Introduction to Bindings and Identifiers

January 28, 2014

Riccardo Pucella

Last time

- Core of an interpreter for a simple mathematical call-by-value language
- Call-by-value is an evaluation strategy
- It answers the question of when to reduce expressions during evaluation
  - Call-by-value: before you invoke a primitive operation (or a defined function)
  - Call-by-name, call-by-need (homework)

Call-by-value interpreter

```
datatype expr = EInt of int
              | EVec of int list
              | EAdd of expr * expr
              | ESub of expr * expr
              | EMul of expr * expr
              | ENeg of expr
```

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

Call-by-value interpreter

```
datatype expr = EInt of int
              | EVec of int list
              | EAdd of expr * expr
              | ESub of
              | EMul of
              | ENeg of
```

Values:

```
fun eval (EInt i) = VInt i
  | eval (EVec v) = VVec v
  ...
```

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

Call-by-value interpreter

```
datatype expr = EInt of int
              | EVec of int list
              | EAdd of expr * expr
              | ESub of expr * expr
              | EMul of expr * expr
              | ENeg of expr
```

Primitive operations (functions):

```
  | eval (EAdd (e,f)) = applyAdd (eval e) (eval f)
  | eval (ESub (e,f)) = applySub (eval e) (eval f)
  ...
```

Call-by-value interpreter

```
datatype expr = EInt of int
              | EVec of int list
              | EAdd of expr * expr
              | ESub of expr * expr
              | EMul of expr * expr
              | ENeg of expr
```

Primitive operations (functions):

call-by-value

```
  | eval (EAdd (e,f)) = applyAdd (eval e) (eval f)
  | eval (ESub (e,f)) = applySub (eval e) (eval f)
  ...
```

Booleans and conditionals

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

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

Booleans and conditionals

```
datatype expr = EInt of int
              | EVec of int list
              | EAdd of expr * expr
              | EMul of expr * expr
              | EBool of bool
              | EAnd of expr * expr
              | EIf of expr * expr * expr

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

Booleans and conditionals

da

```
if (true and false)
  then 1 + 2
else 1 + 3
```

```
EIf (EAnd (EBool true, EBool false),
     EAdd (EInt 1, EInt 2),
     EAdd (EInt 1, EInt 3))
```

da

Booleans and conditionals

da

```
1 + (if (true and false)
       then 2
     else 3)
```

```
EAdd (EInt 1,
      EIf (EAnd (EBool true, EBool false),
           EInt 2,
           EInt 3))
```

da

Evaluation function

```
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)
```

Evaluation function

```
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 (EBool b) = VBool b
  | eval (EAnd (e,f)) = applyAnd (eval e) (eval f)
  | eval (EIf (e,f,g)) = evalIf (eval e) f g
```

Evaluation function

```
fun eval (EInt i) = VInt i
  | e
  | e
  | e
  | ev
  | eval (EAnd (e,f)) = applyAnd (eval e) (eval f)
  | eval (EIf (e,f,g)) = evalIf (eval e) f g
```

```
fun applyAnd (VBool b) (VBool c) = VBool (b andalso c)
  | applyAnd _ _ = raise TypeError “applyAnd”
```

Evaluation function

```
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 (EBool b) = VBool b
  | eval (EAnd (e,f)) = applyAnd (eval e) (eval f)
  | eval (EIf (e,f,g)) = evalIf (eval e) f g

and evalIf (VBool true) f g = eval f
  | evalIf (VBool false) f g = eval g
  | evalIf _ _ _ = raise TypeError "evalIf"
```

Evaluation function

```
fun eval (EInt i) = VInt i
  | eval (EVec v) = VVec v
  | eval (EAdd (e,
  | eval (EMul (e,
  | eval (EBool b)
  | eval (EAnd (e,f)) = applyAnd (eval e) (eval f)
  | eval (EIf (e,f,g)) = evalIf (eval e) f g

and evalIf (VBool true) f g = eval f
  | evalIf (VBool false) f g = eval g
  | evalIf _ _ _ = raise TypeError "evalIf"
```

`EIf` is a special form

(doesn't follow evaluation strategy)

Evaluation function

```
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 (EBool b) = VBool b
  | eval (EAnd (e,f)) = applyAnd (eval e) (eval f)
  | eval (EIf (e,f,g)) =
      (case (eval e) of
          VBool true => eval f
        | VBool false => eval g
        | _ => raise TypeError "eval/EIf")
```

More special forms

Special forms arise whenever you want to bypass the call-by-value evaluation mechanism.

Consider the Boolean operation and:

```
true and true = ?
```

More special forms

Special forms arise whenever you want to bypass the call-by-value evaluation mechanism.

Consider the Boolean operation and:

```
true and false = ?
```

More special forms

Special forms arise whenever you want to bypass the call-by-value evaluation mechanism.

Consider the Boolean operation and:

```
false and true = ?
```

More special forms

Special forms arise whenever you want to bypass the call-by-value evaluation mechanism.

Consider the Boolean operation and:

```
false and whatever = ?
```

More special forms

Special forms arise whenever you want to bypass the call-by-value evaluation mechanism.

Consider the Boolean operation and:

`false and` *whatever* `= false`

No need to evaluate *whatever*

More special forms

Special forms arise whenever you want to bypass the call-by-value evaluation mechanism.

Consider the Boolean operation and:

`true and` *`whatever`* `= ?`

Short-circuiting Boolean operations

```
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 (EBool b) = VBool b
  | eval (EAnd (e,f)) = applyAnd (eval e) (eval f)
  | eval (EIf (e,f,g)) = evalIf (eval e) f g
```

Short-circuiting Boolean operations

```
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 (EBool b) = VBool b
  | eval (EAnd (e,f)) = evalAnd (eval e) f
  | eval (EIf (e,f,g)) = evalIf (eval e) f g

and evalAnd (VBool true) f = eval f
  | evalAnd (VBool false) f = VBool false
  | evalAnd _ _ = raise TypeError “evalAnd”
```

Functions versus special forms

- Some things just have to be special forms in a call-by-value language
  - conditionals, while loops, etc
  - in general: control flow is a special form

- Other things are up to the designer
  - and, or
  - * ?

- BUT: the less uniform the evaluation model, the more difficult the language is to use
  - hard to predict what will be evaluated when

Let bindings

Introduce a way to give a local name to an expression

```
let x = 10 + 10
  in x * x
```

What do we need in our internal representation?

Let bindings

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

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

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
```

Evaluation function

```
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)
```

Evaluation function

```
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) = ??

and evalLet id exp body = ??
```

The substitution model

A `let` gives a local name to an expression

```
let x = 10 + 10
  in  x * x
```

The substitution model

A `let` gives a local name to an expression

```
let x = 10 + 10
  in  (10 + 10) * (10 + 10)
```

substitute x with 10 + 10...

The substitution model

A `let` gives a local name to an expression

```
(10 + 10) * (10 + 10)
```

substitute x with 10 + 10...
and get rid of the `let`

Nested bindings

```
let x = 10
 let y = 20
  in x * y
```

Nested bindings

```
let y = 20
 in 10 * y
```

Nested bindings

10 * 20

Nested bindings

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

Nested bindings

```
let y = 10
 in 10 * y
```

Nested bindings

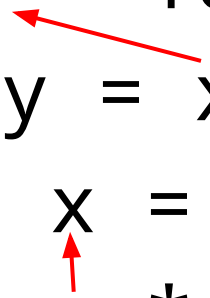
10 * 10

Nested bindings

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

Nested bindings

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



An identifier always refers to the nearest enclosing definition

Nested bindings

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

Substituting for x is "blocked" by a `let` for x

Nested bindings

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

Nested bindings

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

Nested bindings

```
30 * 10
```

Substitution function

```
fun subst (EInt i) id e =
  | subst (EVec v) id e =
  | subst (EAdd (f,g)) id e =

  | subst (EMul (f,g)) id e =

  | subst (ELet (id’,f,g)) id e =



  | subst (EIdent id’) id e =
```

Substitution function

```
fun subst (EInt i) id e = EInt i
  | subst (EVec v) id e =
  | subst (EAdd (f,g)) id e =

  | subst (EMul (f,g)) id e =

  | subst (ELet (id',f,g)) id e =



  | subst (EIdent id') id e =
```

Substitution function

```
fun subst (EInt i) id e = EInt i
  | subst (EVec v) id e = EVec v
  | subst (EAdd (f,g)) id e =

  | subst (EMul (f,g)) id e =

  | subst (ELet (id',f,g)) id e =



  | subst (EIdent id') id e =
```

Substitution function

```
fun subst (EInt i) id e = EInt i
  | subst (EVec v) id e = EVec v
  | subst (EAdd (f,g)) id e =
      EAdd (subst f id e, subst g id e)
  | subst (EMul (f,g)) id e =

  | subst (ELet (id',f,g)) id e =



  | subst (EIdent id') id e =
```

Substitution function

```
fun subst (EInt i) id e = EInt i
  | subst (EVec v) id e = EVec v
  | 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 =



  | subst (EIdent id') id e =
```

Substitution function

```
fun subst (EInt i) id e = EInt i
  | subst (EVec v) id e = EVec v
  | 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 =
```

Substitution function

```
fun subst (EInt i) id e = EInt i
  | subst (EVec v) id e = EVec v
  | 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'
```

Evaluation function

```
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) = ??

and evalLet id exp body = ??
```

Evaluation function

```
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) = ??

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

Evaluation function

```
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)
```

A slight discrepancy

The substitution model is not call-by-value

In call-by-values languages, like SML:

```
let x = 10 + 10
   in  x * x
```

A slight discrepancy

The substitution model is not call-by-value

In call-by-values languages, like SML:

```
let x = 20
   in  x * x
```

A slight discrepancy

The substitution model is not call-by-value

In call-by-values languages, like SML:

```
let x = 20
   in  20 * 20
```

A slight discrepancy

The substitution model is not call-by-value

In call-by-values languages, like SML:

```
20 * 20
```

A slight discrepancy

The substitution model is not call-by-value

In call-by-values languages, like SML:

```
400
```

A slight discrepancy

The substitution model is not call-by-value

In call-by-values languages, like SML:

Your task for next time

modify the interpreter to implement a call-by-value substitution

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