Pragmatism, Puritanism and Functional Programming

Ben Moseley — ben@moseley.name

Pragmatism, Puritanism and Functional Programming

Ben Moseley — ben@moseley.name

Pragmatism, Puritanism and Functional Programming

Ben Moseley — ben@moseley.name

Why am I here?

Why am I here?

- Software in Industry ...

Why am I here?

- Software in Industry ...
- ... is broken.

Why am I here?

- Software in Industry ...
- ... is broken.
- ... because it's too **complex** !

TECH MESH
LONDON 2012

Why am I here?

- Software in Industry ...
- ... is broken.
- ... because it's too **complex** !
- My perspective on why FP techniques can help
    - Based on 8 years of Haskell (5 full-time commercial)

Haskell…

Haskell...

- Haskell makes ***programs***:
    - Simpler to Write
    - Simpler to Read (ie understand)

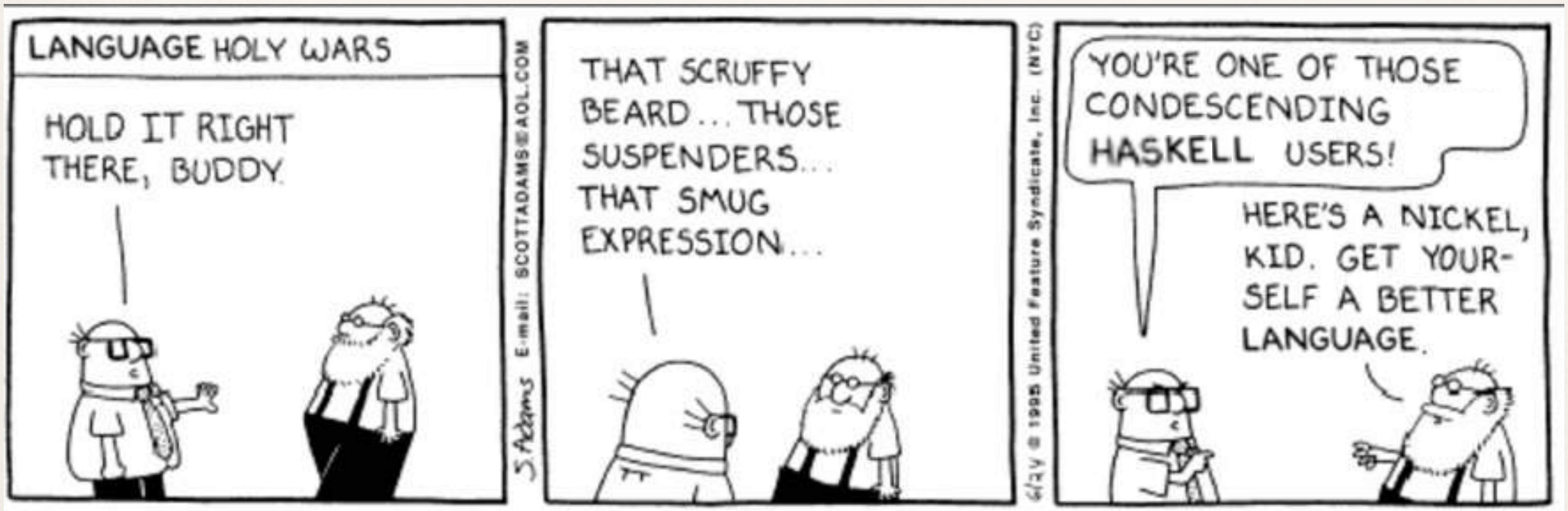
Haskell…

- Haskell makes ***programs***:
    - Simpler to Write
    - Simpler to Read (ie understand)
- Is Haskell simple?

Haskell...

- Haskell makes ***programs***:
  - Simpler to Write
  - Simpler to Read (ie understand)
- Is Haskell simple?
- Is Haskell easy?

A Better Language?



What is important about FP?

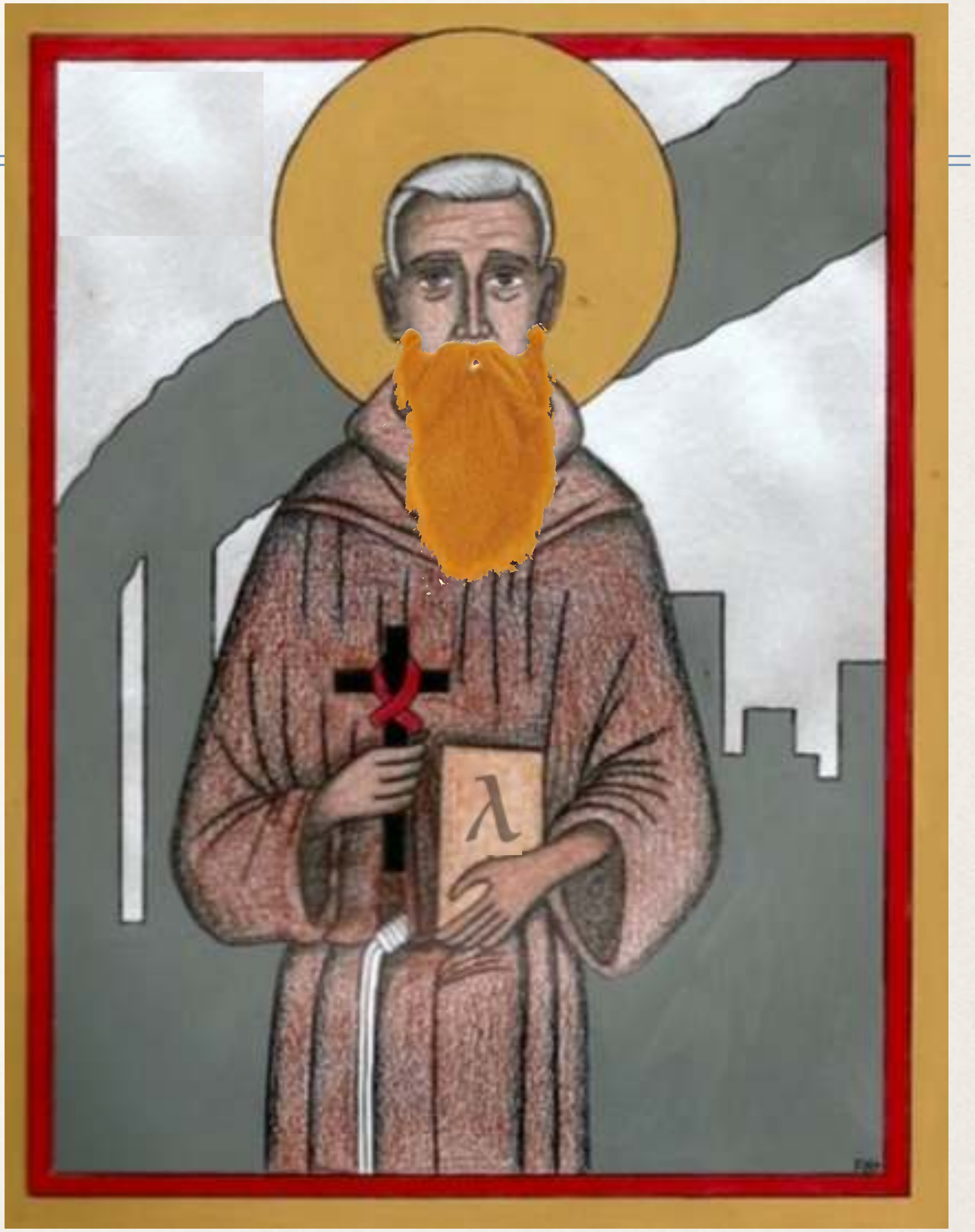
- Purity
- Type Systems
- Higher Order Functions (HOFs)
- Laziness
- Extensive abstract libraries

What makes FP hard to learn?

What makes FP hard to learn?

- Purity
- Type Systems
- HOFs
- Laziness
- Extensive abstract libraries

Purity



“The functional programmer sounds rather like a medieval monk, denying himself the pleasures of life in the hope that it will make him virtuous.”

-- John Hughes [~1984]

Modularity

- We build systems by naming and assembling chunks of code
- Interested in:
  - ***Result*** of a chunk
  - What a chunk ***Does***

```
var y = chunk1(a,b);
var z = chunk2(c);
chunk3(x);
chunk4(y,z);
```

Read & Understand...

Read & Understand...

v0.9

```
var x = chunk1(a,b);
var y = chunk2(c);
chunk3(x,y);
```

Read & Understand...

v1.0

```
var x = chunk1(a,b);
var y = chunk2(c);
chunk3(x,y);
```

Read & Understand...

v1.0

```
var x = chunk1(a,b);
var y = chunk2(c);
chunk3(x,y);
```

v2.0

```
var y = chunk2(c);
var x = chunk1(a,b);
chunk3(x,y);
```

Read & Understand...

v1.0

```
var x = chunk1(a,b);
var y = chunk2(c);
chunk3(x,y);
```

```
var x = chunk4(a,b);
var y = chunk4(a,b);
chunk5(x,y);
```

v2.0

```
var y = chunk2(c);
var x = chunk1(a,b);
chunk3(x,y);
```

```
var x = chunk4(a,b);
var y = x;
chunk5(x,y);
```

What *is* Purity?

```
var x = chunk1(a,b);
var y = chunk2(c);
chunk3(x,y);
```

```
var x = chunk4(a,b);
var y = chunk4(a,b);
chunk5(x,y);
```

What *is* Purity?

```
var x = chunk1(a,b);
var y = chunk2(c);
chunk3(x,y);
```

```
var x = chunk4(a,b);
var y = chunk4(a,b);
chunk5(x,y);
```

```
f :: A -> B
```

Purity Pitfall 1 - Effects

- Pitfall 1 : Sometimes you *want* chunks which have state:

```
f :: A -> B
```

```
var y = chunk1(a,b);
var z = chunk1(a,b);
chunk3(x);
chunk4(y,z);
```

```
f :: (A,s) -> (B,s)
```

```
var (y,s2) = chunk1(a,b,s);
var (z,s3) = chunk1(a,b,s2);
var s4 = chunk3(x,s3);
var s5 = chunk4(y,z,s4);
```

Purity Pitfall 1 - Effects

- Pitfall 1 : Sometimes you *need* chunks which can fail:

```
f :: A -> B
```

```
var y = chunk1(a,b);
var z = chunk2(c);
chunk3(x);
chunk4(y,z);
```

```
f :: A -> Maybe B
```

```
case chunk1(a,b) of
  Just y ->
    case chunk2(c) of
      Just z ->
        case chunk3(x) of
          Just () ->
            case chunk4(y,z)
  ...
```

Purity Pitfall 1 - Effects

- Pitfall 1b : The syntax is quite verbose

```
f :: (A,s) -> (B,s)
```

```
var (y,s2) = chunk1(a,b,s);
var (z,s3) = chunk1(a,b,s2);
var s4 = chunk3(x,s3);
var s5 = chunk4(y,z,s4);
```

Purity Pitfall 1 - Effects

- Pitfall 1b : The syntax is quite verbose

```
f :: (A,s) -> (B,s)
```

```
var (y,s2) = chunk1(a,b,s);
var (z,s3) = chunk1(a,b,s2);
var s4 = chunk3(x,s3);
var s5 = chunk4(y,z,s4);
```

```
f :: A -> State s B
```

```
do
  y <- chunk1(a,b)
  z <- chunk1(a,b)
  chunk3(x)
  chunk4(y,z)
```

Purity - Is this all a con ?

- SIDE-EFFECTS pitfall is avoided by:
  - Take them away....
  - ..... put them back

Purity - Is this all a con ?

Purity - Is this all a con ?

- SIDE-EFFECTS pitfall is avoided by:
  - Take them away....
  - ..... put them back if necessary

Purity - Is this all a con ?

- SIDE-EFFECTS pitfall is avoided by:
  - Take them away....
  - ..... put them back if necessary
  - ... in a restricted way that makes the code simpler to understand!!

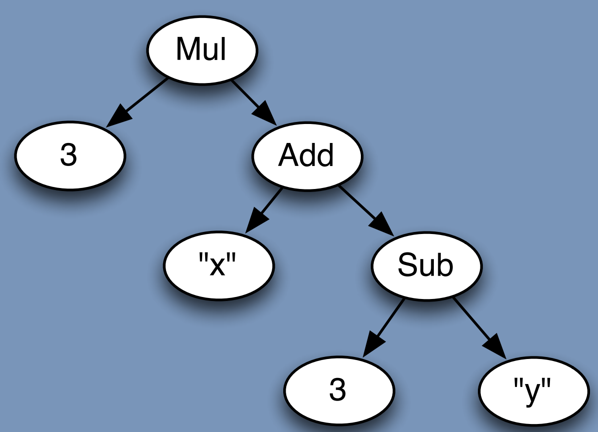
Purity - Is this all a con ?

- SIDE-EFFECTS pitfall is avoided by:
  - Take them away....
  - ..... put them back if necessary
  - ... in a restricted way that makes the code simpler to understand!!

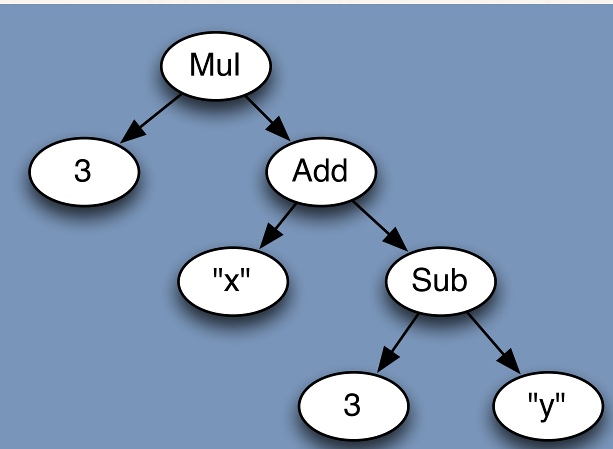
1. Each chunk uses only the effects *it* needs
2. The exact effects are visible in the type
3. We can now control how different effects interact

Purity Pitfall 2 - Equality

`var x =`



`var y =`

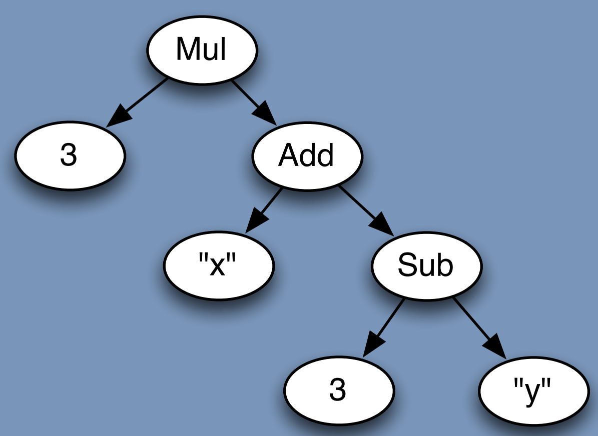



```
x == y
```

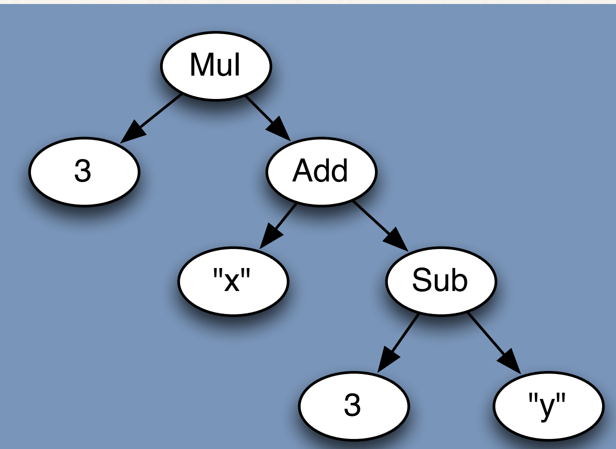
- Yes .... but ....
-

Purity Pitfall 2 - Equality

`var x =`



`var y =`



`x == y`

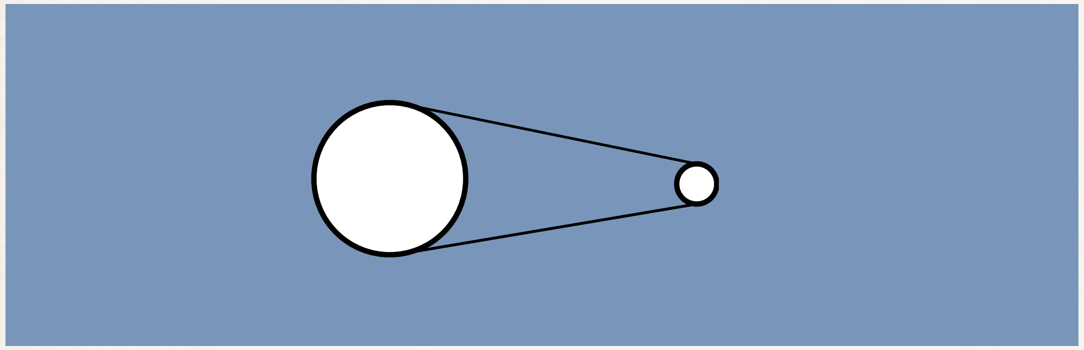
- Yes .... but ....
- ... slow to find out

Purity Pitfall 2 - Equality

- POINTER EQUALITY pitfall is avoided by:
  - Take it away....
  - ..... put it back if necessary
  - ... in a restricted way that makes the code simpler to understand!!
    - Explicit
    - Localized

Purity Pitfall 3 - Identity

- CAD system

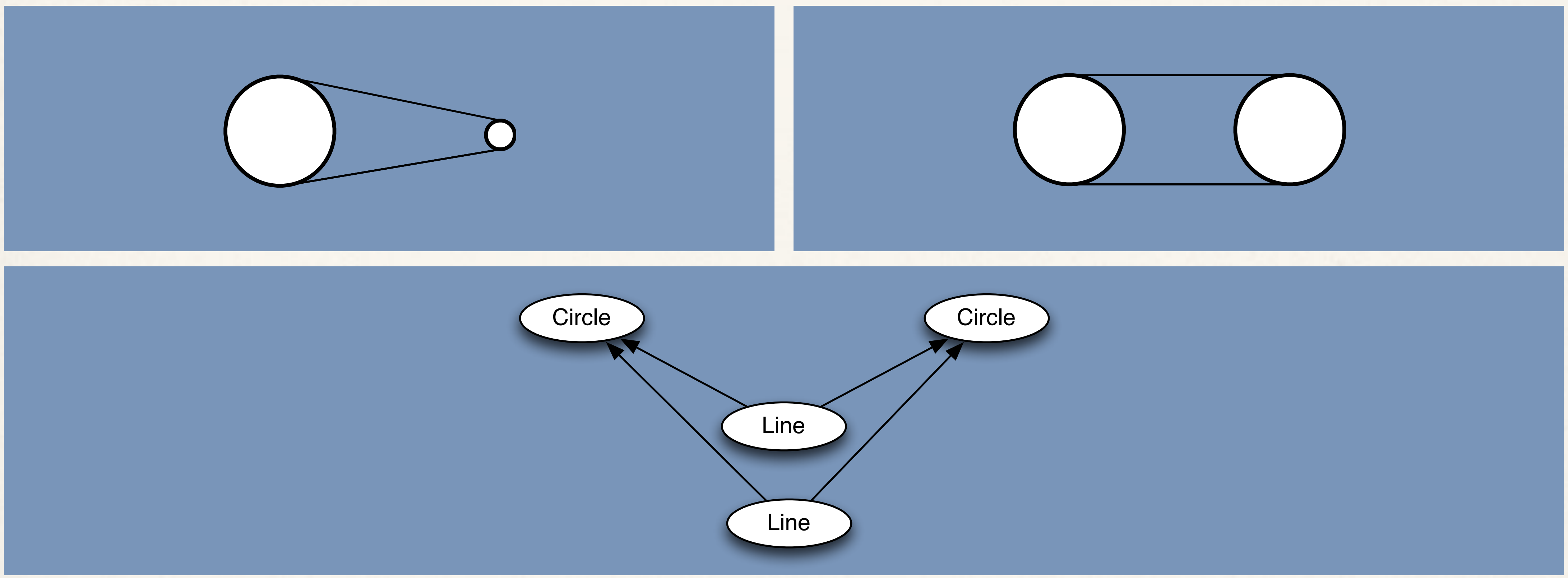


Purity Pitfall 3 - Identity

- CAD system

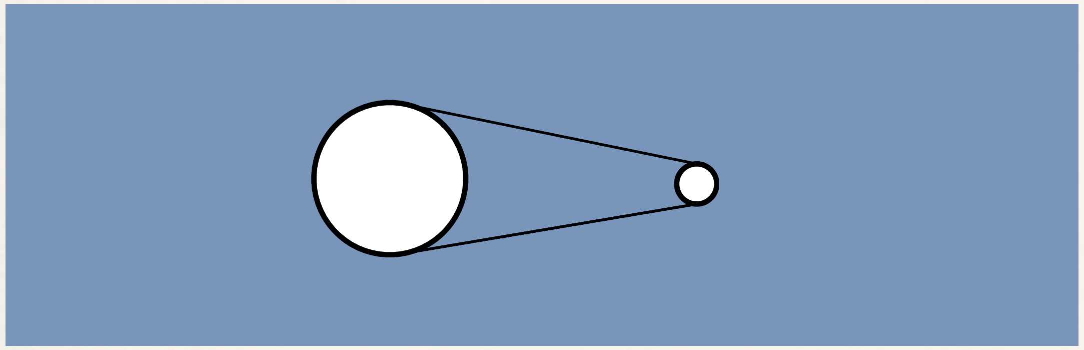
Purity Pitfall 3 - Identity

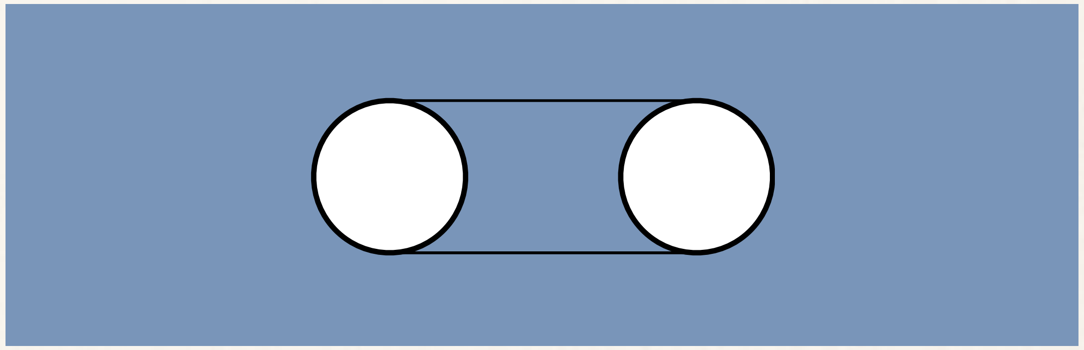
- CAD system

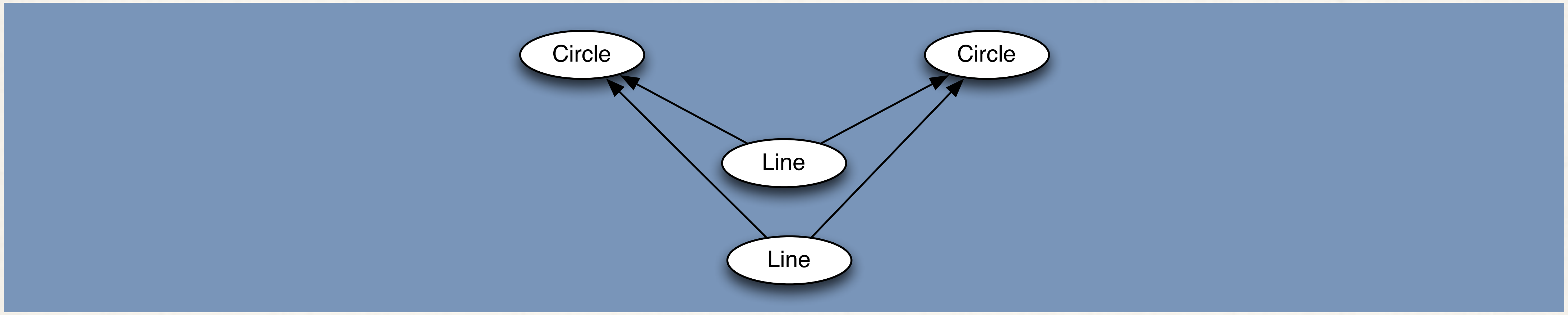


Purity Pitfall 3 - Identity

- CAD system



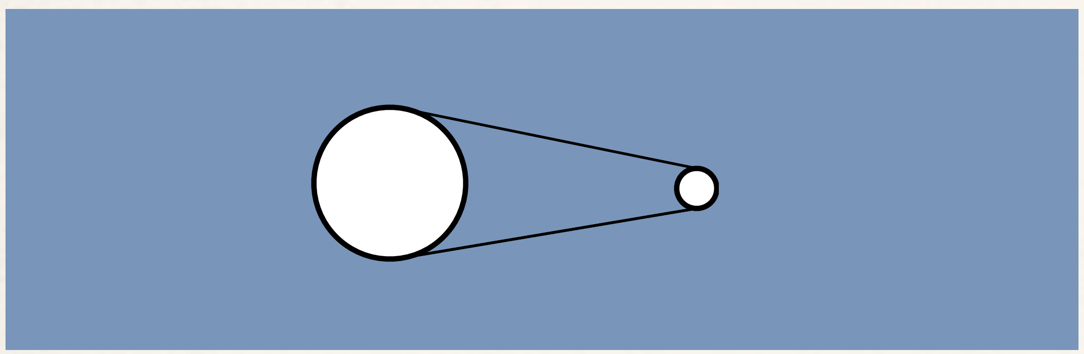


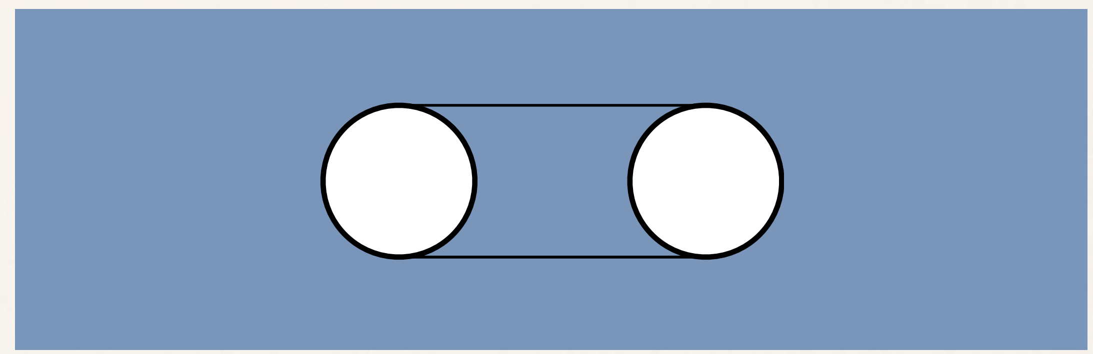


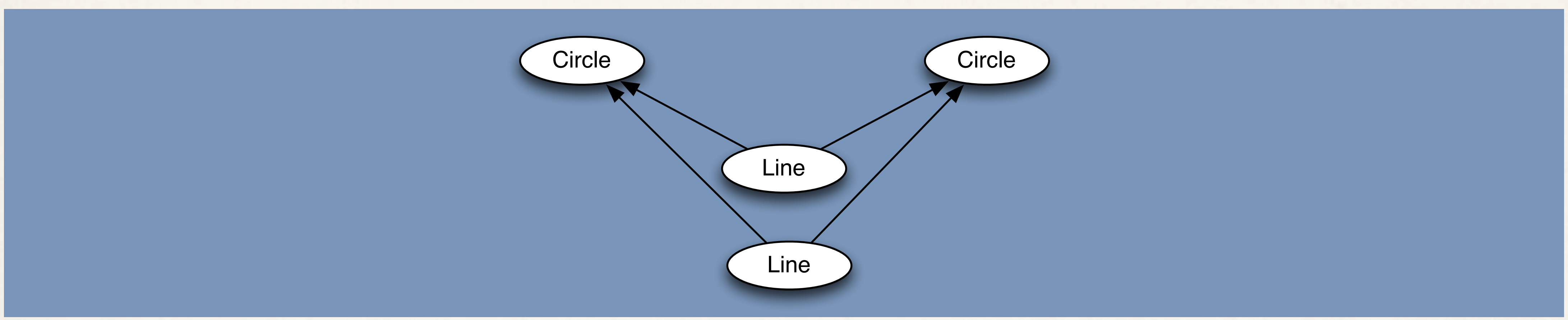
ID: 1001	Circle	Centre: 0,0 Radius: 100
ID: 1002	Circle	Centre: 250,0 Radius: 20
ID: 1003	Line	CircleOne: 1001 CircleTwo: 1002
ID: 1004	Line	CircleOne: 1001 CircleTwo: 1002

Purity Pitfall 3 - Identity

- CAD system







Circle	Centre: 250,0 Radius: 20

ID: 1001	Circle	Centre: 0,0 Radius: 100
ID: 1002	Circle	Centre: 250,0 Radius: 100
ID: 1003	Line	CircleOne: 1001 CircleTwo: 1002
ID: 1004	Line	CircleOne: 1001 CircleTwo: 1002

Purity Pitfall 3 - Identity

- IDENTITY pitfall is avoided by:
  - Take it away....
  - ..... put it back if necessary
  - ... in an explicit way that makes the code simpler to understand!!

Purity - Pitfalls

1. Sometimes you really *want* **Side Effects** (eg Failure / Mutability)

2. Sometimes you really *want* **Pointer Equality**

3. Sometimes you really *want* **Identity**

Purity - Pitfalls

1. Sometimes you really *want* **Side Effects** (eg Failure / Mutability)

2. Sometimes you really *want* **Pointer Equality**

3. Sometimes you really *want* **Identity**

4. **Learning Curve** - Need to ***think*** in a different (*unnatural*) way

Type Systems

- Static vs Dynamic Types
- Benefits…
- Pitfalls

Type Systems - Confusion

- Statically Typed Languages *are* Dynamically Typed Languages
  - ... with an extra feature:
    - ... any “Classification” you make can be Static (or Dynamic)
  - ... typically use this in their std libs

Types in Haskell

```
data Bool    = True
             | False
```

Types in Haskell

```
data Bool   = True
            | False
```

```
data Int    = ...
```

```
data Float  = ...
```

```
data String = ...
```

Dynamic Types in Haskell

```
data Value = TagInt    Int
           | TagFloat  Float
           | TagBool   Bool
           | TagString String
```

FilePaths 1 - Strings vs Paths

Strings vs Paths

FilePaths 1 - Strings vs Paths

Strings vs Paths	`String`	No Distinction

FilePaths 1 - Strings vs Paths

Strings vs Paths

`String`

No Distinction

```
appendPath :: (String, String) -> String
```

FilePaths 1 - Strings vs Paths

Strings vs Paths | String | No Distinction

```
appendPath :: (String, String) -> String
```

```
appendPath ("\usr\bm", "bar.csv")
```

FilePaths 1 - Strings vs Paths

Strings vs Paths | String | No Distinction

```
appendPath :: (String, String) -> String
```

```
appendPath (“line1\n line2\r\n...”, “bar.csv”)
```

...no error

FilePaths 1 - Strings vs Paths

Strings vs Paths

No Distinction

```
data Value = TagInt    Int
           | TagFloat  Float
           | TagBool   Bool
           | TagString String
           | TagPath   String
```

Runtime Distinction

```
appendPath :: (Value, Value) -> Value
```

FilePaths 1 - Strings vs Paths

Strings vs Paths

No Distinction

```
data Value = TagInt    Int
           | TagFloat  Float
           | TagBool   Bool
           | TagString String
           | TagPath   String
```

Runtime Distinction

```
appendPath :: (Value, Value) -> Value
```

```
appendPath (TagString “line1\n line2\r\n...”,
            TagPath   “bar.csv”)
```

...dynamic error

FilePaths 1 - Strings vs Paths

Strings vs Paths

No Distinction

```
data Value =...| TagString String
               | TagPath   String
```

Runtime

FilePaths 1 - Strings vs Paths

Strings vs Paths

No Distinction

```
data Value =...| TagString String
               | TagPath   String
```

Runtime

```
data Path = Path String
```

Compile time

FilePaths 1 - Strings vs Paths

Strings vs Paths

No Distinction

```
data Value =...| TagString String
               | TagPath   String
```

Runtime

```
data Path = Path String
```

Compile time

```
Path “/usr/bm” :: Path
```

```
Path “bar.csv” :: Path
```

FilePaths 1 - Strings vs Paths

Strings vs Paths

No Distinction

```
data Value =...| TagString String
               | TagPath   String
```

Runtime

```
data Path = Path String
```

Compile time

```
Path "/usr/bm" :: Path
```

```
Path "bar.csv" :: Path
```

```
appendPath :: (Path, Path) -> Path
```

FilePaths 1 - Strings vs Paths

Strings vs Paths

No Distinction

```
data Value =...| TagString String
               | TagPath   String
```

Runtime

```
data Path = Path String
```

Compile time

```
Path “/usr/bm” :: Path
```

```
Path “bar.csv” :: Path
```

```
appendPath :: (Path, Path) -> Path
```

```
appendPath (“line1\n line2\r\n...”, Path “bar.csv”)
```

...static error

FilePaths 1 - Strings vs Paths

Strings vs Paths

No Distinction

```
data Value =...| TagString String
               | TagPath   String
```

Runtime

```
data Path = Path String
```

Compile time

```
Path "/usr/bm" :: Path
```

```
Path "bar.csv" :: Path
```

```
appendPath :: (Path, Path) -> Path
```

```
appendPath (    Path "/usr/bm",      Path "bar.csv")
```

FilePaths 1 - Strings vs Paths

Strings vs Paths

No Distinction

```
data Value =...| TagString String
               | TagPath   String
```

Runtime

```
data Path = Path String
```

Compile time

```
Path “/usr/bm” :: Path
```

```
Path “bar.csv” :: Path
```

```
appendPath :: (Path, Path) -> Path
```

```
appendPath (    Path “foo.txt”,      Path “bar.csv”)
```

...no error

FilePaths 2 - File vs Dir Paths

Strings vs Paths

No Distinction

```
data Value =...| TagString String
               | TagPath   String
```

Runtime

Compile time

File Paths vs Dir Paths

```
data Path = FilePath String
          | DirPath  String
```

Runtime

```
appendPath :: (Path, Path) -> Path
```

```
appendPath (    Path “foo.txt”,      Path “bar.csv”)
```

FilePaths 2 - File vs Dir Paths

Strings vs Paths

No Distinction

```
data Value =...| TagString String
               | TagPath   String
```

Runtime

Compile time

File Paths vs Dir Paths

```
data Path = FilePath String
          | DirPath  String
```

Runtime

```
appendPath :: (Path, Path) -> Path
```

```
appendPath (FilePath “foo.txt”, FilePath “bar.csv”)
```

...dynamic error

FilePaths 2 - File vs Dir Paths

Strings vs Paths

No Distinction

```
data Value =...| TagString String
               | TagPath   String
```

Runtime

Compile time

File Paths vs Dir Paths

```
data Path = FilePath String
          | DirPath  String
```

Runtime

```
data Path    = Path String
```

No Distinction

```
Path “/usr” :: Path
Path “fred.csv” :: Path
appendPath :: (Path    , Path    ) -> Path
```

FilePaths 2 - File vs Dir Paths

Strings vs Paths

No Distinction

```
data Value =...| TagString String
               | TagPath   String
```

Runtime

Compile time

File Paths vs Dir Paths

```
data Path = FilePath String
          | DirPath  String
```

Runtime

```
data Path fd = Path String
```

Compile time

```
Path "/usr" :: Path
```

```
Path "fred.csv" :: Path
```

```
appendPath :: (Path     , Path     ) -> Path
```

FilePaths 2 - File vs Dir Paths

Strings vs Paths — No Distinction

```
data Value =...| TagString String
               | TagPath   String
```

Runtime

Compile time

File Paths vs Dir Paths

```
data Path = FilePath String
          | DirPath  String
```

Runtime

```
data Path fd = Path String
```

Compile time

```
Path “/usr” :: Path Dir
Path “fred.csv” :: Path File
appendPath :: (Path     , Path     ) -> Path
```

FilePaths 2 - File vs Dir Paths

Strings vs Paths — No Distinction

```
data Value =...| TagString String
               | TagPath   String
```

Runtime

Compile time

File Paths vs Dir Paths

```
data Path = FilePath String
          | DirPath  String
```

Runtime

```
data Path fd = Path String
```

Compile time

```
Path “/usr” :: Path Dir
Path “fred.csv” :: Path File
```

```
appendPath :: (Path Dir, Path fd) -> Path fd
```

Static Types - Sliding Scale

Strings vs Paths	`String`	No Distinction
	`data Value =...| TagString String` `| TagPath String`	Runtime
	`data Path = Path String`	Compile time
File Paths vs Dir Paths	`data Path = Path String`	No Distinction
	`data Path = FilePath String` `| DirPath String`	Runtime
	`data Path fd = Path String`	Compile time

Static Type Systems - Benefits

- Make code simpler to ***write***
  - detect errors earlier
  - the first thing you write - a design language
  - Refactoring, IDEs, Performance

Static Type Systems - Benefits

- Make code simpler to ***read***
  - Help locate bugs (in type-correct code !)
  - Help understand normal code
  - Help understand very abstract code
  - Document side-effects

Types: Assumptions & Guarantees

```
mychunk :: [Int] -> [Int]
mychunk = ...
```

Types: Assumptions & Guarantees

```
mychunk :: [Int] -> [Int]
mychunk = ...
```

Types: Assumptions & Guarantees

```
mychunk :: [Int] -> [Int]
mychunk = ...
```

Types: Assumptions & Guarantees

```
mychunk :: [a] -> [a]
mychunk = ...
```

Types: Assumptions & Guarantees

```
mychunk :: [a] -> [a]
mychunk = ...
```

Types: Assumptions & Guarantees

```
mychunk :: [a] -> [a]
mychunk = ...
```

Types: Assumptions & Guarantees

```
mychunk :: Eq a => [a] -> [a]
mychunk = ...
```

Types: Assumptions & Guarantees

```
mychunk :: Num a => [a] -> [a]
mychunk = ...
```

Types: Assumptions & Guarantees

```
mychunk :: Typeable a => [a] -> [a]
mychunk = ...
```

Types: Assumptions & Guarantees

```
mychunk :: [a] -> [a]
mychunk = ...
```

Types: Locating Bugs

```
processWith f = g (f [100,101])

g xs = ...
```

Types: Locating Bugs

```
processWith :: (forall a. [a] -> [a]) -> [Int]
processWith f = g (f [100,101])

g :: [Int] -> [Int]
g xs = ...
```

Types: Locating Bugs

```
processWith :: (forall a. [a] -> [a]) -> [Int]
processWith f = g (f [100,101])

g :: [a] -> [a]
g xs = ...
```

Types: Understanding Code

```
trans xs ys =
    case xs of
      []       -> []
      ((a,b):xs) -> (map (\(x,y) -> (a,y)) $ f ys)
                    ++ trans xs ys
          where
            f [] = []
            f ((c,d):ys) | c==b = (b,d) : f ys
                         | otherwise = f ys
```

Types: Understanding Code

```
trans :: [(Int,Int)] -> [(Int,Int)] -> [(Int,Int)]
trans xs ys =
    case xs of
      []         -> []
      ((a,b):xs) -> (map (\(x,y) -> (a,y)) $ f ys)
                    ++ trans xs ys
          where
            f [] = []
            f ((c,d):ys) | c==b = (b,d) : f ys
                         | otherwise = f ys
```

Types: Understanding Code

```
trans :: Eq b => [(b, b)] -> [(b, b)] -> [(b, b)]
trans xs ys =
    case xs of
      []       -> []
      ((a,b):xs) -> (map (\(x,y) -> (a,y)) $ f ys)
                   ++ trans xs ys
            where
              f [] = []
              f ((c,d):ys) | c==b = (b,d) : f ys
                           | otherwise = f ys
```

Types: Understanding Code

```
trans :: Eq b => [(b, b)] -> [(b, b)] -> [(b, b)]
trans xs ys =
    case xs of
      []        -> []
      ((a,b):xs) -> (map (\(x,y) -> (a,y)) $ f ys)
                    ++ trans xs ys
          where
            f [] = []
            f ((c,d):ys) | c==b = (b,d) : f ys
                         | otherwise = f ys
```

Types: Understanding Code

```
trans :: Eq b => [(a, b)] -> [(b, c)] -> [(a, c)]
trans xs ys =
    case xs of
      []        -> []
      ((a,b):xs) -> (map (\(x,y) -> (a,y)) $ f ys)
                   ++ trans xs ys
          where
            f [] = []
            f ((c,d):ys) | c==b = (b,d) : f ys
                         | otherwise = f ys
```

Types: Understanding Code

```haskell
trans :: Eq b => [(a, b)] -> [(b, c)] -> [(a, c)]
trans xs ys =
    case xs of
      []       -> []
      ((a,b):xs) -> (map (\(x,y) -> (a,y)) $ f ys)
                    ++ trans xs ys
          where
            f [] = []
            f ((c,d):ys) | c==b = (b,d) : f ys
                         | otherwise = f ys
```

Types: Understanding Code

```
trans :: Eq b => [(a, b)] -> [(b, c)] -> [(a, c)]
trans xs ys =
  [(a,c) | (a,b1) <- xs, (b2,c) <- ys, b1 == b2]
```

Types: Handling very abstract code

What is ‘h’ ?

```
ala w h f = unwrap . h (w . f)
```

Types: Handling very abstract code

```
ala :: Newtype srb =>
        (b -> sb) ->
       ((a -> sb) -> ta -> srb) ->
        (a ->  b) ->
       (ta -> Unwrap srb)
ala w h f = unwrap . h (w . f)
```

Types: Handling very abstract code

```
ala :: Newtype srb =>
        (b -> sb) ->
       ((a -> sb) -> ta -> srb) ->
        (a ->  b) ->
       (ta -> Unwrap srb)
ala w h f = unwrap . h (w . f)
```

Types: Side Effects

```
f :: A -> State s B
```

```
do
  y <- chunk1(a,b)
  z <- chunk1(a,b)
  chunk3(x)
  chunk4(y,z)
```

Types: Side Effects

```
chunk0 a = do
           y <- chunk1(a,b)
           z <- chunk1(a,b)
           chunk3(x)
           chunk4(y,z)
           return z
```

Types: Side Effects

```
chunk0 :: A -> State (Int,Bool) Z
chunk0 a = do
           y <- chunk1(a,b)
           z <- chunk1(a,b)
           chunk3(x)
           chunk4(y,z)
           return z
```

Types: Side Effects

```
chunk0 :: Int -> ReaderT Config
                   (StateT Connection
                    (EitherT DBError Int))
chunk0 a = do
           y <- chunk1(a,b)
           z <- chunk1(a,b)
           chunk3(x)
           chunk4(y,z)
           return z
```

Type Systems - Benefits

- Make code simpler to ***read***
  - Help locate bugs (in type-correct code !)
  - Help understand normal code
  - Help understand very abstract code
  - Document side-effects
- Make code simpler to UNDERSTAND !!!!!!!

Type Systems - Pitfalls

Type Systems - Pitfalls

- Longer to get running code
- Changing your mind
- Learning Curve - “A Second Language”

Summary

- Functional Programming makes code **SIMPLER**
  - PURITY + TYPES are the two biggest ways it does this

End