CS 3110

Lecture 7: The dynamic environment

Prof. Clarkson

Fall 2014

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

Review

Features so far: variables, operators, let expressions, if expressions, functions (higher-order, anonymous), datatypes, records, lists, options, match expressions, type variables

Today:

- Improved evaluation rules

Question #1

How much of PS2 have you finished?

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

PS1 handback

- Numeric scores on CMS this afternoon
- Written comments on hardcopies in the homework handback room around the same time
- Go over questions & talk about solutions in recitation on Wednesday
  - Also go over a Git tutorial
- Regrades? Sure! Submit request by CMS within one week
  - Always good to talk to your TA in advance; can save time and trouble
  - We reserve the right to regrade entire solution; grade could go up or down
  - Want to improve your final grade in course? Spend your time on making PS2 great, rather than getting one more point on PS1

Semantics

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

Dynamic semantics

Today: careful account of dynamic semantics of the essential, core features of OCaml

- many rules we've seen already
- some new twists along the way

Change our *model of evaluation*:

- **Substitution model:** substitute value for variable in body of let expression & in body of function
  - What we've done doing so far
  - Very tricky to define substitution correctly
  - Good mental model, not really what OCaml does
- **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 ::= c | (op) | x | (e1, …, en)
      | C e
      | e1 e2
      | fun x -> e
      | let x = e1 in e2
      | match e0 with pi -> ei
```

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`

Evaluation

- Expressions evaluate to values

```
e --> v
```

- Long arrow means "evaluates to"
- Recall: evaluation is meaningless if expression does not type check
- Values "have no further computation to do"
  - So they trivially evaluate to themselves: `v-->v`

Values

Values are a *syntactic subset* of expressions:

```
v ::= c | (op) | (v1, …, vn)
    | C v
    | fun x -> e
```

Not values: function application, let expression, match expression

Tuples

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

Evaluate the subexpressions:

Evaluate **`en --> vn`**

and ... **`e1 --> v1`**

Return `(v1,...vn)`

In which case,

`(e1,...en) --> (v1,...vn)`

Tuple evaluation rule

If `en --> vn`
and ...
and `e2 --> v2`
and `e1 --> v1`
then `(e1,...en) --> (v1,...vn)`

e.g.,
`(+) 1 1 --> 2` (trust me)
and `(+) 2 2 --> 4` (trust me)
so `((+) 1 1, (+) 2 2) --> (2,4)`

Question #2

If we changed evaluation order to be e1 first, then e2, ... up to en, 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 e1 first, then e2, ... up to en, 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.** Without imperative features, no program can ever distinguish the order of evaluation.

A: Right to left: **`en`** then … then **`e1`**.

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

(exceptions are actually side effects…but we let you use them anyway on the problem sets)

Constructors

To evaluate `C e`,

Evaluate the subexpression:

```
e --> v
```

Return `C v`

In which case, `C e --> C v`

Constructor evaluation rule

If `e --> v`

then `C e --> C v`

e.g.,

```
(+) 1 1 --> 2
```

so `Some ((+) 1 1) --> Some 2`

Constants

- Constants are already values
  - `42` is already a value
  - `"3110"` is already a value
  - `()` is already a value
- So `c-->c`
  - (evaluation rule here is trivial)
- Constructors that carry no data behave like constants
  - `true` is already a value
  - `Monday` is already a value

Operators and functions

- Functions are values
  - Operators **`(op)`** are built-in functions
  - Anonymous functions **`fun x-> e`** are user-defined functions
- So both are already values
  - **`fun x -> x+1 --> fun x -> x+1`**
  - **`(+) --> (+)`**
  - **`(~-) --> (~-)`**
- In general,
  - **`(op) --> (op)`**
  - **`(fun x -> e) --> (fun x -> e)`**
- Evaluation rule again trivial, like for constants

Progress

```
e ::= c | (op) | x | (e1, …, en)
    | C e
    | e1 e2
    | fun x -> e
    | 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**
- **Solution: dynamic environment**

Dynamic environment

- Set of bindings of all current variables
  - e.g.,{ x=42, y="3110"} would be bindings at ^^ in
    ```
    let x=42 in let y = "3110" in ^^ e
    ```
- 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
    ```

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

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

Variable evaluation rule

```
env :: x --> v
   where v = env(x)
```

so we could instead more simply write

```
env :: x --> env(x)
```

Redo: rules with environment

Constants, operators, functions:

```
env :: c --> c
env :: (op) --> (op)
env :: (fun x -> e) --> (fun x -> e)
```

Constructors:

If `env :: e --> v`
then `env :: C e --> C v`

Tuples:

If `env :: en --> vn`
and ...
and `env :: e1 --> v1`
then `env :: (e1,...en) --> (v1,...vn)`

Why the same environment?

Scope

- Bindings are in effect only in the *scope* (the "block") in which they occur

```
let x=42 in
  ^^ x + (let y="3110" in
          int_of_string y)
```

  - `y` is not in scope at ^^
- 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 ::= c | (op) | x | (e1, …, en)
    | C e
    | e1 e2
    | fun x -> e
    | 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

If `env :: e1 --> v1`
and if `env+{x=v1} :: e2 --> v2`
then `env :: let x=e1 in e2 --> v2`

Example:

```
let x = 42 in x --> 42
```

Why?

1. Evaluate binding expression **`42`** to value **`42`**
   - By constant rule, **`{} :: 42 --> 42`**
2. Extend environment to bind **`x`** to **`42`**
3. Evaluate body expression **`x`** to value 42 in extended environment
   - By variable rule, **`{x=42} :: x --> 42`**
     (why? if **`env={x=42}`** then **`env(x) = 42`**)
4. Return value of body expression, **`42`**

Let expression longer example

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

1. Evaluate binding expression 42 to value 42
2. Extend environment to bind x to 42
   - env is now {x=42}
3. Evaluate body expression let y = "3110" in x to value 42
   1. Evaluate binding expression "3110" to value "3110"
   2. Extend environment to bind y to "3110"
      - env is now {x=42,y="3110"}
   3. Evaluate body expression x to value 42
      1. Look up value of x in environment, return 42

Let expression example

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

Another way to express previous slide:

1. By variable rule, `{x=42,y="3110"} :: x --> 42`
2. By constant rule, `{x=42} :: "3110" --> "3110"`
3. By `let` rule with (1) and (2), `{x=42} :: let y = "3110" in x --> 42`
4. By constant rule, `{} :: 42 --> 42`
5. By `let` rule with (3) and (4), `{} :: let x = 42 in let y = "3110" in 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))
```

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
```

(Proof sketch)

1. By constant rule, `{x=0} :: 1 --> 1`
2. By variable rule, `{x=1} :: x --> 1`
3. By `let` rule with 1 and 2, `{x=0} :: let x = 1 in x --> 1`
4. By variable rule, `{x=0} :: x --> 0`
5. By intuition (haven't done function application yet) with 3 and 4, `{x=0} :: x + (let x =1 in x) --> 1`
6. By constant rule, `{} :: 0 --> 0`
7. By `let` rule with 5 and 6, `{} :: let x = 0 in x + (let x = 1 in x) --> 1`

Progress

```
e ::= c | (op) | x | (e1, …, en)
    | C e
    | e1 e2
    | fun x -> e
    | let x = e1 in e2
    | match e0 with pi -> ei
```

Match expressions

To evaluate `match e0 with p1 -> e1 | ... | pn -> en` in environment **`env`**

Evaluate expression **`e0`** to value **`v0`** in **`env`**

Find the first pattern **`pi`** that matches **`v0`**

That match produces new bindings **`b`**

Evaluate expression **`ei`** to value **`vi`** in environment **`env+b`**

Return `vi`

Match expression rule

If `env :: e0 --> v0`

and `pi` is the first pattern to match `v0`

and that match produces bindings `b`

and `env+b :: ei --> vi`

then `env :: match e with p1 -> e1`
`| ... | pn -> en --> vi`

Example of match

```
{} :: match 42 with x -> x --> 42
```

1. Evaluate expression **`42`** to value **`42`**
2. Match **`42`** against patterns; pattern x is the first that matches; it produces binding **`{x=42}`**
3. Evaluate expression **`x`** to value **`42`** in environment **`{}+{x=42}`**
4. Return **`42`**

Example of match

```
{} :: match 42 with x -> x --> 42
```

Another way to express previous slide:

1. By constant rule, `{} :: 42 --> 42`
2. By pattern matching rules, `x` matches `42` and produces binding `x=42`
3. By variable rule, `{x=42} :: x --> 42`
4. By `match` rule with 2 and 3, `{} :: match 42 with x -> x --> 42`

Progress

```
e ::= c | (op) | x | (e1, …, en)
    | C e
    | e1 e2
    | fun x -> e
    | let x = e1 in e2
    | match e0 with pi -> ei
```

Please hold still for 1 more minute

WRAP-UP FOR TODAY

Upcoming events

- **PS2 is due Thursday at 11:59 pm**
- Clarkson permanent(?) office hours:
  Tuesday & Thursday 3-4 pm

This is dynamic.

THIS IS 3110