Implementing OCaml in OCaml

COS 326
David Walker
Princeton University
Implementing an Interpreter

text file containing program as a sequence of characters

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

Parsing

data structure representing program

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

data structure representing result of evaluation

```
Num 6
```

Evaluation

text file/stdout containing with formatted output

```
6
```

Pretty Printing

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

type value = 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

type value = exp

let e1 = Int_e 3
let e2 = Int_e 17
let e3 = Op_e (e1, Plus, e2)
```

represents “3 + 17”
let is_value (e:exp) : bool =
    match e with
    | Int_e _ -> true
    | ( Op_e _
    |  Let_e _
    |  Var_e _ ) -> false

eval_op : value -> op -> value -> exp

(* substitute v x e: replace free occurrences of x with v in e *)
substitute : value -> variable -> exp -> exp
is_value : exp -> bool
eval_op : value -> op -> value -> value
substitute : value -> variable -> exp -> exp

let rec eval (e:exp) : exp =
    match e with
    | Int_e i ->
    | Op_e(e1,op,e2) ->
    | Let_e(x,e1,e2) ->
A Simple Evaluator

```ml
let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) ->
  | Let_e(x,e1,e2) ->
```

- `is_value` : `exp` -> `bool`
- `eval_op` : `value` -> `op` -> `value` -> `value`
- `substitute` : `value` -> `variable` -> `exp` -> `exp`
A Simple Evaluator

is_value : exp -> bool
eval_op : value -> op -> value -> value
substitute : value -> variable -> exp -> exp

let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) ->
    let v1 = eval e1 in
    let v2 = eval e2 in
    eval_op v1 op v2
  | Let_e(x,e1,e2) ->
A Simple Evaluator

is_value : exp -> bool
eval_op : value -> op -> value -> value
substitute : value -> variable -> exp -> exp

let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) ->
    let v1 = eval e1 in
    let v2 = eval e2 in
    eval_op v1 op v2
  | Let_e(x,e1,e2) ->
    let v1  = eval e1 in
    let e2' = substitute v1 x e2 in
    eval e2'
is_value : exp -> bool

eval_op : value -> op -> value -> value

substitute : value -> variable -> exp -> exp

let rec eval (e:exp) : exp =

   match e with
   | Int_e i -> Int_e i
   | Op_e(e1,op,e2) ->
     eval_op (eval e1) op (eval e2)
   | Let_e(x,e1,e2) ->
     eval (substitute (eval e1) x e2)

Why?
is_value : exp -> bool
eval_op : value -> op -> value -> value
substitute : value -> variable -> exp -> exp

let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) ->
    eval_op (eval e1) op (eval e2)
  | Let_e(x,e1,e2) ->
    eval (substitute (eval e1) x e2)

Which gets evaluated first?
Does OCaml use left-to-right eval order or right-to-left?
Always use OCaml let if you want to specify evaluation order.
Simpler but Dangerous

\[
\begin{align*}
\text{is\_value} & : \text{exp} \rightarrow \text{bool} \\
\text{eval\_op} & : \text{value} \rightarrow \text{op} \rightarrow \text{value} \rightarrow \text{value} \\
\text{substitute} & : \text{value} \rightarrow \text{variable} \rightarrow \text{exp} \rightarrow \text{exp}
\end{align*}
\]

\[
\text{let rec eval (e:exp) : exp =}
\]

\[
\text{match e with}
\]

\[
\begin{align*}
| \text{Int\_e} i & \rightarrow \text{Int\_e} i \\
| \text{Op\_e}(e1,op,e2) & \rightarrow \\
& \quad \text{eval\_op (eval e1) op (eval e2)} \\
| \text{Let\_e}(x,e1,e2) & \rightarrow \\
& \quad \text{eval (substitute (eval e1) x e2)}
\end{align*}
\]

Since the language we are interpreting is *pure* (no effects), it won’t matter which expression gets evaluated first. We’ll produce the same answer in either case.
Oops! We Missed a Case:

```ocaml
let eval_op v1 op v2 = ...
let substitute v x e = ...

let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
  | Var_e x -> ???
```

We should never encounter a variable – they should have been substituted with a value! (This is a type-error.)

We could leave out the case for variables if we type check before evaluating but that will create a mess of OCaml warnings – bad style. (Bad for debugging.)
We Could Use Options:

```ocaml
let eval_op v1 op v2 = ...
let substitute v x e = ...

let rec eval (e:exp) : exp option =
    match e with
    | Int_e i -> Some(Int_e i)
    | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
    | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
    | Var_e x -> None
```

But this isn’t quite right – we need to match on the recursive calls to eval to make sure we get Some value!
exception UnboundVariable of variable

let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
  | Var_e x -> raise (UnboundVariable x)

Instead, we can throw an exception.
Note that an exception declaration is a lot like a datatype declaration. Really, we are extending one big datatype (exn) with a new constructor (UnboundVariable).

Later on, we’ll see how to catch an exception.
AUXILIARY FUNCTIONS
Evaluating the Primitive Operations

let eval_op (v1:exp) (op:operand) (v2:exp) : exp =
  match v1, op, v2 with
  | Int_e i, Plus, Int_e j -> Int_e (i+j)
  | Int_e i, Minus, Int_e j -> Int_e (i-j)
  | Int_e i, Times, Int_e j -> Int_e (i*j)
  | _, (Plus | Minus | Times), _ ->
    if is_value v1 && is_value v2 then raise TypeError
    else raise NotValue

let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
  | Var_e x -> raise (UnboundVariable x) ;;
let substitute (v:value) (x:variable) (e:exp) : exp =

  ...

;;

Want to replace x (and only x) with v.
let substitute (v:value) (x:variable) (e:exp) : exp =
let rec subst (e:exp) : exp =
    match e with
    | Int_e _ ->
    | Op_e(e1,op,e2) ->
    | Var_e y -> ... use x ...
    | Let_e (y,e1,e2) -> ... use x ...

in
subst e
;;
let substitute (v:value) (x:variable) (e:exp) : exp =
  let rec subst (e:exp) : exp =
    match e with
    | Int_e _ -> e
    | Op_e(e1,op,e2) ->
    | Var_e y ->
    | Let_e (y,e1,e2) ->
    in
    subst e
  ;;
let substitute (v:value) (x:variable) (e:exp) : exp =
    let rec subst (e:exp) : exp =
        match e with
        | Int_e _ -> e
        | Op_e(e1,op,e2) -> Op_e(subst e1,op,subst e2)
        | Var_e y ->
        | Let_e (y,e1,e2) ->

    in
    subst e
    ;;
let substitute (v:exp) (x:variable) (e:exp) : exp =

let rec subst (e:exp) : exp =

  match e with
  | Int_e _ -> e
  | Op_e(e1,op,e2) -> Op_e(subst e1,op,subst e2)
  | Var_e y -> if x = y then v else e
  | Let_e (y,e1,e2) ->

in
subst e
;;
let substitute (v:exp) (x:variable) (e:exp) : exp =
  let rec subst (e:exp) : exp =
    match e with
    | Int_e _ -> e
    | Op_e(e1,op,e2) -> Op_e(subst e1,op,subst e2)
    | Var_e y -> if x = y then v else e
    | Let_e (y,e1,e2) ->
      Let_e (y,
        subst e1,
        subst e2)
  in
  subst e
;;
let substitute (v:exp) (x:variable) (e:exp) : exp =
  let rec subst (e:exp) : exp =
    match e with
    | Int_e _ -> e
    | Op_e(e1,op,e2) -> Op_e(subst e1,op,subst e2)
    | Var_e y -> if x = y then v else e
    | Let_e (y,e1,e2) ->
      Let_e (y, if x = y then e1 else subst e1,
             if x = y then e2 else subst e2)
  in
  subst e
;;
let substitute (v:exp) (x:variable) (e:exp) : exp =

let rec subst (e:exp) : exp =

match e with
    | Int_e _ -> e
    | Op_e(e1,op,e2) -> Op_e(subst e1,op,subst e2)
    | Var_e y -> if x = y then v else e
    | Let_e (y,e1,e2) ->
        Let_e (y, subst e1,
            if x = y then e2 else subst e2)
in
subst e

;;

evaluation/substitution must implement our variable scoping rules correctly
**Substitution**

```ml
let substitute (v:exp) (x:variable) (e:exp) : exp =
  let rec subst (e:exp) : exp =
    match e with
    | Int_e _ -> e
    | Op_e(e1,op,e2) -> Op_e(subst e1,op,subst e2)
    | Var_e y -> if x = y then v else e
    | Let_e (y,e1,e2) ->
      Let_e (y,
        subst e1,
        if x = y then e2 else subst e2)
  in
  subst e
;;
```

If x and y are the same variable, then y **shadows** x.
let substitute (v:exp) (x:variable) (e:exp) : exp =
let rec subst (e:exp) : exp =
  match e with
  | Int_e _ -> e
  | Op_e(e1,op,e2) -> Op_e(subst e1,op,subst e2)
  | Var_e y -> if x = y then v else e
  | Let_e (y,e1,e2) ->
    Let_e (y,
      subst e1,
      if x = y then e2 else subst e2)
  in subst e
;;

If x and y are the same variable, then y shadows x.
SCALING UP THE LANGUAGE
(MORE FEATURES, MORE FUN)
Scaling up the Language

```ocaml
type exp = Int_e of int | Op_e of exp * op * exp
     | Var_e of variable | Let_e of variable * exp * exp
     | Fun_e of variable * exp | FunCall_e of exp * exp ;;
```
Scaling up the Language

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

OCaml’s
fun x -> e
is represented as
Fun_e(x,e)
Scaling up the Language

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

A function call

fact 3

is implemented as

FunCall_e (Var_e "fact", Int_e 3)
scaling up the language:

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

let is_value (e:exp) : bool =
  match e with
  | Int_e _ -> true
  | Fun_e (_,_) -> true
  | ( Op_e (_,_,_) )
  | Let_e (_,_,_)
  | Var_e _
  | FunCall_e (_,_) ) -> false ;;
```

functions are values!

easy exam question:
what value does the OCaml interpreter produce when you enter
(fun x -> 3) in to the prompt?
answer: the value produced is (fun x -> 3)
Scaling up the Language:

type exp = Int_e of int | Op_e of exp * op * exp
  | Var_e of variable | Let_e of variable * exp * exp
  | Fun_e of variable * exp | FunCall_e of exp * exp;;

let is_value (e:exp) : bool =
match e with
  | Int_e _ -> true
  | Fun_e (_,_) -> true
  | ( Op_e (_,_,_) |
  | Let_e (_,_,_)
  | Var_e _
  | FunCall_e (_,_) ) -> false ;;
let rec eval (e:exp) : exp =
    match e with
    | Int_e i -> Int_e i
    | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
    | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
    | Var_e x -> raise (UnboundVariable x)
    | Fun_e (x,e) -> Fun_e (x,e)
    | FunCall_e (e1,e2) ->
       (match eval e1, eval e2 with
        | Fun_e (x,e), v2 -> eval (substitute v2 x e)
        | _ -> raise TypeError)
let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
  | Var_e x -> raise (UnboundVariable x)
  | Fun_e (x,e) -> Fun_e (x,e)
  | FunCall_e (e1,e2) ->
    (match eval e1, eval e2 with
     | Fun_e (x,e), v2 -> eval (substitute v2 x e)
     | _ -> raise TypeError)
let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
  | Var_e x -> raise (UnboundVariable x)
  | Fun_e (x,e) -> Fun_e (x,e)
  | FunCall_e (e1,e2) ->
    (match eval e1, eval e2 with
    | Fun_e (x,e), v2 -> eval (substitute v2 x e)
    | _ -> raise TypeError)

To evaluate a function call, we first evaluate both e1 and e2 to values.
let rec eval (e:exp) : exp =
match e with
| Int_e i -> Int_e i
| Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
| Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
| Var_e x -> raise (UnboundVariable x)
| Fun_e (x,e) -> Fun_e (x,e)
| FunCall_e (e1,e2) ->
  (match eval e1, eval e2 with
   | Fun_e (x,e), v2 -> eval (substitute v2 x e)
   | _ -> raise TypeError)

e1 had better evaluate to a function value, else we have a type error.
let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
  | Var_e x -> raise (UnboundVariable x)
  | Fun_e (x,e) -> Fun_e (x,e)
  | FunCall_e (e1,e2) ->
    (match eval e1, eval e2 with
     | Fun_e (x,e), v2 -> eval (substitute v2 x e)
     | _ -> raise TypeError)

Then we substitute e2’s value (v2) for x in e and evaluate the resulting expression.
let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
  | Var_e x -> raise (UnboundVariable x)
  | Fun_e (x,e) -> Fun_e (x,e)
  | FunCall_e (e1,e2) ->
    (match eval e1
      | Fun_e (x,e) -> eval (substitute (eval e2) x e)
      | _ -> raise TypeError)

We don’t really need to pattern-match on e2. Just evaluate here
```ocaml
let rec eval (e : exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
  | Var_e x -> raise (UnboundVariable x)
  | Fun_e (x,e) -> Fun_e (x,e)
  | FunCall_e (ef,e1) ->
    (match eval ef with
      | Fun_e (x,e2) -> eval (substitute (eval e1) x e2)
      | _ -> raise TypeError)
```

This looks like the case for let!
Let and Lambda

let x = 1 in x+41

\[\rightarrow\]

1+41

\[\rightarrow\]

42

(fun x -> x+41) 1

\[\rightarrow\]

1+41

\[\rightarrow\]

42

In general:

(fun x -> e2) e1 == let x = e1 in e2
So we could write:

```ml
let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
  | Let_e(x,e1,e2) -> eval (FunCall (Fun_e (x,e2), e1))
  | Var_e x -> raise (UnboundVariable x)
  | Fun_e (x,e) -> Fun_e (x,e)
  | FunCall_e (ef,e2) ->
    (match eval ef with
      | Fun_e (x,e1) -> eval (substitute (eval e1) x e2)
      | _ -> raise TypeError)
```

In programming-languages speak: “Let is syntactic sugar for a function call”

**Syntactic sugar**: A new feature defined by a simple, local transformation.
Recursive definitions

definitions

```
type exp = Int_e of int | Op_e of exp * op * exp
  | Var_e of variable | Let_e of variable * exp * exp |
  | Fun_e of variable * exp | FunCall_e of exp * exp |
  | Rec_e of variable * variable * exp ;;
```

```
let rec f x = f (x+1) in f 3
```

```
let f = rec f x -> f (x+1)) in f 3
```

```
let g = rec f x -> f (x+1)) in g 3
```

```
let e ("g,
    Rec_e ("f", "x",
      FunCall_e (Var_e "f", Op_e (Var_e "x", Plus, Int_e 1))
    ),
    FunCall (Var_e "g", Int_e 3)
)
```
Recursive definitions

```ocaml
type exp = Int_e of int | Op_e of exp * op * exp | Var_e of variable | Let_e of variable * exp * exp | Fun_e of variable * exp | FunCall_e of exp * exp | Rec_e of variable * variable * exp ;;

let is_value (e:exp) : bool =
  match e with
  | Int_e _ -> true
  | Fun_e (_,_) -> true
  | Rec_e of (_,_,_) -> true
  | (Op_e (_,_,_) | Let_e (_,_,_) | Var_e _ | FunCall_e (_,_) ) -> false ;;
```
Interlude: Notation for Substitution

“Substitute value \( v \) for variable \( x \) in expression \( e \):” \( e [ v / x ] \)

examples of substitution:

\[
(x + y) [7/y] \quad \text{is} \quad (x + 7)
\]

\[
(\text{let } x = 30 \text{ in let } y = 40 \text{ in } x + y) [7/y] \quad \text{is} \quad (\text{let } x = 30 \text{ in let } y = 40 \text{ in } x + y)
\]

\[
(\text{let } y = y \text{ in let } y = y \text{ in } y + y) [7/y] \quad \text{is} \quad (\text{let } y = 7 \text{ in let } y = y \text{ in } y + y)
\]
Basic evaluation rule for recursive functions:

\[(\text{rec } f \ x = \text{body}) \ \text{arg} \quad \rightarrow \quad \text{body} \ [\text{arg/x}] \ [\text{rec } f \ x = \text{body}/f]\]
Evaluating Recursive Functions

Start out with a let bound to a recursive function:

```
let g =
  rec f x ->
  if x <= 0 then x
  else x + f (x-1)
in g 3
```

The Substitution:

```
g 3 [rec f x ->
  if x <= 0 then x
  else x + f (x-1)  / g]
```

The Result:

```
(rec f x ->
  if x <= 0 then x else x + f (x-1)) 3
```
Evaluating Recursive Functions

Recursive Function Call:

\[(\text{rec } f \, x \to \\
\text{if } x \leq 0 \text{ then } x \text{ else } x + f(x-1)) \, 3\]

The Substitution:

\[(\text{if } x \leq 0 \text{ then } x \text{ else } x + f(x-1)) \]
\[
[ \text{rec } f \, x \to \\
\text{if } x \leq 0 \text{ then } x \\
\text{else } x + f(x-1) \] / f \]
\[
[ 3 / x ]
\]

Substitute argument for parameter
Substitute entire function for function name

The Result:

\[(\text{if } 3 \leq 0 \text{ then } 3 \text{ else } 3 + (\text{rec } f \, x \to \\
\text{if } x \leq 0 \text{ then } x \\
\text{else } x + f(x-1)) \,(3-1))\]
let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
  | Var_e x -> raise (UnboundVariable x)
  | Fun_e (x,e) -> Fun_e (x,e)
  | FunCall_e (e1,e2) ->
    (match eval e1 with
     | Fun_e (x,e) ->
       let v = eval e2 in
       substitute e x v
     | (Rec_e (f,x,e)) as g ->
       let v = eval e2 in
       substitute (substitute e x v) f g
     | _ -> raise TypeError
    )
More Evaluation

(rec fact n = if n <= 1 then 1 else n * fact(n-1)) 3

--> 
if 3 < 1 then 1 else
  3 * (rec fact n = if ... then ... else ...) (3-1)

--> 
3 * (rec fact n = if ... ) (3-1)

--> 
3 * (rec fact n = if ... ) 2

--> 
3 * (if 2 <= 1 then 1 else 2 * (rec fact n = ...)(2-1))

--> 
3 * (2 * (rec fact n = ...)(2-1))

--> 
3 * (2 * (rec fact n = ...)(1))

--> 
3 * 2 * (if 1 <= 1 then 1 else 1 * (rec fact ...)(1-1))

--> 
3 * 2 * 1
A MATHEMATICAL DEFINITION*
OF O’CAML EVALUATION

* it’s a partial definition and this is a big topic; for more, see COS 510
OCaml code can give a language semantics

- **advantage**: it can be executed, so we can try it out
- **advantage**: it is amazingly concise
  - especially compared to what you would have written in Java
- **disadvantage**: it is a little ugly to operate over concrete ML datatypes like “Op_e(e1,Plus,e2)” as opposed to “e1 + e2”

PL researchers have developed their own, relatively standard notation for writing down how programs execute

- it has a mathematical “feel” that makes PL researchers feel special and gives us *goosebumps* inside
- it operates over abstract expression syntax like “e1 + e2”
- it is useful to know this notation if you want to read specifications of programming language semantics
  - eg: Standard ML (of which OCaml is a descendent) has a formal definition given in this notation
  - eg: most papers in the conference POPL (ACM Principles of Prog. Lang.)
Our goal is to explain how an expression $e$ evaluates to a value $v$.

In other words, we want to define a mathematical *relation* between pairs of expressions and values.
We define the “evaluates to” relation using a set of (inductive) rules that allow us to prove that a particular (expression, value) pair is part of the relation.

A rule looks like this:

premise 1  premise 2  ...  premise 3  

conclusion

You read a rule like this:

– “if premise 1 can be proven and premise 2 can be proven and ... and premise n can be proven then conclusion can be proven”

Some rules have no premises
– this means their conclusions are always true
– we call such rules “axioms” or “base cases”
As a rule:

\[
e_1 \rightarrow v_1 \quad e_2 \rightarrow v_2 \quad \text{eval}_\text{op}(v_1, \text{op}, v_2) = v' \\
e_1 \text{ op } e_2 \rightarrow v'
\]

In English:

“If \(e_1\) evaluates to \(v_1\) and \(e_2\) evaluates to \(v_2\) and \(\text{eval}_\text{op}(v_1, \text{op}, v_2)\) is equal to \(v'\) then \(e_1 \text{ op } e_2\) evaluates to \(v'\)"

In code:

```ml
let rec eval (e:exp) : exp =
  match e with
  | Op_e(e1,op,e2) -> let v1 = eval e1 in
    let v2 = eval e2 in
    let v' = eval_op v1 op v2 in
    v'
```

(not great programming style but syntactically very similar to the rule)
An example rule

As a rule:

\[ i \in \mathbb{Z} \quad \frac{}{i \rightarrow i} \]

asserts \( i \) is an integer

In English:

“If the expression is an integer value, it evaluates to itself.”

In code:

```ml
let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  ...
```

In English:

“\( i \in \mathbb{Z} \rightarrow i \) asserts \( i \) is an integer.”
An example rule concerning evaluation

As a rule:

\[
\begin{align*}
\text{e1} & \rightarrow v1 \\
\text{e2} & \rightarrow v2
\end{align*}
\]

\[
\text{let } x = \text{e1 in e2} \rightarrow v2
\]

In English:

“If \text{e1} evaluates to \text{v1} and \text{e2} with \text{v1} substituted for \text{x} evaluates to \text{v2} then \text{let x=e1 in e2 evaluates to v2}.”

In code:

```ml
let rec eval (e:exp) : exp =
  match e with
  | Let_e(x,e1,e2) -> let v1 = eval e1 in
    eval (substitute v1 x e2)
  ...
```
An example rule concerning evaluation

As a rule:

\[ \lambda x.e \rightarrow \lambda x.e \]

typical “lambda” notation for a function with argument \( x \), body \( e \)

In English:

“A function value evaluates to itself.”

In code:

```ml
let rec eval (e:exp) : exp =
  match e with
  ...
  ...
  | Fun_e (x,e) -> Fun_e (x,e)
  ...
```
An example rule concerning evaluation

As a rule:

\[ \begin{align*}
  e_1 \rightarrow & \lambda x. e \\
  e_2 \rightarrow & v_2 \\
  e[v_2/x] \rightarrow & v \\
  e_1 \, e_2 \rightarrow & v
\end{align*} \]

In English:

“if \( e_1 \) evaluates to a function with argument \( x \) and body \( e \) and \( e_2 \) evaluates to a value \( v_2 \) and \( e \) with \( v_2 \) substituted for \( x \) evaluates to \( v \) then \( e_1 \) applied to \( e_2 \) evaluates to \( v \)”

In code:

```ocaml
let rec eval (e:exp) : exp =
  match e with
  ..
  | FunCall_e (e1,e2) ->
    (match eval e1 with
     | Fun_e (x,e) -> eval (substitute e x (eval e2))
     | ...) )
  ...
```
An example rule concerning evaluation

As a rule:

\[
\begin{align*}
\text{e1} & \rightarrow \text{rec } f \ x = e \\
\text{e2} & \rightarrow v \\
\text{e[rec } f \ x = e/f][v/x] & \rightarrow v_2 \\
\text{e1 e2} & \rightarrow v_2
\end{align*}
\]

In English:

“uggh”

In code:

```ml
let rec eval (e:exp) : exp =
  match e with
  ...
  | (Rec_e (f,x,e)) as g ->
    let v = eval e2 in
    substitute (substitute e x v) f g
```
Comparison: Code vs. Rules

Almost isomorphic:

- one rule per pattern-matching clause
- recursive call to eval whenever there is a \( \rightarrow \) premise in a rule
- what’s the main difference?

complete eval code:

```ocaml
let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
  | Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
  | Var_e x -> raise (UnboundVariable x)
  | Fun_e (x,e) -> Fun_e (x,e)
  | FunCall_e (e1,e2) ->
    (match eval e1
    | Fun_e (x,e) -> eval (Let_e (x,e2,e))
    | _ -> raise TypeError)
  | LetRec_e (x,e1,e2) ->
    (Rec_e (f,x,e)) as g ->
    let v = eval e2 in
    substitute (substitute e x v) f g
```

complete set of rules:

```rawtext
\begin{align*}
\text{eval_op} & (\text{v1}, \text{op}, \text{v2}) \rightarrow \text{v} \\
\text{let } x = \text{e1 in } \text{e2 } & \rightarrow \text{v2} \\
\text{λx.e} & \rightarrow \text{λx.e} \\
\text{let } x = \text{e1 in } \text{e2 } & \rightarrow \text{v2} \\
\text{rec } \text{f} = \text{e} & \rightarrow \text{v3} \\
\end{align*}
```
### Comparison: Code vs. Rules

#### complete eval code:

```ocaml
let rec eval (e:exp) : exp =
match e with
| Int_e i -> Int_e i
| Op_e(e1,op,e2) -> eval_op (eval e1) op (eval e2)
| Let_e(x,e1,e2) -> eval (substitute (eval e1) x e2)
| Var_e x -> raise (UnboundVariable x)
| Fun_e (x,e) -> Fun_e (x,e)
| FunCall_e (e1,e2) ->
  (match eval e1
   | Fun_e (x,e) -> eval (Let_e (x,e2,e))
   | _ -> raise TypeError)
| LetRec_e (x,e1,e2) ->
  (Rec_e (f,x,e)) as g ->
  let v = eval e2 in
  substitute (substitute e x v) f g
```

#### complete set of rules:

- \( \frac{i \in \mathbb{Z}}{i \rightarrow i} \)
- \( \frac{e1 \rightarrow v1 \quad e2 \rightarrow v2 \quad \text{eval}_{\text{op}} (v1, \text{op}, v2) = v}{e1 \text{ op } e2 \rightarrow v} \)
- \( \frac{e1 \rightarrow v1 \quad e2 [v1/x] \rightarrow v2}{\text{let } x = e1 \text{ in } e2 \rightarrow v2} \)
- \( \frac{\lambda x.e}{\lambda x.e} \)
- \( \frac{e1 \rightarrow \lambda x.e \quad e2 \rightarrow v2 \quad e[v2/x] \rightarrow v}{e1 \text{ e2} \rightarrow v} \)
- \( \frac{e1 \rightarrow \text{rec } f \ x = e \quad e2 \rightarrow v2 \quad e[\text{rec } f \ x = e/f][v2/x] \rightarrow v3}{e1 \text{ e2} \rightarrow v3} \)

- There’s no formal rule for handling free variables
- No rule for evaluating function calls when a non-function in the caller position
- In general, **no rule when further evaluation is impossible**
  - the rules express the *legal evaluations* and say nothing about what to do in error situations
  - the code handles the error situations by raising exceptions
  - type theorists prove that well-typed programs don’t run in to undefined cases
Summary

• We can reason about OCaml programs using a substitution model.
  – integers, bools, strings, chars, and functions are values
  – value rule: values evaluate to themselves
  – let rule: “let x = e1 in e2” : substitute e1’s value for x into e2
  – fun call rule: “(fun x -> e2) e1”: substitute e1’s value for x into e2
  – rec call rule: “(rec x = e1) e2” : like fun call rule, but also substitute recursive function for name of function
    • To unwind: substitute (rec x = e1) for x in e1

• We can make the evaluation model precise by building an interpreter and using that interpreter as a specification of the language semantics.

• We can also specify the evaluation model using a set of inference rules
  – more on this in COS 510
Some Final Words

• The substitution model is only a model.
  – it does not accurately model all of OCaml’s features
    • I/O, exceptions, mutation, concurrency, ...
    • we can build models of these things, but they aren’t as simple.
    • even substitution was tricky to formalize!
• It’s useful for reasoning about higher-order functions, correctness of algorithms, and optimizations.
  – we can use it to formally prove that, for instance:
    • map f (map g xs) == map (comp f g) xs
    • proof: by induction on the length of the list xs, using the definitions of the substitution model.
  – we often model complicated systems (e.g., protocols) using a small functional language and substitution-based evaluation.
• It is not useful for reasoning about execution time or space
  – more complex models needed there
END