

# Typical Applications

- Map functions on data structures
- Scheduling functions
- PolyTest – polymorphic testing
  - monomorphic testing over functions
- Many, many more...?
  - Today: quantification over functions is avoided
  - No Show
  - (CoArbitrary)

# Observations

- Key insight:
  - functions are infinite objects ...
  - ... but are only applied to a finite number of arguments in any terminating computation

# “Solution” #0 - unsafePerformIO

-- creating “magic” functions

```
makeMagicFun :: (a->b) ->  
              IO (IORef [(a,b)], a->b)
```

dirty trick...

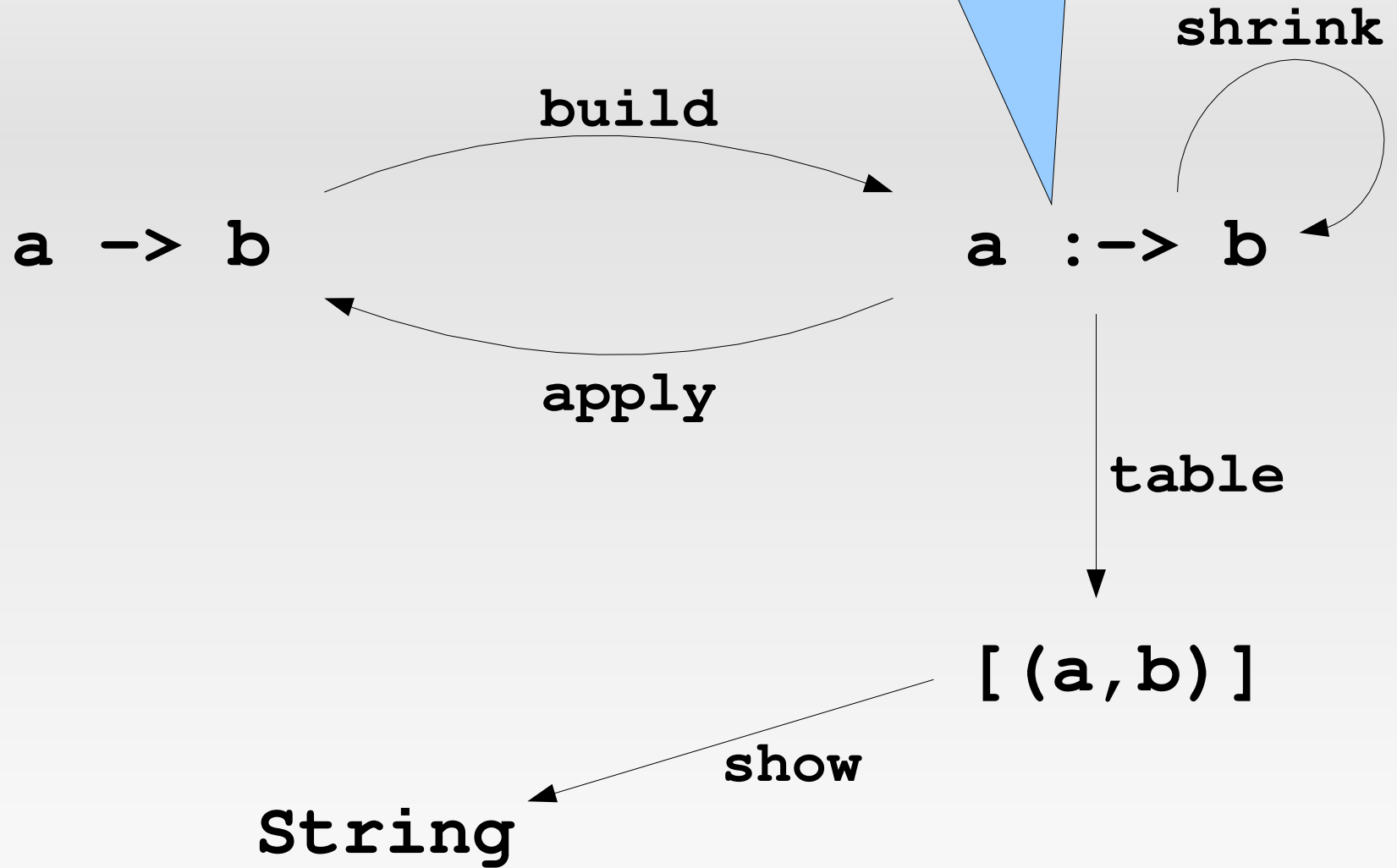
-- function modifier

```
data Fun a b = Fun (IORef [(a,b)]) (a->b)
```

...that does not  
even work well!

# Function Types

concrete  
representation of  
partial functions



# Types of Functions

-- creating concrete functions

**class** Argument a **where**

build :: (a -> b) -> (a :-> b)

-- applying concrete functions

apply :: (a :-> b) -> b -> (a -> b)

-- looking at concrete functions

table :: (a :-> b) -> [(a,b)]

-- shrinking concrete functions

shrink :: (a :-> b) -> [a :-> b]

the only way of  
creating a partial  
function

# Implementing Concrete Functions

```
data a :-> c where
  -- holy trinity
  Unit    :: c ->          ( () :->c)
  Pair    :: (a:->(b:->c)) -> ((a,b):->c)
  (:+:)   :: (a:->c) -> (b:->c) -> (Either a b:->c)

  -- all other argument types
  Map     :: (a->b) -> (b->a) -> (b:->c) -> (a:->c)

  -- for partial functions
  Nil     :: (a:->c)

instance Functor (a:->)
```

# Tabulating Concrete Functions

```
table :: (a :-> c) -> [(a, c)]
table (Unit c)      = [ ((), c) ]
table (Pair p)      = [ (x, y), c) | (x, q) <-table p
                           , (y, c) <-table q ]
table (p :+: q)     = [ (Left x, c) | (x, c) <-table p ]
                   ++ [ (Right y, c) | (y, c) <-table q ]
table Nil           = []
table (Map _ h p)   = [ (h x, c) | (x, c) <-table p ]
```

reason for  
specific type ()

result is often  
infinite...

# Building Concrete Functions

```
class Argument a where
```

```
  build :: (a->b) -> (a:->b)
```

```
instance Argument () where
```

```
  build f = Unit (f ())
```

```
instance (Argument a, Argument b) =>
```

```
        Argument (a,b) where
```

```
  build f = Pair (fmap build (build (curry f)))
```

```
instance (Argument a, Argument b) =>
```

```
        Argument (Either a b) where
```

```
  build f = build (f . Left) :+: build (f . Right)
```



# Building Concrete Functions

```
buildMap :: Argument b => (a->b) -> (b->a) ->  
          (a->c) -> (a->c)
```

```
buildMap g h f = Map g h (build (f . h))
```

```
-- instance for lists
```

```
instance Argument a => Argument [a] where
```

```
  build = buildMap g h
```

```
  where
```

```
    g []      = Left  ()  
    g (x:xs) = Right (x, xs)
```

```
    h (Left _)      = []  
    h (Right (x, xs)) = x:xs
```

...and Bool,  
Integer, Int,  
Char, Maybe, ...

# Applying Concrete Functions

```
apply :: (a :-> c) -> c -> (a -> c)
apply (Unit c)      _ ()      = c
apply (Pair p)      d (x,y) = apply (fmap (\q ->
                                         apply q d y) p) d x
apply (p :+: q)     d exy     = either (apply p d)
                                         (apply q d) exy
apply Nil           d _       = d
apply (Map g _ p)   d x       = apply p d (g x)
```

**-- providing a default argument**

```
func :: (a :-> c) -> (a -> c)
func cf = apply cf (snd (head (table cf)))
```

# Shrinking Concrete Functions

```
shrink' :: (c -> [c]) -> (a :-> c) -> [a :-> c]
shrink' shr (Pair p) =
  [Pair p' | p' <- shrink' (\q -> shrink' shr q) p]

shrink' shr (p :+: q) =
  [p :+: Nil | not (isNil q)] ++
  [Nil :+: q | not (isNil p)] ++
  [p' :+: q | p' <- shrink' shr p ] ++
  [p :+: q' | q' <- shrink' shr q ]

shrink' shr (Unit c) =
  [ Nil ] ++
  [ Unit c' | c' <- shr c ]
```

# Shrinking Concrete Functions

```
shrink' :: (c -> [c]) -> (a :-> c) -> [a :-> c]
```

```
...
```

```
shrink' shr Nil =  
  []
```

```
shrink' shr (Map g h p) =  
  [ Map g h p' | p' <- shrink' shr p ]
```

# Fun Modifier

do not show  
before shrinking!

```
data Fun a b = Fun (a->b) (a->b)
```

```
instance (Show a, Show b) => Show (Fun a b)  
  -- uses show on (a->b)
```

```
instance (CoArbitrary a, Arbitrary b) =>  
  Arbitrary (Fun a b)  
  -- uses arbitrary on (a->b)  
  -- uses shrink on (a->b)
```

(demo)

# Extensions

more efficient  
shrinking  
methods

```
data a :-> c where
```

```
...
```

```
-- finite tables
```

```
Table :: Eq a => [(a, c)] -> (a :-> c)
```

```
-- higher-order functions
```

```
Function :: [a] -> ([b] :-> c) -> ((a :-> b) :-> c)
```

only works for  
second-order  
functions...

concrete  
function; need to  
be able to “show”

# Conclusions

- Modifiers are a useful idiom
- Shrinking & showing at the same time
- Higher-order functions?
- Related: Concrete algorithms, generalized tries