Implementing OCaml in OCaml

Part 3: More Features, More Fun!

Speaker: David Walker

COS 326

Princeton University

Scaling up the Language

```
type exp = Int of int | Op of exp * op * exp
   | Var of variable | Let of variable * exp * exp
   | Fun of variable * exp | App of exp * exp
```

Scaling up the Language

```
type exp = Int of int | Op of exp * op * exp
   | Var of variable | Let of variable * exp * exp
   | Fun of variable * exp | App of exp * exp
```

OCaml’s
fun x -> e
is represented as
Fun(x,e)

Scaling up the Language

```
type exp = Int of int | Op of exp * op * exp
   | Var of variable | Let of variable * exp * exp
   | Fun of variable * exp | App of exp * exp
```

A function "application"
(ie: function call)
fact 3
is implemented as
App (Var “fact”, Int 3)

Scaling up the Language

```
type exp = Int of int | Op of exp * op * exp
   | Var of variable | Let of variable * exp * exp
   | Fun of variable * exp | App of exp * exp

let is_value (e:exp) : bool =
  match e with
   | Int _ -> true
   | Fun (_,_) -> true
   | (  Op (_,_,_)
      | Let (_,_,_)
      | Var _
      | FunApp (_,_) ) -> false
```

Functions are values!

Easy Exam Question: What value does the OCaml interpreter produce when it evaluates the expression (fun x -> 3)?

Answer: the value produced is (fun x -> 3)

Scaling up the Language

```
type exp = Int of int | Op of exp * op * exp
  | Var of variable | Let of variable * exp * exp
  | Fun of variable * exp | App of exp * exp

let is_value (e:exp) : bool =
  match e with
  | Int _ -> true
  | Fun (_,_) -> true
  | ( Op (_,_,_)
    | Let (_,_,_)
    | Var _
    | App (_,_) ) -> false
```

Function Apps are not values.

Scaling up the Language

```
let rec eval (e:exp) : exp =
  match e with
  | Int i -> Int i
  | Op(e1,op,e2) -> eval_op (eval e1) op (eval e2)
  | Let(x,e1,e2) -> eval (substitute (eval e1) x e2)
  | Var x -> raise (UnboundVariable x)
  | Fun (x,e) -> Fun (x,e)
  | App (e1,e2) ->
      (match eval e1, eval e2 with
       | Fun (x,e), v2 -> eval (substitute v2 x e)
       | _ -> raise TypeError)
```

Simplifying a little

```
let rec eval (e:exp) : exp =
  match e with
  | Int i -> Int i
  | Op(e1,op,e2) -> eval_op (eval e1) op (eval e2)
  | Let(x,e1,e2) -> eval (substitute (eval e1) x e2)
  | Var x -> raise (UnboundVariable x)
  | Fun (x,e) -> Fun (x,e)
  | App (e1,e2) ->
      (match eval e1 with
       | Fun (x,e) -> eval (substitute (eval e2) x e)
       | _ -> raise TypeError)
```

We don’t really need
to pattern-match on e2.
Just evaluate here

Simplifying a little

```
let rec eval (e:exp) : exp =
  match e with
   | Int i -> Int i
   | Op(e1,op,e2) -> eval_op (eval e1) op (eval e2)
   | Let(x,e1,e2) -> eval (substitute (eval e1) x e2)
   | Var x -> raise (UnboundVariable x)
   | Fun (x,e) -> Fun (x,e)
   | App (ef,e1) ->
        (match eval ef with
         | Fun (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
-->
1+41
-->
42
```

```
(fun x -> x+41) 1
-->
1+41
-->
42
```

In general:

```
let x = e1 in e2 == (fun x -> e2) e1
```

So we could write:

```
let rec eval (e:exp) : exp =
  match e with
   | Int i -> Int i
   | Op(e1,op,e2) -> eval_op (eval e1) op (eval e2)
   | Let(x,e1,e2) -> eval (App (Fun (x,e2), e1))
   | Var x -> raise (UnboundVariable x)
   | Fun (x,e) -> Fun (x,e)
   | App (ef,e2) ->
        (match eval ef with
         | Fun (x,e1) -> eval (substitute (eval e1) x e2)
         | _ -> raise TypeError)
```

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

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

Recursive Function Definitions in OCaml

A "let rec" definition does two independent things

The "rec" part: allows f to show up in the function body

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

The "let" part: allows f to show up in the following expression

Recursive Function Definitions in OCaml

In our interpreter, we are going to split those things apart into two different constructs

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

A new construct for our interpreter: a recursive function

Often called the "Principle of Orthogonality"

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

Just an ordinary "let"

Recursive definitions

```
type exp = Int of int | Op of exp * op * exp
  | Var of variable | Let of variable * exp * exp
  | Fun of variable * exp | App of exp * exp
  | Rec of variable * variable * exp
```

function name (eg: "f")

argument name (eg: "x")

body of the function

Recursive Function Definitions in OCaml

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

```
Let (“f”,
  Rec (“f”, “x”,
    App (Var “f”, Op (Var “x”, Plus, Int 1))
 ),
 App (Var “f”, Int 3)
)
```

Recursive Function Definitions in OCaml

To avoid confusion, let's rename the variable used in the following expression (but not the function body).

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

```
Let (“g”,
  Rec (“f”, “x”,
    App (Var “f”, Op (Var “x”, Plus, Int 1))
 ),
 App (Var “g”, Int 3)
)
```

Recursive Values

```
type exp = Int of int | Op of exp * op * exp
   | Var of variable | Let of variable * exp * exp
   | Fun of variable * exp | App of exp * exp
   | Rec of variable * variable * exp
```

Notice that the following values are the same:

`fun x = x+1`

`rec f x = x+1`

`rec g x = x+1`

`rec i_dont_care x = x+1`

So now that we have the "Rec" form in our syntax, we could delete the "Fun" form as it is unnecessary and can be encoded:

`Fun(var, body)` → `Rec("_", var, body)`

Recursive definitions

```
type exp = Int of int | Op of exp * op * exp
   | Var of variable | Let of variable * exp * exp
   | Fun of variable * exp | App of exp * exp
   | Rec of variable * variable * exp
```

```
let is_value (e:exp) : bool =
  match e with
   | Int _ -> true
   | Fun (_,_) -> true
   | Rec of (_,_,_) -> true
   | (Op (_,_,_) | Let (_,_,_) |
      Var _ | App (_,_) ) -> false
```

Interlude: Notation for Substitution

"Substitute value v for variable x in expression e:" e [v / x]

Examples of substitution:

(x + y) [7/y]	is	(x + 7)
(let x =30 in let y=40 in x + y) [7/y]	is	(let x =30 in let y=40 in x + y)
(let y = y in let y = y in y + y) [7/y]	is	(let y = 7 in let y = y in y + y)

Evaluating Recursive Functions

Basic evaluation rule for recursive functions:

`(rec f x = body) arg --> body [arg/x] [rec f x = body/f]`

argument value substituted
for parameter

entire function substituted
for function name

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 App:

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

The Substitution:

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

Substitute argument for parameter

Substitute entire function for function name

The Result:

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

Evaluating Recursive Functions

```
let rec eval (e:exp) : exp =
  match e with
   | Int i -> Int i
   | Op(e1,op,e2) -> eval_op (eval e1) op (eval e2)
   | Let(x,e1,e2) -> eval (substitute (eval e1) x e2)
   | Var x -> raise (UnboundVariable x)
   | Fun (x,e) -> Fun (x,e)
   | App (e1,e2) ->
       (match eval e1 with
        | Fun (x,e) ->
             let v = eval e2 in
             substitute e x v

        | (Rec (f,x,e)) as f_val ->
             let v = eval e2 in
             let body = substitute f_val f
                               (substitute v x e) in
             eval body

        | _ -> raise TypeError)
```

pattern as *x*

match the pattern and binds x to value

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
```

Exercise 1

(a) What is the result of the following substitution? In your answer, rename variables so you have as many unique variable names as possible.

```
(let g = rec f (x) =  let g = fun x -> g (f x) in 0 in g (fun g -> g)) [(fun g -> g + 1)/g]
```

(b) What are the free variables of the following expression?

```
let g = rec f (x) =  let g = fun x -> g (f x) in 0 in g (fun g -> g)
```

(c) What are the free variables of your answer to (a)? More generally, how are the free variables of the expression e and the expression (e[v/x]) related?

Exercise 2

Try extending the language and its evaluation system with:

- booleans (true, false, and, or, not, if)
- pairs (with pair creation and field extraction operations)