Interpreters Activity

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

Objectives

You should be able to …

- Read the source code for a simple interpreter and find some bugs.
- Explain what lifting is.
- Add variables to an interpreter.

Walkthrough: Types

```
0 -- The Types
1
2 data Val = IntVal Integer
3    deriving (Show,Eq)
4
5 data Exp = IntExp Integer
6          | IntOpExp String Exp Exp
7    deriving (Show,Eq)
8
9 type Env = [(String,Exp)]
```

Bug 1 is on this slide.

Walkthrough: Bug 1

```
0 -- The Types
1
2 data Val = IntVal Integer
3    deriving (Show,Eq)
4
5 data Exp = IntExp Integer
6          | IntOpExp String Exp Exp
7    deriving (Show,Eq)
8
9 type Env = [(String,Val)]
```

Walkthrough: Lifting

```
0 intOps :: [(String, Integer -> Integer -> Integer)]  -- One variation
1 intOps = [ ("+",(+))
2          , ("-",(-))
3          , ("*",(*))
4          , ("/",div)]
5
6 liftIntOp :: (Integer -> Integer -> Integer) -> Val -> Val -> Val
7 liftIntOp f (IntVal i1) (IntVal i2) = IntVal (f i1 i2)
8 liftIntOp f _           _           = 0
```

This allows us to say

```
0 Main> liftIntOp (*) (IntVal 3) (IntVal 7)  -- Bug two is just above!
1 IntVal 21
```

Walkthrough: Bug 2

```
0 intOps :: Integral a => [(String, a -> a -> a)]  -- another Variation
1 intOps = [ ("+",(+))
2          , ("-",(-))
3          , ("*",(*))
4          , ("/",div)]
5
6 liftIntOp :: (Integer -> Integer -> Integer) -> Val -> Val -> Val
7 liftIntOp f (IntVal i1) (IntVal i2) = IntVal (f i1 i2)
8 liftIntOp f _           _           = IntVal 0          -- One way
9 liftIntOp f _           _           = error "Kablam!"   -- Another way
```

You will get a warning from the compiler.

Walkthrough, Eval

```
0 eval :: Exp -> Env -> Val
1 eval (IntExp i) _ = IntVal i   -- Why bother with IntVal then?
2
3 eval (IntOpExp op e1 e2) env =
4   let v1 = eval e1 env
5       v2 = eval e2 env
6       Just f = lookup op intOps
7   in liftIntOp f e1 e2    -- bug 3
```

Why does `lookup` return a `Maybe` type?

Adding Variables: Types

```
0  -- The Types
1
2  data Val = IntVal Integer
3     deriving (Show,Eq)
4
5  data Exp = IntExp Integer
6           | VarExp String  -- new type
7           | IntOpExp String Exp Exp
8     deriving (Show,Eq)
9
10 type Env = [(String,Val)]
```

Eval

```
0 eval (VarExp v) env =
1   case lookup v env of
2     Just val -> val
3     Nothing  -> IntVal 0
4     Nothing  -> error $ "Variable " ++ v ++ " undefined."
```

- Use line 3 if you like scripting languages.
- Use line 4 if you prefer to stay sane.

Adding Comparisons: Types

```
0 -- The Types
1
2 data Val = IntVal Integer
3          | BoolVal Bool
4    deriving (Show,Eq)
5
6 data Exp = IntExp Integer
7          | VarExp String
8          | IntOpExp String Exp Exp
9          | CompOpExp String Exp Exp
10   deriving (Show,Eq)
11
12 type Env = [(String,Val)]
```

Adding Comparisons: compOps

```
0 compOps :: Ord a => [(String, a -> a -> Bool)]
1 compOps = [ (">" ,(>))
2           , (">=",(>=))
3           , ("<" ,(<))
4           , ("<=",(<=))
5           , ("==",(==)
6           , ("/=",(/=))]
7
8 liftCompOp :: (Integer -> Integer -> Bool) -> Val -> Val -> Val
9 liftCompOp f (IntVal i1) (IntVal i2) = BoolVal (f i1 i2)
10 liftCompOp f _           _           = error "Type error!"
```

At this point the compiler warnings for the lifting functions will go away.

Adding Comparisons: eval

```
0 eval :: Exp -> Env -> Val
1 eval (CompOpExp op e1 e2) env =
2   let v1 = eval e1 env
3       v2 = eval e2 env
4       Just f = lookup op compOps
5   in liftCompOp f v1 v2
```