



CS4450: Principles of Programming Languages

Environments & Variables

Dr William Harrison

10/28/09



Today

- Environments are the structure we use to handle variables in the interpreter
 - the environment interface
 - ...and implementations
 - lists, higher-order functions, etc.



Environments

- an expression may have free variables
 - “free” means not bound by a “lambda”
- ... and we need something that says what value a free variable stands for

(+ **x** 2)

N.b., **x** is free

- an **environment** is a structure that associates variables with values

`interp (Var x) env = “look up x in env”`

● ● ● | Where are we going with this?

If you want to interpret variable, as in:

```
data Op      = Plus | Minus | Times | Div
data Exp     = Const Int | Aexp Op [Exp]
              | Var String
```

Let environments be lists and use “lookup” for applyenv:

```
interp env (Var x) = lookup x env
```

Q: What's the type of `interp` then?

Q: What about the other cases for `interp` – how do they change?



Environments – what are the basic operations?

- **The empty environment**
 - contains either no bindings
 - ...or initial bindings
 - it's the environment for starting execution
- **Extending an environment**; i.e., given variables x_1, \dots, x_n and values v_1, \dots, v_n
 - add bindings (x_i, v_i) to an env
- **Applying an environment to a variable**
 - look up most recent binding of x in env



Environment interface

```
data Value = ...
data Env   = ...

(* emptyenv :: Env *)
emptyenv = ...

(* extendenv :: [String] → [Value] → Env → Env *)
extendenv vars vals env = ...

(* applyenv :: ... *)
applyenv env var = ...
```

Depending on how **Env** is represented, will have different implementations of this interface

● ● ● | List implementation

```
data Env = Env [(String, Value)]
```

```
emptyenv = Env []
```

```
extendenv vars vals (Env rho)  
    = let  
        newbindings = zip vars vals  
    in  
        Env (newbindings ++ rho)
```

```
applyenv (Env rho) x = lookup x rho
```



The tweek function

```
tweek x v f = \ n -> if n==x then v else (f n)
```

Q: What's the type of tweek?



The tweek function

```
tweek x v f = \ n -> if n==x then v else (f n)
```

Q: What's the type of tweek?

```
tweek :: Eq a => a -> b -> (a -> b) -> (a -> b)
```



The tweek function

```
tweek x v f = \ n -> if n==x then v else (f n)
data Value  = I Int | R Float
env0        = \ n -> (I 0)
env1        = tweek "x" (R 3.14) env0
```

Use tweek to extend an environment (i.e., function from `String`→`Value`)

```
> :t env0 "x"
      (env0 "x") :: Value

> env1 "x"
      val it = R 3.14 : Value
```



Environments as functions

- Idea: represent an environment as a function of type `String → Value`
- Use “tweek” to make an `extendenv` operation
- Looking up a binding is just applying the function



Environments as functions*

```
data Env = Env (String → Value)
```

```
tweek (x,v) (Env rho)  
    = Env (\ n -> if n==x then v else rho n)
```

```
tweekList bindings env  
    = foldr tweek env bindings
```

```
emptyenv = Env (\ x -> I 0)
```

```
extendenv vars vals (Env rho)  
    = tweekList (zip vars vals) (Env rho)
```

```
applyenv (Env rho) x = rho x
```

* Definition of tweek corrected 11/4/09