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 $e \rightarrow e' \rightarrow e''$

• **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:

\[
\begin{align*}
e & ::= \ v \mid C \ e \mid (e_1, \ldots, e_n) \mid e_1 + e_2 \\
& \quad \mid x \mid e_1 \ e_2 \\
& \quad \mid \text{let } x = e_1 \text{ in } e_2 \\
& \quad \mid \text{match } e_0 \text{ with } \pi \rightarrow e_i \\
\end{align*}
\]

\[
\begin{align*}
v & ::= \ c \mid \text{fun } x \rightarrow e \mid C \ v \mid (v_1, \ldots, v_n) \\
\end{align*}
\]

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

**Missing, important:** **rec**

**Extraneous:** all we really need is

\[
\begin{align*}
e & ::= x \mid e_1 \ e_2 \mid \text{fun } \ x \rightarrow e \\
\end{align*}
\]
Review: evaluation

• Expressions *step to* new expressions

  \[ e \to e_1 \to e_2 \to \ldots \]

• 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: \( v \to / \to \)
  – But they could take zero steps: \( v \to \star v \)

• *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 $e \rightarrow e' \rightarrow \ldots \rightarrow v$
- We'll just record the fact that $e \Downarrow v$
  - new notation means $e$ evaluates (down) to $v$
  - in ASCII: $e \parallel v$
Values

• Values are already done:
  – Evaluation rule: \( v \mid \mid v \)

• Constants are already values
  – 42 is already a value
  – “3110” is already a value
  – () is already a value

• same for \( C \, v \) and \((v_1, \ldots, v_n)\)

• Functions are already values
  – heads-up: we'll reconsider this choice next lecture
  – fun \( x \to e \) is already a value, no matter what \( e \) is
FUn X -> E
YOU SHALL NOT EVALUATE E
Operator evaluation rule

\[ e_1 + e_2 \ || \ v \]
\[ \text{if } e_1 \ || \ v_1 \]
\[ \text{and } e_2 \ || \ v_2 \]
\[ \text{and } v \text{ is the result of primitive operation } v_1 + v_2 \]

e.g.,
\[ 1 + 2 \ || \ 3 \]
\[ 3.110 \ast 1.0 \ || \ 3.11 \]
\[ 0 < 1 \ || \ \text{true} \]
\[ "zar" \ ^ \ "doz" \ || \ "zardoz" \]
Tuples

To evaluate \((e_1, \ldots, e_n)\),

Evaluate the subexpressions:
\[
e_1 \; || \; v_1
\]

\[
\ldots
\]

\[
e_n \; || \; v_n
\]

Return \((v_1, \ldots, v_n)\)

In which case,
\[
(e_1, \ldots, e_n) \; || \; (v_1, \ldots, v_n)
\]
Tuple evaluation rule

\[(e_1, \ldots, e_n) \mid\mid (v_1, \ldots, v_n)\]
if \(e_1 \mid\mid v_1\)
and \(\ldots\)
and \(e_n \mid\mid v_n\)

e.g.,
so \((1+1, 2+2) \mid\mid (2, 4)\)
because \(1+1 \mid\mid 2\) and \(2+2 \mid\mid 4\)
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 $e_i$ 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: $e_2$ then $e_1$.

```ocaml
((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: \( C \ e_1 \ldots \ e_n \). Rule easily extends.
- Constructors that carry no data behave like constants
  - \texttt{true} is already a value
  - \texttt{[]} is already a value
Progress

e ::=  v  |  C e  |  (e_1, ..., e_n)  |  e_1 + e_2
     |  x  |  e_1 e_2
     |  let x = e_1 in e_2
     |  match e_0 with pi -> e_i
Variables

• What does a variable name evaluate to?
  \[ x \mid \_\_ \_ \_ \_ ?? \]

• 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?

- \( \text{let } x = 2 \text{ in } x+1 \)
- \((\text{fun } x \rightarrow x+1) \ 2\)
- \(\text{match } 2 \text{ with } x \rightarrow x+1\)

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?
  \[ x \mid \text{??}\]

• Trick question: we don’t have enough information to answer it

• Need to know what value variable was \textit{bound} to
  – e.g., \texttt{let x = 2 in x+1}
  – e.g., \texttt{(fun x -> x+1) 2}
  – e.g., \texttt{match 2 with x -> x+1}
  – All evaluate to \texttt{3}, but we reach a point where we need to know binding of \texttt{x}

• Until now, \texttt{we've never needed this}, because we always substituted before we ever get to a variable name
Variables

• OCaml doesn't actually do substitution
  \((\text{fun } x \rightarrow 42) \ 0\)
  – 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 $:
    \[
    \$ \text{let } x = 42 \text{ in} \\
    \quad \text{let } y = "3110" \text{ in} \\
    \quad e
    \]
  – One binding \{x=42\} at $:
    \[
    \text{let } x = 42 \text{ in} \\
    \quad \$ \text{let } y = "3110" \text{ in} \\
    \quad e
    \]
  – Two bindings \{x=42,y="3110"\} at $:
    \[
    \text{let } x = 42 \text{ in} \\
    \quad \text{let } y = "3110" \text{ in} \\
    \quad \$ e
    \]
Variable evaluation

To evaluate \( x \) in environment \( \text{env} \)

Look up value \( v \) of \( x \) in \( \text{env} \)

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

\[ \text{env} :: x \mid \mid v \]
\[ \text{if } v = \text{env}(x) \]

New notation:

- \( \text{env} :: e \mid \mid v \)
  - meaning: in dynamic environment \( \text{env} \), expression \( e \) evaluates down to value \( v \)
- \( \text{env}(x) \)
  - meaning: the value to which \( \text{env} \) binds \( x \)
Redo: rules with environment

Values:

\[
\text{env} :: v || v
\]

Operators:

\[
\text{env} :: e1 + e2 || v
\]

if \(\text{env} :: e1 || v1\)

and \(\text{env} :: e2 || v2\)

and \(v\) is the result of primitive operation \(v1 + v2\)

tuples:

\[
\text{env} :: (e1,...en) || (v1,...vn)
\]

if \(\text{env} :: e1 || v1\)

and ...

and \(\text{env} :: en || vn\)

constructors:

\[
\text{env} :: C e || C v
\]

if \(\text{env} :: e || v\)

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
  - \(((\text{let } x = 1 \text{ in } x+1), (\text{let } y=2 \text{ in } y+2))\)
  - *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

\[ e ::= \ v \mid C\ e \mid (e_1, \ldots, e_n) \mid e_1 + e_2 \]
\[ \mid x \mid e_1 \ e_2 \]
\[ \mid \text{let } x = e_1 \text{ in } e_2 \]
\[ \mid \text{match } e_0 \text{ with } \pi \rightarrow e_i \]
Let expressions

To evaluate \texttt{let } x = e1 \texttt{ in } e2 \texttt{ in environment env}

Evaluate the binding expression \texttt{e1} to a value \texttt{v1} in environment \texttt{env}

\[
\texttt{env} :: e1 \ |\ | \ v1
\]

Extend the environment to bind \texttt{x} to \texttt{v1}

\[
\texttt{env}' = \texttt{env} + \{ x = v1 \}
\]

Evaluate the body expression \texttt{e2} to a value \texttt{v2} in environment \texttt{env}'

\[
\texttt{env'} :: e2 \ |\ | \ v2
\]

Return \texttt{v2}
Let expression evaluation rule

\[
\text{env} :: \text{let } x=e_1 \text{ in } e_2 \mid \mid v_2 \\
\text{if } \text{env} :: e_1 \mid \mid v_1 \\
\text{and } \text{env}+\{x=v_1\} :: e_2 \mid \mid v_2
\]

Example:
\[
\{\} :: \text{let } x = 42 \text{ in } x \mid \mid 42
\]
Why? Because...

• \[
\{\} :: 42 \mid \mid 42
\]
• and \[
\text{and } \{\}+\{x=42\} :: x \mid \mid 42
\]
  – Why? because if \text{env} is \{x=42\} then \text{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:
  ```ocaml
  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):
  ```ocaml
  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
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 \mid \text{C e} \mid (e_1, \ldots, e_n) \mid e_1 + e_2
| \ x \mid e_1 \ e_2
| \text{let x = e_1 in e_2}
| \text{match e_0 with pi -> ei}