A Functional Evaluation Model

COS 326

David Walker
Princeton University
In order to be able to write a program, you have to have a solid grasp of how a programming language works.

We often call the definition of “how a programming language works” its *semantics*.

There are many kinds of programming language semantics.

In this lecture, we will look at O’Caml’s *call-by-value* evaluation:

– First, informally, giving *program rewrite rules by example*
– Second, using code, by specifying an *OCaml interpreter* in OCaml
– Third, more formally, using logical *inference rules*

In each case, we are specifying what is known as OCaml's *operational semantics*
O’CAML BASICS:
CORE EXPRESSION EVALUATION
Evaluation

• Execution of an OCaml expression
  – produces a value
  – and may have some effect (eg: it may raise an exception, print a string, read a file, or store a value in an array)

• A lot of OCaml expressions have no effect
  – they are pure
  – they produce a value and do nothing more
  – the pure expressions are the easiest kinds of expressions to reason about

• We will focus on evaluation of pure expressions
Evaluation of Pure Expressions

• Given an expression `e`, we write:
  
  \[
  e \rightarrow v
  \]

  to state that expression `e` evaluates to value `v`

• Note that "`e \rightarrow v`" is not itself a program -- it is some notation that we use talk about how programs work
Evaluation of Pure Expressions

• Given an expression $e$, we write:

\[ e \rightarrow v \]

...to state that expression $e$ evaluates to value $v$

• Some examples:
Evaluation of Pure Expressions

- Given an expression $e$, we write:

\[ e \rightarrow v \]

...to state that expression $e$ evaluates to value $v$

- Some examples:

\[ 1 + 2 \]
Evaluation of Pure Expressions

• Given an expression e, we write:

  \[ e \rightarrow v \]

  to state that expression e evaluates to value v

• Some examples:

  \[ 1 + 2 \rightarrow 3 \]
Evaluation of Pure Expressions

• Given an expression e, we write:

\[ e \rightarrow v \]

to state that expression e evaluates to value v

• Some examples:

1 + 2 \rightarrow 3

2
Evaluation of Pure Expressions

• Given an expression $e$, we write:
  
  \[ e \rightarrow v \]

  to state that expression $e$ evaluates to value $v$

• Some examples:
  
  \[ 1 + 2 \rightarrow 3 \]

  \[ 2 \rightarrow 2 \]
Evaluation of Pure Expressions

• Given an expression $e$, we write:
  
  $e \rightarrow v$

  to state that expression $e$ evaluates to value $v$

• Some examples:

  1 + 2 $\rightarrow$ 3

  2 $\rightarrow$ 2

  int_to_string 5 $\rightarrow$ "5"
More generally, we say expression $e$ (partly) evaluates to expression $e'$:

\[ e \rightarrow e' \]
Evaluation of Pure Expressions

More generally, we say expression e (partly) evaluates to expression e’:

\[ e \rightarrow e' \]

Evaluation is *complete* when e’ is a value

- In general, I’ll use the letter “v” to represent an arbitrary value
- The letter “e” represents an arbitrary expression
- Concrete numbers, strings, characters, etc. are all values, as are:
  - tuples, where the fields are values
  - records, where the fields are values
  - datatype constructors applied to a value
  - *functions*
Evaluation of Pure Expressions

- Some expressions (all the interesting ones!) take many steps to evaluate them:

$$(2 \times 3) + (7 \times 5)$$
Evaluation of Pure Expressions

• Some expressions (all the interesting ones!) take many steps to evaluate them:

$$(2 \times 3) + (7 \times 5) \rightarrow 6 + (7 \times 5)$$
Evaluation of Pure Expressions

• Some expressions (all the interesting ones!) take many steps to evaluate them:

\[(2 \times 3) + (7 \times 5)\]
\[\longrightarrow 6 + (7 \times 5)\]
\[\longrightarrow 6 + 35\]
Some expressions (all the interesting ones!) take many steps to evaluate them:

\[(2 \times 3) + (7 \times 5)\]
\[\rightarrow 6 + (7 \times 5)\]
\[\rightarrow 6 + 35\]
\[\rightarrow 41\]
Some expressions do not compute a value and it is not obvious how to proceed:

```
"hello" + 1 --> ????
```

- A **strongly typed language rules out a lot of nonsensical expressions that compute no value**, like the one above
- Other expressions compute no value but raise an exception:

```
7 / 0 --> raise Divide_by_zero
```

- Still others simply fail to terminate ...
Let Expressions: Evaluate using Substitution

\[
\begin{align*}
\text{let } x &= 30 \text{ in} \\
\text{let } y &= 12 \text{ in} \\
x + y
\end{align*}
\]

\[
\begin{align*}
\text{let } y &= 12 \text{ in} \\
30 + y
\end{align*}
\]

\[
\begin{align*}
30 + 12
\end{align*}
\]

\[
42
\]
Informal Evaluation Model

To evaluate a function call “f a”

• first evaluate f until we get a function value (fun x -> e)
• then evaluate a until we get an argument value v
• then substitute v for x in e, the function body
• then evaluate the resulting expression.

this is why we say O’Caml is “call by value”

(let f = (fun x -> x + 1) in f) (30+11)  -->

(fun x -> x + 1) (30 + 11)  -->

(fun x -> x + 1) 41  -->

41 + 1  -->  42
Another example:

```plaintext
let add x y = x+y in
let inc = add 1 in
let dec = add (-1) in
dec(inc 42)
```
Recall the syntactic sugar:

```ocaml
let add = fun x -> (fun y -> x+y) in
let inc = add 1 in
let dec = add (-1) in
dec(inc 42)
```
Then we use the let rule – we substitute the *value* for `add`:

```ocaml
let add = fun x -> (fun y -> x+y) in
let inc = add 1 in
let dec = add (-1) in
dec(inc 42)
```

--> 

```ocaml
let inc = (fun x -> (fun y -> x+y)) 1 in
let dec = (fun x -> (fun y -> x+y)) -1 in
dec(inc 42)
```
\[\begin{align*}
\text{let } \texttt{inc} &= \ (\text{\texttt{fun}} \ \texttt{x} \ \rightarrow \ (\text{\texttt{fun}} \ \texttt{y} \ \rightarrow \ \texttt{x}+\texttt{y})) \ 1 \ \text{in} \\
\text{let } \texttt{dec} &= \ (\text{\texttt{fun}} \ \texttt{x} \ \rightarrow \ (\text{\texttt{fun}} \ \texttt{y} \ \rightarrow \ \texttt{x}+\texttt{y})) \ (-1) \ \text{in} \\
\texttt{dec}(\texttt{inc} \ 42) \\
\rightarrow \ \text{not a value; must reduce before substituting for inc} \\
\text{let } \texttt{inc} &= \ \text{\texttt{fun}} \ \texttt{y} \ \rightarrow \ \texttt{1+y} \ \text{in} \\
\text{let } \texttt{dec} &= \ (\text{\texttt{fun}} \ \texttt{x} \ \rightarrow \ (\text{\texttt{fun}} \ \texttt{y} \ \rightarrow \ \texttt{x}+\texttt{y})) \ (-1) \ \text{in} \\
\texttt{dec}(\texttt{inc} \ 42)\end{align*}\]
let inc = \( \text{fun } y \to 1+y \) in
let dec = \( \text{fun } x \to (\text{fun } y \to x+y) \) (-1) in
dec(inc 42)

\[ \text{now a value} \]
Next: simplify `dec`’s definition using the function-call rule.

\[
\text{let } \text{dec} = \text{(fun } x \rightarrow (\text{fun } y \rightarrow x+y)) (-1) \text{ in }
\]
\[
\text{dec}((\text{fun } y \rightarrow 1+y) 42)
\]

\[
\rightarrow
\]
\[
\text{let } \text{dec} = \text{fun } y \rightarrow -1+y \text{ in }
\]
\[
\text{dec}((\text{fun } y \rightarrow 1+y) 42)
\]
And we can use the let-rule now to substitute \( \texttt{dec} \):

\[
\text{let } \texttt{dec} = \texttt{fun } y \rightarrow -1+y \texttt{ in } \\
\texttt{dec}(\texttt{fun } y \rightarrow 1+y) \texttt{ 42) -->}
\]

\[
\texttt{(fun } y \rightarrow -1+y)(\texttt{(fun } y \rightarrow 1+y) \texttt{ 42)}
\]
Informal Evaluation Model

Now we can’t yet apply the first function because the argument is not yet a value – it’s a function call. So we need to use the function-call rule to simplify it to a value:

\[(\text{fun } y \rightarrow -1+y)\ (\ (\text{fun } y \rightarrow 1+y)\ 42)\ -->\]

\[(\text{fun } y \rightarrow -1+y)\ (1+42)\ -->\]

\[(\text{fun } y \rightarrow -1+y)\ 43\ -->\]

\[-1+43\ -->\]

42
Consider the following OCaml code:

```ocaml
let x = 30 in
let y = 12 in
x+y;;
```

Does this evaluate any differently than the following?

```ocaml
let a = 30 in
let b = 12 in
a+b;;
```
A basic principle of programs is that systematically changing the names of variables shouldn’t cause the program to behave any differently – it should evaluate to the same thing.

```
let x = 30 in
let y = 12 in
x+y;;
```

But we do have to be careful about systematic change.

```
let a = 30 in
let a = 12 in
a+a;;
```

Systematic change of variable names is called *alpha-conversion*. 
Substitution

Wait a minute, how do we evaluate this using the let-rule? If we substitute 30 for “a” naively, then we get:

\[
\begin{align*}
&\text{let } a = 30 \text{ in} \\
&\text{let } a = 12 \text{ in} \\
& a+a \\
\rightarrow \\
&\text{let } 30 = 12 \text{ in} \\
&30+30
\end{align*}
\]

Which makes no sense at all!
Besides, Ocaml returns 24 not 60.
What went wrong with our informal model?
Scope and Modularity

• Lexically scoped (a.k.a. statically scoped) variables have a simple rule: the nearest enclosing “let” in the code defines the variable.

• So when we write:

```
let a = 30 in
let a = 12 in
a+a;;
```

• we know that the “a+a” corresponds to “12+12” as opposed to “30+30” or even weirder “30+12”.

A Revised Let-Rule:

- To evaluate “let \( x = e_1 \) in \( e_2 \)”:  
  - First, evaluate \( e_1 \) to a value \( v \).  
  - Then substitute \( v \) for the *corresponding uses* of \( x \) in \( e_2 \).  
  - Then evaluate the resulting expression.

\[
\begin{align*}
\text{let } a &= 30 \text{ in } \\
\text{let } a &= 12 \text{ in } \\
(a+a) &
\end{align*}
\]

\[
\begin{align*}
\text{let } a &= 12 \text{ in } \\
(a+a) &
\end{align*}
\]

This “\( a \)” doesn’t correspond to the uses of “\( a \)” below.

So when we substitute 30 for it, it doesn’t change anything.

\[
\begin{align*}
12+12 &
\end{align*}
\]

\[
\begin{align*}
24 &
\end{align*}
\]
• But what does “corresponding uses” mean?

• Consider:

```plaintext
let a = 30 in
let a = (let a = 3 in a*4) in
a+a;;
```
• We can view a program as a tree – the parentheses and precedence rules of the language help determine the structure of the tree.

```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;
```

==

```
(let a = (30) in
 (let a =
   (let a = (3) in (a*4))
in
  (a+a)))
```
An occurrence of a variable where we are defining it via let is said to be a *binding occurrence* of the variable.

let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;
A non-binding occurrence of a variable is said to be a free variable.

That is a use of a variable as opposed to a definition.

```
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;
```
Given a free variable occurrence, we can find where it is bound by ...

let a = 30 in
let a =
  (let a = 3 in a*4) in
a+a;;
• crawling up the tree to the nearest enclosing let...

let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;
• crawling up the tree to the nearest enclosing let...

let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;
• crawling up the tree to the nearest enclosing let...

let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;
• and see if the “let” binds the variable – if so, we’ve found the nearest enclosing definition. If not, we keep going up.

```
let a = 30 in
let a =
    (let a = 3 in a*4)
in
a+a;;
```
Abstract Syntax Trees

- Now we can also systematically rename the variables so that it’s not so confusing. Systematic renaming is called \textit{alpha-conversion}

```ocaml
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;
```
• Start with a let, and pick a fresh variable name, say “x”

```ocaml
let a = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;
```
- Rename the binding occurrence from “a” to “x”.

```ocaml
let x = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;
```
• Then rename all of the free occurrences of the variables that this let binds.

```ocaml
let x = 30 in
let a = (let a = 3 in a*4) in
a+a;;
```
let x = 30 in
let a =
  (let a = 3 in a*4)
in
a+a;;

• There are none in this case!
• There are none in this case!

```
let x = 30 in
let a = (let a = 3 in a*4) in
a+a;;
```
• Let’s do another let, renaming “a” to “y”.

```ocaml
define x = 30 in
  let a =
    (let a = 3 in a*4)
in
  a+a;

let x = 30 in
let a =
  (let a = 3 in a*4)
in
  a+a;;
```
• Let’s do another let, renaming “a” to “y”.

```ocaml
def x = 30 in
let y =
    (let a = 3 in a*4)
in
y+y;;
```
• And if we rename the other let to "z":

```plaintext
let x = 30 in
let y =
  (let z = 3 in z*4)
in
y+y;;
```
• And if we rename the other let to “z”:

```ocaml
let x = 30 in
let y =
  (let z = 3 in z*4)
in
y+y;;
```
AN O’CAML DEFINITION OF O’CAML EVALUATION
Implementing an Interpreter

text file containing program as a sequence of characters

```
let x = 3 in
x + x
```

Parsing

data structure representing program

```haskell
Let ("x",
  Num 3,
  Binop(Plus, Var "x", Var "x"))
```

data structure representing result of evaluation

```
Num 6
```

Evaluation

```
6
```

Pretty Printing

the data type and evaluator tell us a lot about program semantics

text file/stdout containing with formatted output
We can define a datatype for simple OCaml expressions:

```
type variable = string ;;
type op = Plus | Minus | Times | ... ;;
type exp =
  | Int_e of int
  | Op_e of exp * op * exp
  | Var_e of variable
  | Let_e of variable * exp * exp ;;
```
We can define a datatype for simple OCaml expressions:

```ocaml
type variable = string ;;
type op = Plus | Minus | Times | ... ;;
type exp =
    | Int_e of int
    | Op_e of exp * op * exp
    | Var_e of variable
    | Let_e of variable * exp * exp ;;

let three = Int_e 3 ;;
let three_plus_one =
    Op_e (Int_e 1, Plus, Int_e 3) ;;
```
We can represent the OCaml program:

```
let x = 30 in
let y =
  (let z = 3 in
   z*4)
in
y+y;;
```

as an exp value:

```
Let_e("x", Int_e 30,
   Let_e("y",
      Let_e("z", Int_e 3,
         Op_e(Var_e "z", Times, Int_e 4)),
         Op_e(Var_e "y", Plus, Var_e "y"))
```
Making These Ideas Precise

Notice how this reflects the “tree”:

```
Let_e("x", Int_e 30,
    Let_e("y", Let_e("z", Int_e 3,
        Op_e(Var_e "z", Times, Int_e 4)),
        Op_e(Var_e "y", Plus, Var_e "y")
)
```

```
let
  x
  30
let
  y
  let
    z
    3
    *
      y
      y
  +
    y
    y
```
Free versus Bound Variables

```ocaml
type exp =
  | Int_e of int
  | Op_e of exp * op * exp
  | Var_e of variable
  | Let_e of variable * exp * exp
```

This is a free occurrence of a variable.
Free versus Bound Variables

```
type exp =
  | Int_e of int
  | Op_e of exp * op * exp
  | Var_e of variable
  | Let_e of variable * exp * exp
```

This is a **free** occurrence of a variable

This is a **binding** occurrence of a variable
END