Values and Identifiers

January 30, 2020

Riccardo Pucella

Last time

- Simple expression language
- Abstract representation as a tree of expressions
- Evaluation: reduce an expression to a value

Today:

- Add different kinds of values
- Add local bindings and identifiers

Adding values

Our Language

Really an abstract representation of one:

```
exp :== EInteger (int)
        EPlus (exp,exp)
        ETimes (exp,exp)
        EIf (exp,exp,exp)
```

Adding values

The expression language we have only handles one type of data, integers

- method `eval () : Int`

It would be nice to support other kinds of values:

- Booleans
- Fractions / floats
- Vectors / matrices
- ...

The approach

- we introduce a class of values
- we change the definition of eval() to return values instead of integers
- we add expression nodes for literal of all values, not just integer literals
  - EInteger, EBoolean, EFraction, …
  - Sometimes simpler to use a single ELiteral

A class of values

```
abstract class Value {

   // methods to check the type of a value
   // avoids instanceOf
   def isInteger () : Boolean
   def isBoolean () : Boolean

   // extract the underlying value in a Value
   def getInt () : Int
   def getBool () : Boolean
}
```

Integers

```
class VInteger (val i:Int) extends Value {

   override def toString () : String = i.toString()

   def isInteger () : Boolean = true
   def isBoolean () : Boolean = false

   def getInt () : Int = i

   def getBool () : Boolean =
      throw new Exception("Not a Boolean")

}
```

Booleans

```
class VBoolean (val b:Boolean) extends Value {

   override def toString () : String = b.toString()

   def isInteger () : Boolean = false
   def isBoolean () : Boolean = true

   def getInt () : Int =
      throw new Exception("Not an Integer")

   def getBool () : Boolean = b
}
```

Changing eval ()

```
abstract class Exp {

    def eval (): Int

}
```

Changing eval ()

```
abstract class Exp {

    def eval (): Value

}
```

Modifying EInteger

```
class EInteger (val i:Int) extends Exp {

    override def toString () : String =
       "EInteger(" + i + ")"

    def eval () : Int =
       i
}
```

Modifying EInteger

```
class EInteger (val i:Int) extends Exp {

    override def toString () : String =
       "EInteger(" + i + ")"

    def eval () : Value =
       new VInteger(i)
}
```

Adding EBoolean

```
class EBoolean (val b:Boolean) extends Exp {

    override def toString () : String =
       "EBoolean(" + b + ")"

    def eval () : Value =
       new VBoolean(b)
}
```

Modifying EPlus

```
class EPlus (val e1:Exp, val e2:Exp) extends Exp {
    ...

    def eval () : Int = {
       val i1 = e1.eval()
       val i2 = e2.eval()
       return i1 + i2
    }

}
```

Modifying EPlus

```
class EPlus (val e1:Exp, val e2:Exp) extends Exp {
    ...

    def eval () : Value = {
       val v1 = e1.eval()
       val v2 = e2.eval()
       if (v1.isInteger() && v2.isInteger()) {
          return new VInteger(v1.getInt() + v2.getInt())
       }
       throw new Exception("Type error in EPlus")
    }
}
```

Modifying EIf

```
class EIf (val c:Exp, val t:Exp, val e:Exp) extends Exp {
    ...

    def eval () : Int = {
       val ci = c.eval()
       if (ci == 0) {
          return e.eval()
       } else {
          return t.eval()
       }
    }
}
```

Modifying EIf

```
class EIf (val c:Exp, val t:Exp, val e:Exp) extends Exp {
    ...

    def eval () : Value = {
       val cv = c.eval()
       if (cv.isBoolean()) {
          if (cv.getBool()) {
             return t.eval()
          } else {
             return e.eval()
          }
       }
       throw new Exception("Type error in EIf")
    }
}
```

Adding EAnd

```
class EAnd (val e1:Exp, val e2:Exp) extends Exp {

   override def toString () : String =
       "EAnd(" + e1 + "," + e2 + ")"

    def eval () : Value = {
       val v1 = e1.eval()
       val v2 = e2.eval()
       if (v1.isBoolean() && v2.isBoolean()) {
         return new VBoolean(v1.getBool() && v2.getBool())
       }
       throw new Exception("Type error in EAnd")
    }
}
```

Adding fractions

```
abstract class Value {

   // methods to check the type of a value
   // avoids instanceOf
   def isInteger () : Boolean
   def isBoolean () : Boolean


   // extract the underlying value in a Value
   def getInt () : Int
   def getBool () : Boolean


}
```

Adding fractions

```
abstract class Value {

   // methods to check the type of a value
   // avoids instanceOf
   def isInteger () : Boolean
   def isBoolean () : Boolean
   def isFraction () : Boolean

   // extract the underlying value in a Value
   def getInt () : Int
   def getBool () : Boolean
   def getNum () : Int
   def getDen () : Int
}
```

Adding fractions

```
class VFraction (val num:Int, val den:Int) extends Value {

   override def toString () : String = num + "/" + den

   def isInteger () : Boolean = false
   def isBoolean () : Boolean = false
   def isFraction () : Boolean = true

   def getInt () : Int =
      throw new Exception("Not a fraction")
   def getBool () : Boolean =
      throw new Exception("Not a fraction")
   def getNum () : Int = num
   def getDen () : Int = den
}
```

Adding fractions

```
class EFraction (val num:Int, val den:Int) extends Exp {

   override def toString () : String =
       "EFraction(" + num + ", " + den + ")"

    def eval () : Value =
       new VFraction(n, d)
}
```

Modifying EPlus for fractions

```
class EPlus (val e1:Exp, val e2:Exp) extends Exp {
      ...
    def eval () : Value = {
       val v1 = e1.eval()
       val v2 = e2.eval()
       if (v1.isInteger() && v2.isInteger()) {
          return new VInteger(v1.getInt() + v2.getInt())
       }
       throw new Exception("Type error in EPlus")
    }
}
```

Modifying EPlus for fractions

```
class EPlus (val e1:Exp, val e2:Exp) extends Exp {
      ...
    def eval () : Value = {
       val v1 = e1.eval()
       val v2 = e2.eval()
       if (v1.isInteger() && v2.isInteger()) {
          return new VInteger(v1.getInt() + v2.getInt())
       } else if (v1.isFraction() && v2.isFraction()) {
          return new VFraction(v1.getNum()*v2.getDen()
                                + v2.getNum()*v1.getDen(),
                               v1.getDen()*v2.getDen())
       }
       throw new Exception("Type error in EPlus")
    }
}
```

Adding local bindings

Our language

Really an abstract representation of one:

```
exp :== EInteger (int)
        EBoolean (bool)
        EPlus (exp,exp)
        ETimes (exp,exp)
        EAnd (exp,exp)
        EIf (exp,exp,exp)
```

Local bindings

Introduce a way to give a local name to an expression, e.g.,

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

What do we need in our abstract representation?

Local bindings

Introduce
expressio

```
let
  x
```

- A representation for a let expression
  let (*id* = *exp*) *exp*
- A representation for an identifier use
  id

What do we need in our abstract representation?

New expression nodes

```
class ELet (val id:String, val e:Exp, val body:Exp)
                                                extends Exp {
    ...

    def eval () : Value = ???
}


class EId (val id:String) extends Exp {
    ...

    def eval () : Value = ???
}
```

Evaluating bindings and identifiers

There are two basic models:

(1) Substitution model

(2) Environment model

We implement the environment model

Environment model

Evaluate all expressions in the context of an environment (mapping identifiers to values)

To evaluate *let (x=e) body*

- evaluate *e* to a value *v*
- add $x \leftarrow v$ to the environment
- evaluate *body* in that new environment

To evaluate *x*

- lookup value of *x* in the environment

Example

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

Example

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

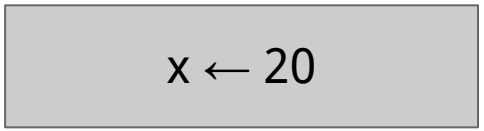
Example

x * x

x ← 20

Example

400



Nested example (1)

```
let (x = 10)
 let (y = 20)
  x * y
```

Nested example (1)

```
let (y = 20)
 x * y
```

x ← 10

Nested example (1)

x * y

y ← 20
x ← 10

Nested example (1)

200

y ← 20
x ← 10

Nested example (2)

```
let (x = 10)
 let (y = x)
  x * y
```

Nested example (2)

```
let (y = x)
 x * y
```

x ← 10

Nested example (2)

x * y

y ← 10
x ← 10

Nested example (2)

100

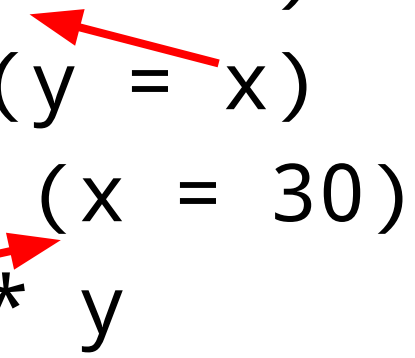
y ← 10
x ← 10

Nested example (3)

```
let (x = 10)
 let (y = x)
  let (x = 30)
   x * y
```

Nested example (3)

```
let (x = 10)
 let (y = x)
  let (x = 30)
   x * y
```



An identifier always refers to the nearest enclosing definition

Nested example (3)

```
let (x = 10)
 let (y = x)
  let (x = 30)
   x * y
```

Nested example (3)

```
let (y = x)
 let (x = 30)
  x * y
```

x ← 10

Nested example (3)

```
let (x = 30)
 x * y
```

y ← 10
x ← 10

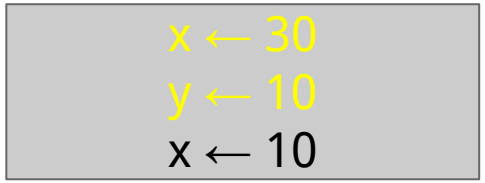
Nested example (3)

x * y

x ← 30
y ← 10
x ← 10

Nested example (3)

300



Environments

```
class Env (val content: List[(String, Value)]) {

      def push (id : String, v : Value) : Env =
          new Env((id,v)::content)

      def lookup (id : String) : Value = {
          for (entry <- content) {
              if (entry._1 == id) {
                 return entry._2
              }
          }
          throw new Exception("Unbound identifier "+id)
      }
}
```

Environments

```
class Env (val content: List[(String, Value)]) {
                    if (entry._1 == id) {
                        return entry._2
                    }
                }
                throw new Exception("Unbound identifier "+id)
            }
}
```

An environment is a list of bindings each represented as a pair

(*identifier* , *value*)

Environments

```
class Env (val content: List[(String, Value)]) {

      def push (id : String, v : Value) : Env =
          new Env((id,v)::content)

                                                    id)
}
```

Pushing an new binding (*identifier*,*value*) returns a new environment

Environments are immutable

To lookup an identifier:

loop through the list and return the first value found bound to that identifier

```
clas
        ne                    V)::content)

    def lookup (id : String) : Value = {
        for (entry <- content) {
            if (entry._1 == id) {
               return entry._2
            }
        }
        throw new Exception("Unbound identifier "+id)
    }
}
```

Modifying eval()

```
abstract class Exp {

    def eval (): Value

}
```

Modifying eval()

```
abstract class Exp {

    def eval (env : Env): Value

}
```

Thread environments through

For example:

```
class EPlus (val e1:Exp, val e2:Exp) extends Exp {
    ...

    def eval () : Value = {
       val v1 = e1.eval()
       val v2 = e2.eval()
       if (v1.isInteger() && v2.isInteger()) {
          return new VInteger(v1.getInt() + v2.getInt())
       }
       throw new Exception("Type error in EPlus")
    }
}
```

Thread environments through

Most evaluation methods do not use the environment

But they need to pass it on to evaluate subexpressions

```
    ...

    def eval (env : Env) : Value = {
       val v1 = e1.eval(env)
       val v2 = e2.eval(env)
       if (v1.isInteger() && v2.isInteger()) {
          return new VInteger(v1.getInt() + v2.getInt())
       }
       throw new Exception("Type error in EPlus")
    }
}
```

Evaluation for ELet

```
class ELet (val id:String, val e:Exp, val body:Exp)
                                                extends Exp {
    ...

    def eval (env : Env) : Value = {
        val v = e.eval(env)
        val new_env = env.push(id, v)
        return body.eval(new_env)
    }
}
```

Evaluation for EId

```
class EId (val id:String) extends Exp {
    ...

    def eval (env : Env) : Value = env.lookup(id)

}
```

Second homework

- More types of values
- `let` with concurrent/sequential bindings