The Environment Model

February 25, 2014

Riccardo Pucella

Last time

A simple functional language:

- — First class functions
- — Multiargument functions via currying
- — Substitution model

Internal Representation

```
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
          l
          l

aterm_list
```

OCaml / Haskell syntax
(to distinguish from SML)

Note: `f 10 20 30` interpreted as
`((f 10) 20) 30`

to match the use of currying for functions

Surface syntax

expr ::= *aterm aterm_list*

aterm ::= ***integer***
true
false

aterm

Easy exercise:

add syntax for anonymous functions

\ sym -> expr

E.g. \ x -> x

should parse as `EVal (VFun (“x”, EIdent “x”))`

Issues

(1) Dealing with *primitive operations*

— no primitives in our object language

(2) Dealing with *recursion*

— function definition

```
let f x = f x in f 0
```

was interpreted as

```
let f = \x -> f x in f 0
```

caused an error instead of looping

Issue 1: primitives

Primitives are annoying to implement, but of course important

— the car trunk problem

Key: you want to be able to treat primitives like any other function

— pass them as arguments to functions

— curried (partially applicable)

Many primitives are infix operators: +, *, &&

Issue 1: primitives

Primitives are annoying to implement, but of course important

— the car trunk problem

Key: you want to b
like any other func

— pass them as

— curried (parti

Surface syntax problem

(How do you partially apply infix operations?)

Many primitives are infix operators: +, *, &&

Primitives as special values

Two approaches to implementing primitives:

(1) treat them as special values

– recognize them as the first argument to `EApp`

(2) treat them as special expressions

Code from Lecture 8: new type of value `VPrim`

— acts like a `VFun`

— instead of `expr` holds SML function

— difficult to curry

Primitives as special expressions

One expression form per primitive

Alternatively:

```
...
| EPrimCall1 of (value -> value) * expr
| EPrimCall2 of (value -> value -> value) * expr * expr
| EPrimCall3 of (value -> value -> value -> value) * …
```

Obvious evaluation rules:

```
| eval env (I.EPrimCall2 (f,e1,e2)) =
                f (eval env e1) (eval env e2)
```

Primitives as special expressions

Treat primitives uniformly with defined functions by adding *wrappers* to the environment:

```
("+", VFun ("a",
            EVal (VFun ("b",
                        EPrimCall2 (primPlus,
                                    EIdent "a",
                                    EIdent "b")))
```

A bit like writing, say, `fun add x y = x + y`

— see code for this Lecture

Issue 2: Recursion

Our code from last time treats

```
let f x = e1 in e2
```

as an abbreviation for

```
let f = (\ x -> e1) in e2
```

and the substitution model tells us that
in a *let*, we substitute *f* into *e2*, not into *e1*

Distinguishing *let* and *letfun*

The Standard ML approach (common)

New IR expression for representing a recursive function binding:

```
...
| ELetFun of string * string * expr * expr
```

`let` *f* *x* `=` *e1* `in` *e2* parses as

`ELetFun (`*f*`,`*x*`,`*e1*`,`*e2*`)`

Substitution into `ELetFun` a bit tricky

Evaluating *letfun*

Obvious approach:

to evaluate

```
let f x = e1 in e2
```

evaluate *e2* where *f* has been replaced by

```
VFun (“x”, e1’)
```

where *e1’* is *e1* where *f* has been replaced by

```
VFun (“x”, e1’)
```

(Oops… try to implement this?)

Evaluating *letfun*

Standard approach: use a *fixed-point operator*

To evaluate `let f x = e1 in e2`

evaluate *e2* after replacing *f* by

```
Z (\f -> \x -> e1)
```

where

```
Z = \f -> (\x -> f (\v -> ((x x) v)))
              (\x -> f (\v -> ((x x) v)))
```

Alternative implementation

The substitution model works fine

— matches intuition

— reasonably clean

But it sucks as an implementation

— inefficient!!

— tied to the internal representation

We can do better—we already have!

Environments

Function environment for *predefined functions*

- Homework 3: add to function environment in the shell

Idea: when we see an identifier that hasn't been substituted for, we look for it in the function environment

Generalize this idea

Environments

Instead of substituting in let or application
— record the binding in the environment
— look up the value of the identifier
when needed

The environment model of evaluation

Example: nested bindings

Environment:

```
let x = 10
 let y = x
  let x = 30
   in x * y
```

Example: nested bindings

Environment:

$x \rightarrow 10$

```
let y = x
 let x = 30
  in x * y
```

Example: nested bindings

Environment:

x → 10

```
let y = x
 let x = 30
  in x * y
```

Example: nested bindings

Environment:

$x \rightarrow 10$
$y \rightarrow 10$

```
let x = 30
 in x * y
```

Example: nested bindings

Environment:

$x \rightarrow 10$
$y \rightarrow 10$
$x \rightarrow 30$

```
in x * y
```

Example: nested bindings

Environment:

x → 10
y → 10
x → 30

```
in x * y
```

Example: nested bindings

Environment:

x $\rightarrow$ 10
y $\rightarrow$ 10
x $\rightarrow$ 30

300

Implementation

Same surface syntax

Same internal representation

No substitution function needed

Evaluation is a *bit* different

Evaluation function

```
fun eval _ (I.EVal v) = v
  | eval env (I.EIf (e,f,g)) = evalIf env (eval env e) f g
  | eval env (I.ELet (n,e,f)) = evalLet env n (eval env e) f
  | eval env (I.ELetFun (n,param,e,f)) = evalLetFun env n param e f
  | eval env (I.EIdent n) = lookup n env
  | eval env (I.EApp (e1,e2)) = evalApp env (eval env e1) (eval env e2)
  | eval env (I.EPrimCall2 (f,e1,e2)) = f (eval env e1) (eval env e2)

and evalApp env (I.VFun (n,body)) v = eval ((n,v)::env) body
  | evalApp env _ _ = evalError "cannot apply non-functional value"

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

and evalLet env id v body = eval ((id,v)::env) body
and evalLetFun env id param expr body = let
      eval ((id,I.VFun (param,expr))::env) body
```

Evaluation function

```
fun eval _ (I.EVal v) = v
  | eval env (I.EIf (e,f,g)) = evalI
  | eval env (I.ELet (n,e,f)) = eval
  | eval env (I.ELetFun (n,param,e,f
  | eval env (I.EIdent n) = lookup n
  | eval env (I.EApp (e1,e2)) = eval
  | eval env (I.EPrimCall2 (f,e1,e2))

and evalApp env (I.VFun (n,body)) v = eval ((n,v)::env) body
  | evalApp env _ _ = evalError "cannot apply non-functional value"

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

and evalLet env id v body = eval ((id,v)::env) body
and evalLetFun env id param expr body = let
      eval ((id,I.VFun (param,expr))::env) body
```

Most recent bindings on the left

Seems to work…

But does it?

Test:

```
let f x = f x in f 0
```

Test:

```
let f x = x in let f = \ x -> f x in f 0
```

Seems to work…

But does it?

Test:

```
let f x = f x in f 0
```

Test:

```
let f x = x in let f = \ x -> f x in f 0
```

Functions seem *always* recursive

— hints at a problem…

What??

```
let x = 10 in
  let f y = x + y in
    f 100
```

```
let x = 10 in
  let f y = x + y in
    let x = 20 in
      f 100
```

What??

```
let x = 10 in
  let f y = x + y in
    f 100
```

```
let x = 10 in
  let f y = x + y in
    let x = 20 in
      f 100
```

This *should* return 110, according to the substitution model

If it returns 120, you've implemented dynamic binding

(Welcome to the 60s)

Binding strategies

A function f may refer to identifiers that are not arguments.
— where do we find the value for these?

Dynamic binding: take the value from the nearest enclosing binding where the function is called

Static binding: take the value from the nearest enclosing binding where the function is defined

(The substitution model gives you static binding)

The upwards FUNARG problem

The problem of implementing static binding in the context of first-class functions is often called the *upwards FUNARG problem*

Solution: *remember the environment that was present when a function was defined*

```
...
| VFun of string * expr
```

The upwards FUNARG problem

The problem of implementing static binding in the context of first-class functions is often called the *upwards FUNARG problem*

Solution: *remember the environment that was present when a function was defined*

```
...
| VFun of string * expr
| VClosure of string * expr *
                   (string * value) list
```

The upwards FUNARG problem

The problem of implementing static binding in
the contex
called the

Solution: *r*
present wh

Still need to do a little bit of work to get recursive functions to work right

One solution in the code for the lecture

Hint: `VRecClosure`

```
...
| VFun of string * expr
| VClosure of string * expr *
                    (string * value) list
```