IMMUTABILITY & PURE FUNCTIONS

Malte Neuss

REASONING ABOUT CODE

```
def main()
  prices = [5,2,7]
  # assert prices[0] == 5
```

REASONING ABOUT CODE

```
def main()
  prices = [5,2,7]
  # assert prices[0] == 5
```

```
  minPrice = minimum(prices)
  # assert  minPrice == 2
```

REASONING ABOUT CODE

```
def main()
  prices = [5,2,7]
  # assert prices[0] == 5
```

```
  minPrice = minimum(prices)
  # assert  minPrice == 2
```

```
  # assert prices[0] == 5                    # Correct?
```

REASONING ABOUT CODE

```
def main()
  prices = [5,2,7]
  # assert prices[0] == 5
```

```
  minPrice = minimum(prices)
  # assert  minPrice == 2
```

```
  # assert prices[0] == 5                    # Correct?
```

```
def minimum(values)
   values.sort()                             # surprise
   return values[0]
```

CONTENT

CONTENT

- Immutability

CONTENT

- Immutability
- Pure Functions

CONTENT

- Immutability
- Pure Functions
- OO vs FP Styles

CONTENT

- Immutability
- Pure Functions
- OO vs FP Styles
- Reasoning

CONTENT

- Immutability
- Pure Functions
- OO vs FP Styles
- Reasoning
- Referential Transparency

CONTENT

- Immutability
- Pure Functions
- OO vs FP Styles
- Reasoning
- Referential Transparency

Easier reasoning, less invalid state

IMMUTABILITY

No mutation:

```
def minimum(values)
   # values.sort()                                  # mutation
   other = sorted(values)                           # no mutation
   return other[0]
```

IMMUTABILITY

No mutation:

```
def minimum(values)
    # values.sort()                    # mutation
    other = sorted(values)             # no mutation
    return other[0]
```

No reassign:

```
  value = 1                            # Assign
# value = 2                            # Reassign
  other = 2
```

REASONING

```
def minimum(values)
    # values.sort()                          # surprise
    other = sorted(values)                   # no surprise
    return other[0]
```

REASONING

```
def minimum(values)
    # values.sort()                          # surprise
    other = sorted(values)                   # no surprise
    return other[0]
```

```
def main()
   prices = [5,2,7]                          # If immutable,
   # assert prices[0] == 5
   minPrice = minimum(prices)
   # assert prices[0] == 5                   # Guaranteed
```

TYPE CHECKER SUPPORT

Make illegal state (more) unrepresentable

IMMUTABLE BUILT-IN CLASSES

Immutable interface:

```
mySet = frozenset([1, 2, 3])                    # Python
mySet.add(4)                                    # type error
```

IMMUTABLE BUILT-IN CLASSES

Immutable interface:

```
mySet = frozenset([1, 2, 3])                    # Python
mySet.add(4)                                    # type error
```

Copy on change:

```
val myList = immutable.List(1, 2, 3)            // Scala
val other  = myList.appended(4)                 // other list
```

IMMUTABLE CUSTOM CLASSES

Immutable interface around mutable data:

```
class MyClass:
   _value: int                                  # hide mutables

   def getValue()                               # no setters
     return _value

   def calcSth()                                # _value read only
     return _value*2

   def incremented()                            # Copy on change
     return MyClass(_value+1)
```

IMMUTABLE CUSTOM CLASSES

With extra language support:

```python
@dataclass(frozen=True)                    # Python
class MyClass:
   value: int

object       = MyClass(1)
object.value = 2                           # compile error!
```

IMMUTABLE CUSTOM CLASSES

With extra language support:

```python
@dataclass(frozen=True)                    # Python
class MyClass:
   value: int

object       = MyClass(1)
object.value = 2                           # compile error!
```

```typescript
interface MyInterface {                    // Typescript
  readonly value: int;
}
```

IMMUTABLE VARIABLES

Mutable:

```
var value = 1                                   // Typescript
value = 2                                       // ok
```

IMMUTABLE VARIABLES

Mutable:

```
var value = 1                                   // Typescript
value = 2                                       // ok
```

Immutable:

```
const value = 1                                 // Typescript
value = 2                                       // type error!
```

PURE FUNCTIONS

$$f : \mathbb{R} \to \mathbb{R} \qquad \textbf{Type}$$

$$f(x) = x + \pi \qquad \textbf{Body}$$

PURE FUNCTIONS

$$f : \mathbb{R} \to \mathbb{R} \qquad \text{Type}$$

$$f(x) = x + \pi \qquad \text{Body}$$

```
def f(x: float) -> float:                    # Type
    return x + math.pi                       # Body
```

PURE FUNCTIONS

$$f : \mathbb{R} \to \mathbb{R} \qquad \text{Type}$$

$$f(x) = x + \pi \qquad \text{Body}$$

```
def f(x: float) -> float:                    # Type
    return x + math.pi                       # Body
```

- Same input, same output

PURE FUNCTIONS

$$f : \mathbb{R} \to \mathbb{R} \qquad \text{Type}$$

$$f(x) = x + \pi \qquad \text{Body}$$

```
def f(x: float) -> float:                    # Type
    return x + math.pi                       # Body
```

- Same input, same output
- No side-effects

PURE FUNCTIONS

$$f : \mathbb{R} \to \mathbb{R} \qquad \text{Type}$$

$$f(x) = x + \pi \qquad \text{Body}$$

```
def f(x: float) -> float:                    # Type
    return x + math.pi                       # Body
```

- Same input, same output
- No side-effects
- External immutable values or constants ok

SIDE-EFFECT

SIDE-EFFECT

- File read call

SIDE-EFFECT

- File read call
- File write call

SIDE-EFFECT

- File read call
- File write call
- Network call

SIDE-EFFECT

- File read call
- File write call
- Network call
- I/O call

SIDE-EFFECT

- File read call
- File write call
- Network call
- I/O call
- Throwing exception

SIDE-EFFECT

- File read call
- File write call
- Network call
- I/O call
- Throwing exception
- (Observable) mutation

SIDE-EFFECT

- File read call
- File write call
- Network call
- I/O call
- Throwing exception
- (Observable) mutation
- ...

SIDE-EFFECT

- File read call
- File write call
- Network call
- I/O call
- Throwing exception
- (Observable) mutation
- ...

Every state change outside of function

```
def f(x: float) -> float
```

EFFECT: GLOBAL MUTATION

```
prices                                          # Global variable

def minPrice()
   prices.sort()                                # Mutation
   return prices[0]
```

EFFECT: GLOBAL MUTATION

```
prices                                          # Global variable

def minPrice()
   prices.sort()                                # Mutation
   return prices[0]
```

```
prices = [5,2,7]                                # Init variable
# assert prices[0] == 5
```

EFFECT: GLOBAL MUTATION

```
prices                                          # Global variable

def minPrice()
   prices.sort()                                # Mutation
   return prices[0]
```

```
prices = [5,2,7]                                # Init variable
# assert prices[0] == 5
```

```
x = minPrice()                                  # Hidden mutation
# assert prices[0] == 5                         # No
```

EFFECT: CLASS MUTATION

```
class MyClass
  prices                                    # Class variable

  def minPrice()
    this.prices.sort()                      # Hidden mutation
    return this.prices[0]
```

EFFECT: CLASS MUTATION

```
class MyClass
  prices                                          # Class variable

  def minPrice()
    this.prices.sort()                            # Hidden mutation
    return this.prices[0]
```

```
object = MyClass([5,2,7])                         # Init variable
# assert object.prices[0] == 5
```

EFFECT: CLASS MUTATION

```
class MyClass
  prices                                          # Class variable

  def minPrice()
    this.prices.sort()                            # Hidden mutation
    return this.prices[0]
```

```
object = MyClass([5,2,7])                         # Init variable
# assert object.prices[0] == 5
```

```
x = object.minPrice()                             # Hidden mutation
# assert object.prices[0] == 5                    # No
```

OO IMPURE STYLE

```
class MyClass
  state=ComplexState()                          # 100s variables

  def do() -> None                              # Announced
        mutation
    state.change()                              # Mutation
    state.change_more()                         # Mutation
    this._do()                                  # Mutation

  def _do()                                     # Impure
    state.change_even_more()                    # Mutation
```

OO IMPURE STYLE

```
class MyClass
  state=ComplexState()                          # 100s variables

  def do() -> None                              # Announced
        mutation
    state.change()                              # Mutation
    state.change_more()                         # Mutation
    this._do()                                  # Mutation

  def _do()                                     # Impure
    state.change_even_more()                    # Mutation
```

```
object = MyClass()                              # Init variable
# assert object.state == ComplexState()
```

OO IMPURE STYLE

```
class MyClass
  state=ComplexState()                          # 100s variables

  def do() -> None                              # Announced
        mutation
    state.change()                              # Mutation
    state.change_more()                         # Mutation
    this._do()                                  # Mutation

  def _do()                                     # Impure
    state.change_even_more()                    # Mutation
```

```
object = MyClass()                              # Init variable
# assert object.state == ComplexState()
```

```
object.do()                                   # Announced
        mutation
# assert object.state == no clue
```

OO FP-ISH STYLE

```
class MyClass

  def do(state)                                  # Pure
    x = change(state)                            # No mutation
    y = change_more(x)                           # No mutation
    z = this._do(y)                              # No mutation
    return z

  def _do(y)                                     # Impure
    return change_even_more(y)                   # Mutation
```

OO FP-ISH STYLE

```
class MyClass

  def do(state)                                    # Pure
    x = change(state)                              # No mutation
    y = change_more(x)                             # No mutation
    z = this._do(y)                                # No mutation
    return z

  def _do(y)                                       # Impure
    return change_even_more(y)                     # Mutation
```

```
state = ComplexState()                             # Init variable
# assert state == ComplexState()
```

OO FP-ISH STYLE

```
class MyClass

  def do(state)                                    # Pure
    x = change(state)                              # No mutation
    y = change_more(x)                             # No mutation
    z = this._do(y)                                # No mutation
    return z

  def _do(y)                                       # Impure
    return change_even_more(y)                     # Mutation
```

```
state = ComplexState()                             # Init variable
# assert state == ComplexState()
```

```
x = MyClass.do(state)                              # No mutation
# assert state == ComplexState()
```

WHY IMMUTABLE CLASSES

Not ok:

```
def do(state):
  # ... change state

x = do(state)                                    # state mutated
```

WHY IMMUTABLE CLASSES

Not ok:

```
def do(state):
  # ... change state

x = do(state)                                    # state mutated
```

So why should this?

```
x = state.do()                                   # state mutated
```

WHY IMMUTABLE CLASSES

Not ok:

```
def do(state):
  # ... change state

x = do(state)                                    # state mutated
```

So why should this?

```
x = state.do()                                   # state mutated
```

```
class State:

    def do(self):                                # self == state
      # ... change self
```

PURE FUNCTIONS

Referential transparency:

```
def main():
  y = compute(5,2) + 7
  z = compute(5,2) + 1
```

PURE FUNCTIONS

Referential transparency:

```
def main():
  y = compute(5,2) + 7
  z = compute(5,2) + 1
```

=

```
def main():
  x = compute(5,2)                    # Factored out
  y = x             + 7
  z = x             + 1
```

PURE FUNCTIONS

Referential transparency:

```
def main():
  y = compute(5,2) + 7
  z = compute(5,2) + 1
```

=

```
def main():
  x = compute(5,2)                    # Factored out
  y = x             + 7
  z = x             + 1
```

Fearless refactoring, also by compiler

IMPURE FUNCTIONS

No referential transparency:

```
def main():                                    # 2 side-effects
  y = fetchNumber() + 7
  z = fetchNumber() + 1
```

IMPURE FUNCTIONS

No referential transparency:

```
def main():                                    # 2 side-effects
  y = fetchNumber() + 7
  z = fetchNumber() + 1
```

!=

```
def main():                                    # 1 side-effect
  x = fetchNumber()                            # Factored out
  y = x             + 7
  z = x             + 1
```

IMPURE FUNCTIONS

No referential transparency:

```
def main():                                    # 2 side-effects
  y = fetchNumber() + 7
  z = fetchNumber() + 1
```

!=

```
def main():                                    # 1 side-effect
  x = fetchNumber()                            # Factored out
  y = x             + 7
  z = x             + 1
```

Careful refactoring, often manual

FURTHER TOPICS

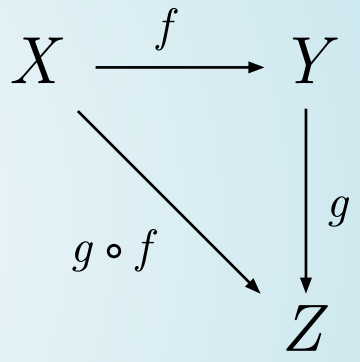
Try Haskell

- All pure
- No mutation

Try Rust:

- Much pure
- No mutation by default

Category Theory



- Functor (map)
- Monad (flatMap)
- Lens
- ...

Thanks