Identifiers and the Substitution Model

September 13, 2016

Riccardo Pucella

Last time

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

Today:

- Simplify the abstract representation
- Add local bindings
- Add defined functions — homework

(1) Simplifying the representation

In the code presented in class and on the homework, a lot of the code looked like:

```
class EPlus (Exp):
    def __init__ (self,e1,e2):
        self._exp1 = e1
        self._exp2 = e2

    def eval (self):
        v1 = self._exp1.eval()
        v2 = self._exp2.eval()
        if v1.type == "integer" and v2.type == "integer":
            return VInteger(v1.value + v2.value)
        raise Exception ("Runtime error: typs")
```

(1) Simplifying the representation

In the code presented in class and on the homework, a lot of the code looked like:

```
class EPlus (Exp):
    def __init__ (self,e1,e2):
        self._exp1 = e1
        self._exp2 = e2

    def eval (self):
        v1 = self._exp1.eval()
        v2 = self._exp2.eval()
        return oper_plus(v1,v2)
```

(1) Simplifying the representation

In the code presented in class and on the homework, a lot of the code looked like:

```
class EMinus (Exp):
    def __init__ (self,e1,e2):
        self._exp1 = e1
        self._exp2 = e2

    def eval (self):
        v1 = self._exp1.eval()
        v2 = self._exp2.eval()
        return oper_minus(v1,v2)
```

(1) Simplifying the representation

In the code presented in class and on the homework, a lot of the code looked like:

```
class ENot (Exp):
    def __init__ (self,e1):
        self._exp1 = e1

    def eval (self):
        v1 = self._exp1.eval()
        return oper_not(v1)
```

(1) Simplifying the representation

In the code presented in class and on the homework, a lot of the code looked like:

```
class EAnd (Exp):                    # NOT short-circuiting
    def __init__ (self,e1,e2):
        self._exp1 = e1
        self._exp2 = e2

    def eval (self):
        v1 = self._exp1.eval()
        v2 = self._exp2.eval()
        return oper_and(v1,v2)
```

Primitive operations

The common structure:

```
class E… (Exp):
    def __init__ (self,e1,…,eN):
        self._exp1 = e1
        …
        self._expN = eN

    def eval (self):
        v1 = self._exp1.eval()
        …
        vN = self._expN.eval()
        return primitive_operation(v1,…,vN)
```

EPrimCall

Let's create a single Expression node for this

```
10 + 20 → EPlus(EInteger(10),EInteger(20))
```

EPrimCall

Let's create a single Expression node for this

```
10 + 20 → EPrimCall("+",[EInteger(10),
                         EInteger(20)])
```

We need a way to map "+" to the underlying primitive function acting on values

- pass a primitives dictionary to `eval()`

EPrimCall

```
class EPrimCall (Exp):

    def __init__ (self,name,es):
        self._name = name
        self._exps = es

    def eval (self, prim_dict):
        vs = [ e.eval(prim_dict) for e in self._exps ]
        return apply(prim_dict[self._name],vs)
```

New interface to eval()

```
class E… (Exp):
    …

    def eval (self, prim_dict):
        …
```

I prefer to pass prim_dict as an argument than having it as a global variable — we'll see why later

Our Expression nodes

Literal (value) expressions:

- EInteger, EBoolean

Calling primitive operations:

- EPrimCall

Special forms (with dedicated eval rules):

- EIf, EAnd, EOr

(2) 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?

New expression nodes

```
class ELet (Exp):
    def __init__ (self,id,e1,e2):
        self._id = id
        self._e1 = e1
        self._e2 = e2

    def eval (self,prim_dict):
        ???

class EId (Exp):
    def __init__ (self,id):
        self._id = id

    def eval (self,prim_dict):
        ???
```

The substitution model

A `let` gives a local name to an expression

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

The substitution model

A `let` gives a local name to an expression

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

Nested bindings

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

Nested bindings

```
10 * 20
```

Nested bindings

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

Nested bindings

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

Nested bindings

10 * 10

Nested bindings

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

Nested bindings

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

An identifier always refers to the nearest enclosing definition

Nested bindings

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

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

Nested bindings

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

Nested bindings

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

Nested bindings

30 * 10

New interface method: substitute()

```
class E… (Exp):
    …

    def substitute (self, id, new_e):
        …
        # should return a new expression
```

Implementing substitution

```
class EInteger (Exp):
    ...

    def substitute (self, id, new_e):
        return self
```

Implementing substitution

```
class EPrimCall (Exp):
    ...

    def substitute (self, id, new_e):
        new_es = [ e.substitute(id,new_e)
                            for e in self._exps]
        return EPrimCall(self._name,new_es)
```

Implementing substitution

```
class EIf (Exp):
    …

    def substitute (self, id, new_e):
        return EIf(self._cond.substitute(id,new_e),
                   self._then.substitute(id,new_e),
                   self._else.substitute(id,new_e))
```

Implementing substitution

```
class EId (Exp):
    …

    def substitute (self, id, new_e):
        if id == self._id:
            return new_e
        return self
```

Implementing substitution

```
class ELet (Exp):
    …

    def substitute (self, id, new_e):
        if id == self._id:
            return ELet(self._id,
                        self._e1.substitute(id,new_e),
                        self._e2)
        return ELet(self._id,
                    self._e1.substitute(id,new_e),
                    self._e2.substitute(id,new_e))
```

Evaluating for ELet

```
class ELet (Exp):
    …

    def eval (self, prim_dict):
        new_e2 = self._e2.substitute(self._id,self._e1)
        return new_e2.eval(prim_dict)
```

Evaluating EId

```
class EId (Exp):
    …

    def eval (self, prim_dict):
        # unknown identifier !
        raise Exception("Runtime error")
```

Second homework

- `let` with concurrent/sequential bindings
- substituting values instead of expressions
- user-defined functions