Introduction to Functions

January 31, 2014

Riccardo Pucella

Last time

- Special forms

- Let bindings (and identifiers)

- Substitution model
    - Substitution model was not call-by-value
    - I left you with a question:
        - What would you need to change?

Call-by-value let bindings

```
datatype expr = EInt of int
              | EVec of int list
              | EAdd of expr * expr
              | EMul of expr * expr
              | ELet of string * expr * expr
              | EIdent of string


datatype value = VInt of int
               | VVec of int list
```

Call-by-value let bindings

```
fun eval (EInt i) = VInt i
  | eval (EVec v) = VVec v
  | eval (EAdd (e,f)) = applyAdd (eval e) (eval f)
  | eval (EMul (e,f)) = applyMul (eval e) (eval f)
  | eval (ELet (id,e,f)) = evalLet id e f
  | eval (EIdent id) = raise EvalError “eval/EIdent”

and evalLet id exp body = eval (subst body id exp)
```

Call-by-value let bindings

```
fun eval (EInt i) = VInt i
  | eval (EVec v) = VVec v
  | eval (EAdd (e,f)) = applyAdd (eval e) (eval f)
  | eval (EMul (e,f)) = applyMul (eval e) (eval f)
  | eval (ELet (id,e,f)) = evalLet id (eval e) f
  | eval (EIdent id) = raise EvalError “eval/EIdent”

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

Call-by-value let bindings

```
fun eval (EInt i) = VInt i
  | eval (EVe
  | eval (EA                                  (eval f)
  | eval (EMu                                 (eval f)
  | eval (ELe                               al e) f
  | eval (EIdent id) = raise EvalErr      "eval/EIdent"

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

subst now takes a value

Original substitution function

```
fun subst (EInt i) id e = EInt i
  | subst (EVec vc) id e = EVec vc
  | subst (EAdd (f,g)) id e =
       EAdd (subst f id e, subst g id e)
  | subst (EMul (f,g)) id e =
       EMul (subst f id e, subst g id e)
  | subst (ELet (id’,f,g)) id e =
       if id = id’
         then ELet (id’, subst f id e, g)
       else ELet (id’, subst f id e, subst g id e)
  | subst (EIdent id’) id e =
       if id = id’
         then e
       else EIdent id
```

Possible substitution function?

```
fun subst (EInt i) id v = EInt i
  | subst (EVec vc) id v = EVec vc
  | subst (EAdd (f,g)) id v =
       EAdd (subst f id v, subst g id v)
  | subst (EMul (f,g)) id v =
       EMul (subst f id v, subst g id v)
  | subst (ELet (id’,f,g)) id v =
       if id = id’
         then ELet (id’, subst f id v, g)
       else ELet (id’, subst f id v, subst g id v)
  | subst (EIdent id’) id v =
       if id = id’
         then v
       else EIdent id
```

Possible substitution function?

```
fun subst (EInt i) id v = EInt i
  | subst (EVec vc) id v = EVec vc
  | subst (EAdd (f,g)) id v =
       EAdd (subst f i
  | subst (EMul (f,g))
       EMul (subst f i
  | subst (ELet (id',f
       if id = id'
         then ELet (id',       f id v, g)
       else ELet (id', s    t f id v, subst g id v)
  | subst (EIdent id') id v =
       if id = id'
         then v
       else EIdent id
```

Type error!

subst should return an expr not a value

Possible substitution function?

```
fun subst (EInt i) id v = EInt i
  | subst (EVec vc) id v = EVec vc
  | subst (EAdd (f,g)) id v =
       EAdd (subst f id v, subst g id v)
  |
         then v
       else EIdent id
```

Keep substitution function as before

Change the internal representation

Call-by-value let bindings (2)

```
datatype expr = EInt of int
              | EVec of int list
              | EAdd of expr * expr
              | EMul of expr * expr
              | ELet of string * expr * expr
              | EIdent of string


datatype value = VInt of int
               | VVec of int list
```

Call-by-value let bindings (2)

```
datatype expr = EVal of value

              | EAdd of expr * expr
              | EMul of expr * expr
              | ELet of string * expr * expr
              | EIdent of string


datatype value = VInt of int
               | VVec of int list
```

Call-by-value let bindings (2)

```
fun eval (EInt i) = VInt i
  | eval (EVec v) = VVec v
  | eval (EAdd (e,f)) = applyAdd (eval e) (eval f)
  | eval (EMul (e,f)) = applyMul (eval e) (eval f)
  | eval (ELet (id,e,f)) = evalLet id (eval e) f
  | eval (EIdent id) = raise EvalError “eval/EIdent”

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

Call-by-value let bindings (2)

```
fun eval (EVal v) = v

  | eval (EAdd (e,f)) = applyAdd (eval e) (eval f)
  | eval (EMul (e,f)) = applyMul (eval e) (eval f)
  | eval (ELet (id,e,f)) = evalLet id (eval e) f
  | eval (EIdent id) = raise EvalError “eval/EIdent”

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

Call-by-value let bindings (2)

```
fun eval (EVal v) = v

  | eval (EAdd (e,f)) = applyAdd (eval e) (eval f)
  | eval (EMul (e,f)) = applyMul (eval e) (eval f)
  | eval (ELet (id,e,f)) = evalLet id (eval e) f
  | eval (EIdent id) = raise EvalError “eval/EIdent”

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

Call-by-value let bindings (2)

```
fun eval (EVal v) = v

  | eval (EA      f)) = applyAdd (eval e) (eval f)
  | ev
  |
  |

and e
```

DON'T FORGET TO ALSO UPDATE THE SUBSTITUTION FUNCTION!

Ever helpful, the type system will "remind" you

Call-by-value let bindings (2)

```
fun eval (EVal v) = v

  | eval (EAdd (e,f)) = applyAdd (eval e) (eval f)
  | eval (EMu
  | eval (ELe
  | eval (EId

and evalLet id
```

Expressions evaluate to the same values as in the original interpreter with expression substitution.

But this may do less work.

Functions

A function is composed of:

- a name
- zero or more parameters
- a body (that can refer to the parameters)

```
fun succ n = n + 1
```

Issues:

- how do you define a function?
- how do you use a function?

Function Environments

- Let’s skip over defining functions (for now)
- We evaluate expressions in the context of existing defined functions
- We maintain the defined functions in a function environment
  - Defining a function = adding it to the environment

Function Environments

- A function environment is a map from names to function definitions
  - environment represented as a list of type `(string * function) list`

```
datatype function =
     FDef of (string * expr)
```

Example:

```
(“succ”,
  FDef (“n”,
        EAdd (EIdent “n”,
              EVal (VInt 1)))
```

Function Environments

- A function environment is a map from names to functio
    - environment repre
      of type `(string`

Single-argument functions for now

```
datatype function =
     FDef of (string * expr)
```

Example:

```
(“succ”,
     FDef (“n”,
           EAdd (EIdent “n”,
                 EVal (VInt 1)))
```

Adding functions

```
datatype expr = EVal of value
              | EAdd of expr * expr
              | EMul of expr * expr

              | ELet of string * expr * expr
              | EIdent of string


datatype value = VInt of int
               | VVec of int list
```

Adding functions

```
datatype expr = EVal of value
              | EAdd of expr * expr
              | EMul of expr * expr
              | EEq of expr * expr
              | ELet of string * expr * expr
              | EIdent of string
              | ECall of string * expr

datatype value = VInt of int
               | VVec of int list
```

Addin

Update the substitution function with these new cases

```
datatype expr = EVal o
              | EAdd of expr * expr
              | EMul of expr * expr
              | EEq of expr * expr
              | ELet of string * expr * expr
              | EIdent of string
              | ECall of string * expr

datatype value = VInt of int
               | VVec of int list
```

Adding functions

```
fun eval (EVal v) = v
  | eval (EAdd (e,f)) = applyAdd (eval e) (eval f)
  | eval (EMul (e,f)) = applyMul (eval e) (eval f)

  | eval (ELet (id,e,f)) = evalLet id (eval e) f
  | eval (EIdent id) = raise EvalError “eval/EIdent”


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

Adding functions

```
fun eval (EVal v) = v
  | eval (EAdd (e,f)) = applyAdd (eval e) (eval f)
  | eval (EMul (e,f)) = applyMul (eval e) (eval f)
  | eval (EEq (e,f)) = applyEq (eval e) (eval f)
  | eval (ELet (id,e,f)) = evalLet id (eval e) f
  | eval (EIdent id) = raise EvalError “eval/EIdent”


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

Adding functions

```
fun eval (EVal v) = v
  | eval (EAdd (e,f)) = applyAdd (eval e) (eval f)
  | eval (EMul (e,f)) = applyMul (eval e) (eval f)
  | eval (EEq (e,f)) = applyEq (eval e) (eval f)
  | eval (ELet (id,e,f)) = evalLet id (eval e) f
  | eval (EIdent id) = raise EvalError “eval/EIdent”

an
```

```
fun applyEq (VInt i) (VInt j) = VBool (i = j)
  | applyEq (VBool b) (VBool c) = VBool (b = c)
  | applyEq _ _ = raise TypeError "applyEq"
```

Adding functions

```
fun eval (EVal v) = v
  | eval (EAdd (e,f)) = applyAdd (eval e) (eval f)
  | eval (EMul (e,f)) = applyMul (eval e) (eval f)
  | eval (EEq (e,f)) = applyEq (eval e) (eval f)
  | eval (ELet (id,e,f)) = evalLet id (eval e) f
  | eval (EIdent id) = raise EvalError “eval/EIdent”
  | eval (ECall (n,e)) = evalCall (lookup n fe) (eval e)

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

and evalCall (FDef (p,body)) v = eval (subst body p (EVal v))
```

```
fun lookup name [] = raise EvalError "lookup"
  | lookup name ((n,f)::fenv) =
      if (n = name)
        then f
      else lookup name fenv
```

```
                                                    f)
                                                    f)
                                                  f)
  | eval (ELet (id,e,f)) = evalL          eval e) f
  | eval (EIdent id) = raise EvalErro    “eval/EIdent”
  | eval (ECall (n,e)) = evalCall (lookup n fe) (eval e)

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

and evalCall (FDef (p,body)) v = eval (subst body p (EVal v))
```

Adding functions

```
fun eval fe (EVal v) = v
  | eval fe (EAdd (e,f)) = applyAdd (eval fe e) (eval fe f)
  | eval fe (EMul (e,f)) = applyMul (eval fe e) (eval fe f)
  | eval fe (EEq (e,f)) = applyEq (eval fe e) (eval fe f)
  | eval fe (ELet (id,e,f)) = evalLet id (eval fe e) f
  | eval fe (EIdent id) = raise EvalError “eval/EIdent”
  | eval fe (ECall (n,e)) = evalCall fe (lookup n fe) (eval fe e)

and evalLet fe id v body = eval fe (subst body id (EVal v))

and evalCall fe (FDef (p,body)) v = eval fe (subst body p (EVal v))
```

Examples

```
(“succ”,
 FDef (“n”, EAdd (EIdent “n”, EVal (VInt 1))))

(“pred”,
 FDef (“n”, ESub (EIdent “n”, EVal (VInt 1))))
```

Examples

```
(“succ”,
 FDef (“n”, EAdd (EIdent “n”, EVal (VInt 1))))

(“pred”,
 FDef (“n”, ESub (EIdent “n”, EVal (VInt 1))))

(“fact”,
 FDef (“n”,
       EIf (EEq (EIdent “n”, EVal (VInt 0)),
            EVal (VInt 1),
            EMul (EIdent “n”,
                  ECall (“fact”,
                         ECall (“pred”,
                                EIdent “n”))))))
```

First effect: nontermination!

```
(“loop”,
 FDef (“x”, ECall (“loop”, EIdent “x”)))
```

This lets us differentiate substituting values/expressions in `let`

```
    let x = loop 0
      in  if true then 1 else x
```

First effect: nontermination!

```
(“loop”,
 FDef (“x”, ECall (“loop”, EIdent “x”)))
```

This lets us differentiate substituting values/expressions in `let`

```
    let x = loop 0
      in  if true then 1 else x
```

First effect: nontermination!

```
(“loop”,
 FDef (“x”, ECall (“loop”, EIdent “x”)))
```

This lets us differentiate substituting values/expressions in `let`

substitute expression

```
    let x = loop 0
      in  if true then 1 else loop 0
```

First effect: nontermination!

```
(“loop”,
 FDef (“x”, ECall (“loop”, EIdent “x”)))
```

This lets us differentiate substituting values/expressions in `let`

```
            if true then 1 else loop 0
```

First effect: nontermination!

```
(“loop”,
 FDef (“x”, ECall (“loop”, EIdent “x”)))
```

This lets us differentiate substituting values/expressions in `let`

First effect: nontermination!

```
(“loop”,
 FDef (“x”, ECall (“loop”, EIdent “x”)))
```

This lets us differentiate substituting values/expressions in `let`

```
    let x = loop 0
      in  if true then 1 else x
```

First effect: nontermination!

```
(“loop”,
 FDef (“x”, ECall (“loop”, EIdent “x”)))
```

This lets us different

values/expressions i

Evaluation will get stuck here trying to evaluate loop 0

```
    let x = loop 0
      in  if true then 1 else x
```

Multi-argument functions

Many approaches to handle functions with multiple arguments:

- bake them in
- fake multiple arguments with tuples
- currying (later)

```
datatype function =
     FDef of (string * expr)
```

Multi-argument functions

Many approaches to handle functions with multiple arguments:

- bake them in
- fake multiple arguments with tuples
- currying   (later)

```
datatype function =
     FDef of (string list * expr)
```

Adding functions (2)

```
datatype expr = EVal of value
              | EAdd of expr * expr
              | EMul of expr * expr
              | EEq of expr * expr
              | EIdent of string
              | ECall of string * expr

datatype value = VInt of int
               | VVec of int list
```

Adding functions (2)

```
datatype expr = EVal of value
              | EAdd of expr * expr
              | EMul of expr * expr
              | EEq of expr * expr
              | EIdent of string
              | ECall of string * expr list

datatype value = VInt of int
               | VVec of int list
```

Adding functions (2)

```
fun eval fe (EVal v) = v
  | eval fe (EAdd (e,f)) = applyAdd (eval fe e) (eval fe f)
  | eval fe (EMul (e,f)) = applyMul (eval fe e) (eval fe f)
  | eval fe (EEq (e,f)) = applyEq (eval fe e) (eval fe f)
  | eval fe (EIdent id) = raise EvalError “eval/EIdent”
  | eval fe (ECall (n,e)) =
                evalCall fe (lookup n fe) (eval fe e)

and evalCall fe (FDef (p,body)) v = eval fe (subst body p (EVal v))
```

Adding functions (2)

```
fun eval fe (EVal v) = v
  | eval fe (EAdd (e,f)) = applyAdd (eval fe e) (eval fe f)
  | eval fe (EMul (e,f)) = applyMul (eval fe e) (eval fe f)
  | eval fe (EEq (e,f)) = applyEq (eval fe e) (eval fe f)
  | eval fe (EIdent id) = raise EvalError “eval/EIdent”
  | eval fe (ECall (n,es)) =
                evalCall fe (lookup n fe) (evalList fe es)

and evalCall fe (FDef (ps,body)) vs = eval fe (substAll body ps vs)

and evalList fe [] = []
  | evalList fe (e::es) = (eval fe e)::(evalList fe es)
```

Adding functions (2)

```
fun eval fe (EVal v) = v
  |                                                         fe f)
  |                                                         fe f)
  |                                                          f)
  |                                                         ,
  |
               evalCall fe (lookup                     lList fe es)
```

```
fun substAll e [] [] = e
  | substAll e (id::ids) (v::vs) =
           substAll (subst e id (EVal v)) ids vs
  | substAll _ _ _ = raise EvalError "substAll"
```

```
and evalCall fe (FDef (ps,body)) vs = eval fe (substAll body ps vs)

and evalList fe [] = []
  | evalList fe (e::es) = (eval fe e)::(evalList fe es)
```

Examples

```
(“pred”,
 FDef ([“n”], ESub (EIdent “n”, EVal (VInt 1))))

(“exp”,
 FDef ([“a”,”n”],
       EIf (EEq (EIdent "n", EVal (VInt 0)),
            EVal (VInt 1),
            EMul (EIdent "a",
                  ECall ("exp",
                         [EIdent "a",
                          ECall ("pred",
                                 [EIdent "n"])])))))
```

Primitive operations as functions

- Primitive operations are just like defined functions
  - Except code for operations is built into the interpreter

- Can uniformize treatment

```
datatype function =
     FDef of string list * expr
```

Primitive operations as functions

- Primitive operations are just like defined functions
  - Except code for operations is built into the interpreter

- Can uniformize treatment

```
datatype function =
     FDef of string list * expr
   | FPrim of value list -> value
```

Primitive operations as functions

- Primitive opera
  functions
  - Except code for
    interpreter

- Can uniformize

We use the fact that functions in SML are first-class: they can be put in data structures and we can pass them as arguments to other functions

```
datatype function =
    FDef of string list * expr
  | FPrim of value list -> value
```

Adding functions (3)

```
datatype expr = EVal of value
              | EAdd of expr * expr
              | EMul of expr * expr
              | EEq of expr * expr
              | EIdent of string
              | ECall of string * expr list


datatype value = VInt of int
               | VVec of int list
```

Adding functions (3)

```
datatype expr = EVal of value
              ~~| EAdd of expr * expr~~
              ~~| EMul of expr * expr~~
              ~~| EEq of expr * expr~~
              | EIdent of string
              | ECall of string * expr list


datatype value = VInt of int
               | VVec of int list
```

Adding functions (3)

```
fun eval fe (EVal v) = v
  | eval fe (EAdd (e,f)) = applyAdd (eval fe e) (eval fe f)
  | eval fe (EMul (e,f)) = applyMul (eval fe e) (eval fe f)
  | eval fe (EEq (e,f)) = applyEq (eval fe e) (eval fe f)
  | eval fe (EIdent id) = raise EvalError “eval/EIdent”
  | eval fe (ECall (n,es)) =
                evalCall fe (lookup n fe) (evalList fe es)

and evalCall fe (FDef (ps,body)) vs = eval fe (substAll body ps vs)


and evalList fe [] = []
  | evalList fe (e::es) = (eval fe e)::(evalList fe es)
```

Adding functions (3)

```
fun eval fe (EVal v) = v
  ~~| eval fe (EAdd (e,f)) = applyAdd (eval fe e) (eval fe f)~~
  ~~| eval fe (EMul (e,f)) = applyMul (eval fe e) (eval fe f)~~
  ~~| eval fe (EEq (e,f)) = applyEq (eval fe e) (eval fe f)~~
  | eval fe (EIdent id) = raise EvalError “eval/EIdent”
  | eval fe (ECall (n,es)) =
                evalCall fe (lookup n fe) (evalList fe es)

and evalCall fe (FDef (ps,body)) vs = eval fe (substAll body ps vs)
  | evalCall fe (FPrim f) vs = f vs

and evalList fe [] = []
  | evalList fe (e::es) = (eval fe e)::(evalList fe es)
```

Slight change to interface

```
fun applyAdd (VInt i) (VInt j) = VInt (i + j)
  | applyAdd (VVec v) (VVec w) =
       if length v = length w then VVec (addVec v w)
       else raise TypeError "applyAdd”
  | applyAdd _ _ = raise TypeError "applyAdd”

fun applyMul (VInt i) (VInt j) = VInt (i * j)
  | applyMul (VInt i) (VVec v) = VVec (scaleVec i v)
  | applyMul (VVec v) (VInt i) = VVec (scaleVec i v)
  | applyMul (VVec v) (VVec w) =
      if length v = length w then VInt (inner v w)
      else raise TypeError "applyMul"
  | applyMul _ _ = raise TypeError "applyMul"
```

Slight change to interface

```
fun applyAdd [VInt i, VInt j] = VInt (i + j)
  | applyAdd [VVec v, VVec w] =
       if length v = length w then VVec (addVec v w)
       else raise TypeError "applyAdd”
  | applyAdd _ = raise TypeError "applyAdd”

fun applyMul [VInt i, VInt j] = VInt (i * j)
  | applyMul [VInt i, VVec v] = VVec (scaleVec i v)
  | applyMul [VVec v, VInt i] = VVec (scaleVec i v)
  | applyMul [VVec v, VVec w] =
      if length v = length w then VInt (inner v w)
      else raise TypeError "applyMul"
  | applyMul _ = raise TypeError "applyMul"
```

Initial function environment

Define the built-in primitive operations in an initial function environment:

For our little language:

```
[“add”, FPrim applyAdd,
 “sub”, FPrim applySub,
 “mul”, FPrim applyMul,
 “neg”, FPrim applyNeg,
 “eq”, FPRim applyEq]
```

Initial function environment

Define th
initial fun

To add new primitive operations:

- no change to internal representation
- only change the initial environment

For our li

```
[“add”, FPrim applyAdd,
 “sub”, FPrim applySub,
 “mul”, FPrim applyMul,
 “neg”, FPrim applyNeg,
 “eq”, FPRim applyEq]
```

Examples

```
(“pred”,
 FDef ([“n”],
       ECall (“sub”, [EIdent “n”,
                      EVal (VInt 1)])))

(“fact”,
 FDef ([“n”],
       EIf (ECall (“eq”, [EIdent “n”, EVal (VInt 0)]),
            EVal (VInt 1),
            ECall (“mul”,
                   [EIdent “n”,
                    ECall (“fact”,
                           [ECall (“pred”,
                                   [EIdent “n”])])]))))
```

To think about

- What we did today lets us call functions

- What about defining functions?
  - What’s special about that?

- What about passing functions as arguments to other functions?

- We’ll return to this in two weeks
  - next week: surface syntax

Second homework

- Implement lists
  - LISP-style: head, tail, isempty, cons

- Implement simultaneous bindings

  -
    ```
    slet x = y
         y = x
      in [x,y]
    ```

- ~~Implement call-by-name~~
  - ~~interaction with lists~~

Something else...