Functional Programming

February 21, 2014

Riccardo Pucella

What we’ve done till now

We’ve developed a core interpreter

— parser for surface syntax

— internal representation

— evaluator for execution

Object language: a simple expression language

— basically a glorified calculator

What’s next?

We take our core interpreter, and extend it in various directions to capture existing programming models

Next up: we beef up expressions
— leads to functional programming languages

(Later: instead of beefing up expressions, we add statements that can modify the state)

Functional programming

Functional programming languages are characterized by:

— everything is an expression returning a value

— functions are first-class citizens

  – they can be created and passed around like any other value without any restriction

Pedantically:

— evaluation has no side-effects

— evaluation is lazy (call-by-name)

SML is mostly functional

— Everything is an expression to be evaluated
— Every expression yields a value
— Functions are first-class:
  — they can be passed as arguments
  — they can be returned from functions
  — they can be created “on the fly”

We’ve been using first-class functions already, as a way to “fake” multi-argument functions

Higher-order functions

A higher-order function is a function that takes another function as argument

```
fun succ x = x + 1                : int -> int

fun twice f x = f (f x)           : (int -> int) -> int -> int

twice succ 10 →
   succ (succ 10) →
      12
```

Utility of higher-order functions

Higher-order functions can capture the structure of a family of functions

```
fun doubles [] = []
  | doubles (x::xs) = (2*x)::(doubles xs)

fun lastDigits [] = []
  | lastDigits (x::xs) = (x mod 10)::(lastDigits xs)
```

Utility of higher-order functions

Higher-order functions can capture the structure of a family of functions

```
fun double x = 2 * x
fun lastDigit x = x mod 10

fun doubles [] = []
  | doubles (x::xs) = (double x)::(doubles xs)

fun lastDigits [] = []
  | lastDigits (x::xs) = (lastDigit x)::(lastDigits xs)
```

Utility of higher-order functions

Higher-order functions can capture the structure of a family of functions

```
fun double x = 2 * x
fun lastDigit x = x mod 10

fun doubles [] = []
  | doubles (x::xs) = (double x)::(doubles xs)

fun lastDigits [] = []
  | lastDigits (x::xs) = (lastDigit x)::(lastDigits xs)
```

Utility of higher-order functions

Higher-order functions can capture the structure of a family of functions

```
fun double x = 2 * x
fun lastDigit x = x mod 10

fun map f [] = []
  | map f (x::xs) = (f x)::(map f xs)
                                  : ('a -> 'b) -> 'a list -> 'b list

fun doubles xs = map double xs
fun lastDigits xs = map lastDigit xs
```

Another example: filtering

```
fun evens [] = []
  | evens (x::xs) = if (x mod 2 = 0) then
                       x::(evens xs)
                    else evens xs
```

Another example: filtering

```
fun isEven x = (x mod 2 = 0)

fun evens [] = []
  | evens (x::xs) = if (isEven x) then
                      x::(evens xs)
                    else evens xs
```

Another example: filtering

```
fun isEven x = (x mod 2 = 0)

fun filter p [] = []
  | filter p (x::xs) = if (p x) then
                          x::(filter p xs)
                        else filter p xs

fun evens xs = filter isEven xs
```

Another example: reducing

```
fun sum [] = 0
  | sum (x::xs) = x + (sum xs)

fun flatten [] = []
  | flatten (xs::xss) = xs @ (flatten xss)
```

Another example: reducing

```
fun add a b = a + b
fun append a b = a @ b

fun sum [] = 0
  | sum (x::xs) = add x (sum xs)

fun flatten [] = []
  | flatten (xs::xss) = append xs (flatten xss)
```

Another example: reducing

```
fun add a b = a + b
fun append a b = a @ b

fun reduce f b [] = b
  | reduce f b (x::xs) = f x (reduce f b xs)

fun sum xs = reduce add 0 xs

fun flatten xss = reduce append [] xss
```

Another example: reducing

```
fun add a b = a + b
fun append a b = a @ b

fun reduce f b [] = b
  | reduce f b (x::xs) = f x (reduce f b xs)

fun sum

fun fla
```

```
fun map f xs = let
   fun applyf x r = (f x)::r
 in
   reduce applyf [] xs
 end
```

Another example: reducing

```
fun add a b = a + b
fun append a b = a @ b

fun foldr  f b [] = b
  | foldr  f b (x::xs) = f x (foldr  f b xs)

fun sum xs = foldr  add 0 xs

fun flatten xss = foldr  append [] xss
```

`map`, `foldr` built-in — `filter` as `List.filter`

Anonymous functions

Giving a name to a function just to pass it to another function is a pain

Sometimes, you just want a function without needing it to have a name

```
fun double x = 2 * x

fun doubles xs = map double xs
```

Anonymous functions

Giving a name to a function just to pass it to another function is a pain

Sometimes, you just want a function without needing it to have a name

```
fun double x = 2 * x
```

```
fun doubles xs = map (fn x => 2 * x) xs
```

Anonymous functions

Giving a n

another f

Sometim

needing t

```
fun double x = 2 * x
```

```
fun doubles xs = map (fn x => 2 * x) xs
```

Syntax:

`fn identifier => expression`

More generally:

`fn pattern => expression`

Anonymous functions

Giving a name to a function just to pass it to another functio

Sometimes, yo ut needing it to h

~~fun double x~~

Obvious restriction:

Anonymous functions cannot be recursive!

```
fun doubles xs = map (fn x => 2 * x) xs
```

Exercises

```
fun add_to_each n xs = <use map>
      : int -> int list -> int list

fun count_zeros xs = <use foldr>
      : int list -> int

fun reverse xs = <use foldr>
      : 'a list -> 'a list

fun last xs = <use foldr>
      : int list -> int option
```

Functions returning functions

Functions can be returned as values:

```
fun double x = 2 * x

fun triple x = 3 * x

…
```

Functions returning functions

Functions can be returned as values:

```
fun makeMultBy n = let
  fun multiply x = n * x
in
  multiply
end

val double = makeMultBy 2

val triple = makeMultBy 3
```

Functions returning functions

Functions can be returned as values:

```
fun makeMultBy n = (fn x => n * x)
```

```
val double = makeMultBy 2

val triple = makeMultBy 3
```

Functions returning functions

Functions can be returned as values:

```
fun makeMultBy n = (fn x => n * x)

    : int -> (int -> int)



val double = makeMultBy 2

val triple = makeMultBy 3
```

Functions returning functions

Functions can be returned as values:

```
fun makeMultBy n = (fn x => n * x)

    : int -> int -> int



val double = makeMultBy 2

val triple = makeMultBy 3
```

Functions returning functions

Functions can be returned as values:

```
fun makeMultBy n x = n * x

    : int -> int -> int



val double = makeMultBy 2

val triple = makeMultBy 3
```

Functions returning functions

Functions can be returned as values:

```
fun makeMultBy n x = n * x

    : int -> i

val double =

val triple =
```

What we've been calling multi-argument functions are really functions of one argument that return functions of the rest of the arguments

Curried functions

Composing functions

```
fun compose f g = (fn x => g (f x))

fun double x = 2 * x
fun succ x = x + 1
```

```
- (compose double succ) 10;
val it = 21 : int
- (compose succ double) 10;
val it = 22 : int
```

Composing functions

```
fun compose f g x = g (f x)

fun double x = 2 * x
fun succ x = x + 1
```

```
- (compose double succ) 10;
val it = 21 : int
- (compose succ double) 10;
val it = 22 : int
```

Composing functions

```
fun compose
```

```
fun double
fun succ x
```

compose is built into SML as an infix o

```
(double o succ) 10

(succ o double) 10
```

```
- (compose double succ) 10;
val it = 21 : int
- (compose succ double) 10;
val it = 22 : int
```

Composing option-valued functions

```
fun choose2 f g x =
  (case f x
    of NONE => g x
     | s => s)
```

Returns the first of
(f x)
(g x)
that does not return NONE

```
choose2 : ('a -> 'b option) ->
            ('a -> 'b option) ->
              ('a -> 'b option)
```

Composing option-valued functions

```
fun choose2 f g x =
  (case f x
    of NONE => g x
     | s => s)
```

Returns the first of
(f x)
(g x)
that does not return NONE

```
fun choose [] x = NONE
  | choose (f::fs) x =
      (case f x
        of NONE => choose fs x
         | s => s)
```

Composing option-valued functions

```
fun seq f g x =
  (case f x
    of NONE => NONE
     | SOME v => g v)
```

Returns `g` applied to the result of `(f x)` if neither is NONE

Returns NONE otherwise

```
seq : ('a -> 'b option) ->
          ('b -> 'c option) ->
            ('a -> 'c option)
```

A functional object language

```
datatype expr = EVal of value
              | EIf of expr * expr * expr
              | ELet of string * expr * expr
              | EIdent of string
              | EApp of expr * expr

datatype value = VInt of int
               | VBool of bool
               | VFun of string * expr
```

Surface syntax

expr ::= *aterm aterm_list*

aterm ::= ***integer***
true
false
symbol
if *expr* **then** *expr* **else** *expr*
let ***symbol*** **=** *expr* **in** *expr*
let ***symbol symbol*** **=** *expr* **in** *expr*

aterm_list ::= *aterm aterm_list*
<empty>

Surface syntax

```
expr ::= aterm aterm_list

aterm ::= integer
          true
          false
          symbol
          if expr then expr else expr
          let symbol = expr in expr
          let symbol symbol = expr in expr

aterm_list ::= ate
               <em
```

Functions of one argument only

— no currying either

Surface syntax

expr ::= *aterm* *aterm_list*

Produces

ELet (s,$e1$,$e2$)

aterm ::= ***integer***
true
false
symbol
if *expr* **then** *expr* *expr*
let ***symbol***$_s$ **=** *expr*$_{e1}$ **in** *expr*$_{e2}$
let ***symbol*** ***symbol*** **=** *expr* **in** *expr*

aterm_list ::= *aterm* *aterm_list*
<empty>

Surface syntax

expr ::= *aterm aterm_list*

aterm ::= ***integer***
true
false
symbol
if *expr* **then** *expr* … *pr*
let ***symbol*** **=** *expr* **i**… *xpr*
let ***symbol***$_{s1}$ ***symbol***$_{s2}$ **=** *expr*$_{e1}$ **in** *expr*$_{e2}$

aterm_list ::= *aterm aterm_list*
<empty>

Produces

```
ELet (s1,
      EVal (VFun (s2,e1)),
      e2)
```

Evaluation function

```
fun eval _ (EVal v) = v
  | eval env (EIf (e,f,g)) = evalIf env (eval env e) f g
  | eval env (ELet (name,e,f)) = evalLet env name (eval env e) f
  | eval env (EIdent n) = lookup n env
  | eval env (EApp (e1,e2)) = evalApp env (eval env e1) (eval env e2)

and evalApp env (VFun (p,body)) v = eval env (subst body p (EVal v))
  | evalApp env _ _ = evalError "cannot apply non-functional value"

and evalIf env (VBool true) f g = eval env f
  | evalIf env (VBool false) f g = eval env g
  | evalIf _ _ _ _ = evalError "evalIf"

and evalLet env id v body = eval env (subst body id (EVal v))
```

Evaluation function

```
fun eval _ (EVal v) = v
  | eval env (EIf (e,f,g)) = evalIf env (eval env e) f g
  | eval env (ELet (name,e,f)) = evalLet env name (eval env e) f
  | eval env (EIdent n) = lookup n env
  | eval env (EApp (e1,e2)) = evalApp env (eval env e1) (eval env e2)

and evalApp env (VFun (p,body)) v = eval env (subst body p (EVal v))
  | evalApp env _ _ = evalError "cannot apply non-functional value"

and evalIf env (VBool true) f g = eval env f
  | evalIf env (VBool f
  | evalIf _ _ _ _ = eval

and evalLet env id v body
```

This is just the evaluation of `ECallE` from homework 2!

Substitution function

```
fun subst (EVal (VFun (p,body))) id e =
      if id = p then
        EVal (VFun (p,body))
      else EVal (VFun (p, subst body id e))
  | subst (EVal v) id e = EVal v
  | subst (EIf (e1,e2,h)) id e =
      EIf (subst e1 id e, subst e2 id e, subst h id e)
  | subst (ELet (id',e1,e2)) id e =
      if id = id' then
        ELet (id',subst e1 id e, e2)
      else ELet (id',subst e1 id e, subst e2 id e)
  | subst (EIdent id') id e =
      if id = id' then e else EIdent id'
  | subst (EApp (e1,e2)) id e = EApp (subst e1 id e, subst e2 id e)
```

This almost works

Something missing: primitive operations

— easy to add

— could bake them into the IR

— better approach next time

BIGGER PROBLEM: recursion!

— What happens if you try to evaluate

`let f x = f x in f 10` ?

Think about this for next time