic environment with **`open Name`**
  - You've been doing that in unit test files
- OCaml always does **`open Pervasives`** at the beginning
  - **`(+), (=), int_of_string, (@), print_string, fst, …`**
  - The environment is never really empty
    - it's always polluted? :)
  - But we write **{ }** anyway

Extending the environment

- What does **`env+{x=v}`** really mean?
- Illuminating example:

```
let x = 0 in
let x = 1 in
x
|| 1
```

- Environment extension can't just be set union
  - We'd get **`{x=0,x=1}`** and now we don't know what **`x`** is!
- Instead inner binding *shadows* outer binding
  - Casts its shadow over it; temporarily replaces it
- Environments at particular places (abuse OCaml syntax here):

```
let x = ({} 0) in
({x=0} let x = 1 in
  ({x=1} x))
```

Question #4

```
let x = 0 in
  x + (let x = 1 in x)
|| ???
```

A. 0
B. 1
C. 2
D. unspecified by language
E. none of the above

Question #4

```
let x = 0 in
  x + (let x = 1 in x)
|| ???
```

A. 0
B. 1
C. 2
D. unspecified by language
E. none of the above

Question #5

```
let x = 0 in
   (let x = 1 in x) + x
|| ???
```

A. 0
B. 1
C. 2
D. unspecified by language
E. none of the above

Question #5

```
let x = 0 in
   (let x = 1 in x) + x
|| ???
```

A. 0
B. 1
C. 2
D. unspecified by language
E. none of the above

Shadowing is not assignment

```
let x = 0 in
  x + (let x = 1 in x)
|| 1

let x = 0 in
  (let x = 1 in x) + x
|| 1
```

Progress

```
e ::=  v | C e | (e1, ..., en) | e1 + e2
     | x | e1 e2
     | let x = e1 in e2
     | match e0 with pi -> ei
```