Design principles in FP

Based on Alexander Granin “Functional Design and Architecture”

Outline

1. Software design
2. DSL and Interpreter. Basic
3. DSL and Interpreter. Details
4. DSL and Interpreter. Advanced

Software design

1. Software design
2. Object oriented design
3. Functional Declarative design

Software design

Idea:

- Develop extendable software that match requirements and goals

Software design

Idea:

- Develop extendable software that match requirements and goals

Requirements:

- Functional
- Nonfunctional

Software design

Idea:

- Develop extendable software that match requirements and goals

Requirements:

- Functional
- Nonfunctional

Goals:

- Develop and deploy product
- Fit into the budget
- Fit into deadlines
- Achieve minimal quality

Software design

Idea:

- Develop extendable software that match requirements and goals

Requirements:

- Functional
- Nonfunctional

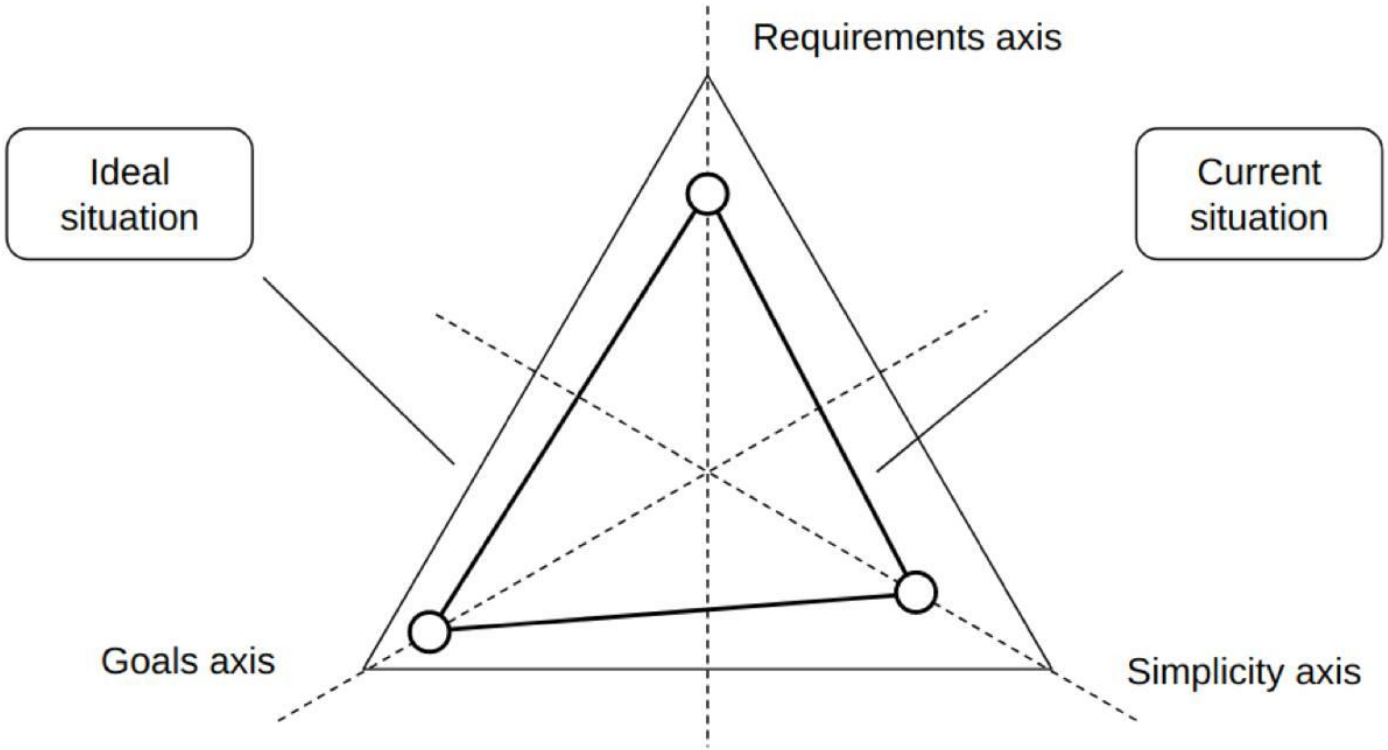
Goals:

- Develop and deploy product
- Fit into the budget
- Fit into deadlines
- Achieve minimal quality

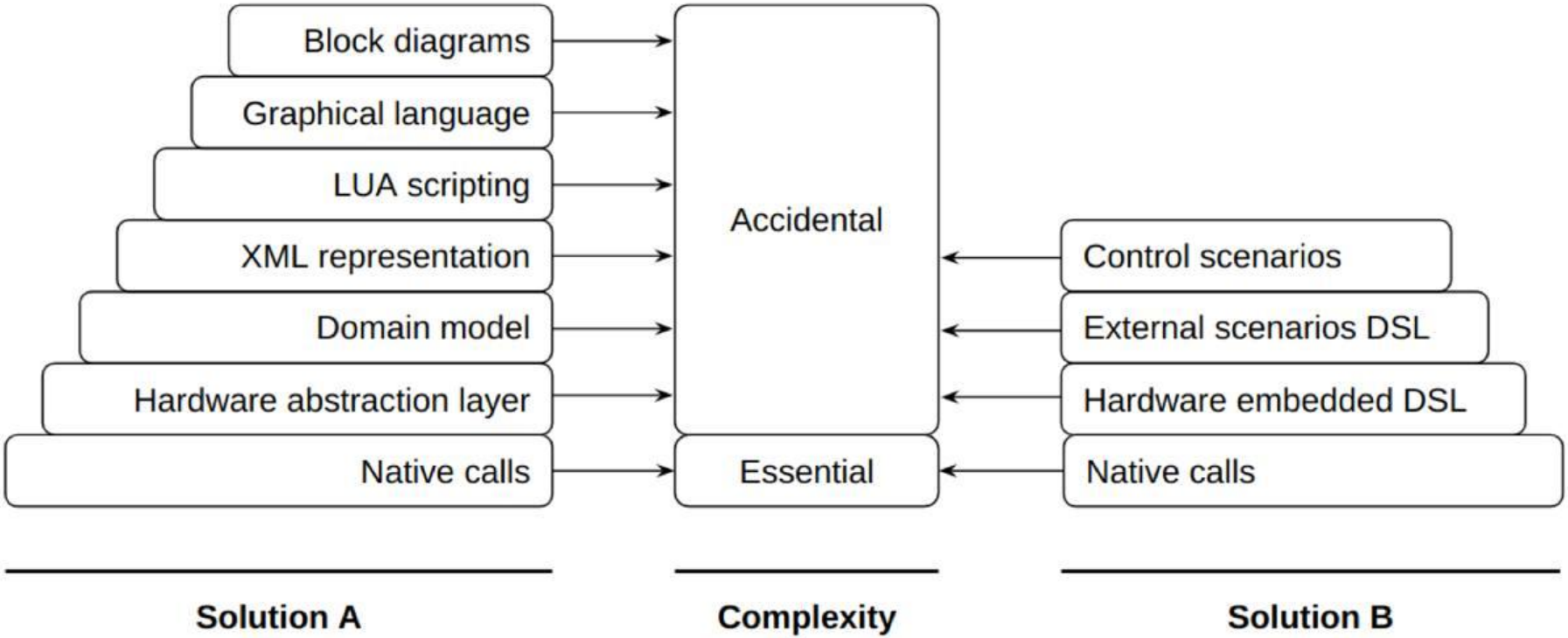
Extendable:

- Simply code is easier to maintain

Compromise: Goals, Simplicity and Requirements



Code complexity



Coupling

Issue explanation

Codependent:

There's a high frequency of `Navigation.tsx` and `Routes.ts` changing together. Out of **23** total changes for `Navigation.tsx` and **19** total changes for `Routes.ts`, **16** of them happened together.

Extremely frequent:

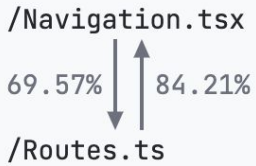
this pair changes more than over 99.93% of other pairs.

Close files:

files are located almost in the same directories.

Dependency schema

`/apps/playground/src/routing`



When modifying file A or B, check another file for necessary updates.

Consider refactoring, if possible, in order to reduce coupling and increase future maintainability.

Object oriented design

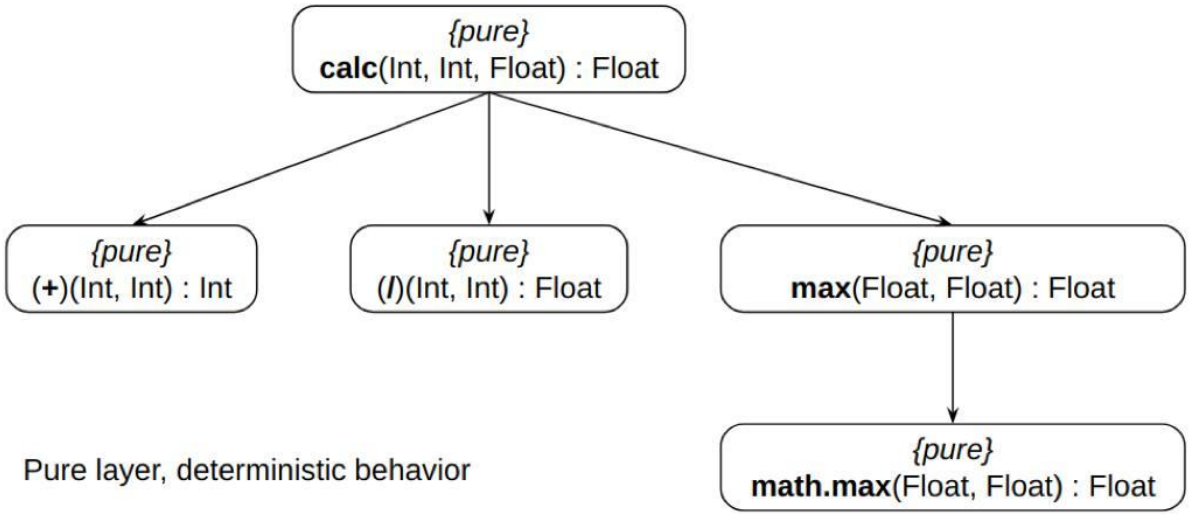
- Impure and imperative approach with rules called “Object Oriented Principles”
- Patterns we are familiar with are applied to OOD

- SOLID as example of Object Oriented Principles:
  - Single Responsibility Principle [single responsibility of object]
  - Open/Closed Principle [open for extension, closed for modification]
  - Liskov Substitution Principle [inherited objects should not break parents logic]
  - Interface Segregation Principle [interfaces should be small & atomic]
  - Dependency Inversion Principle [should depends on abstraction]

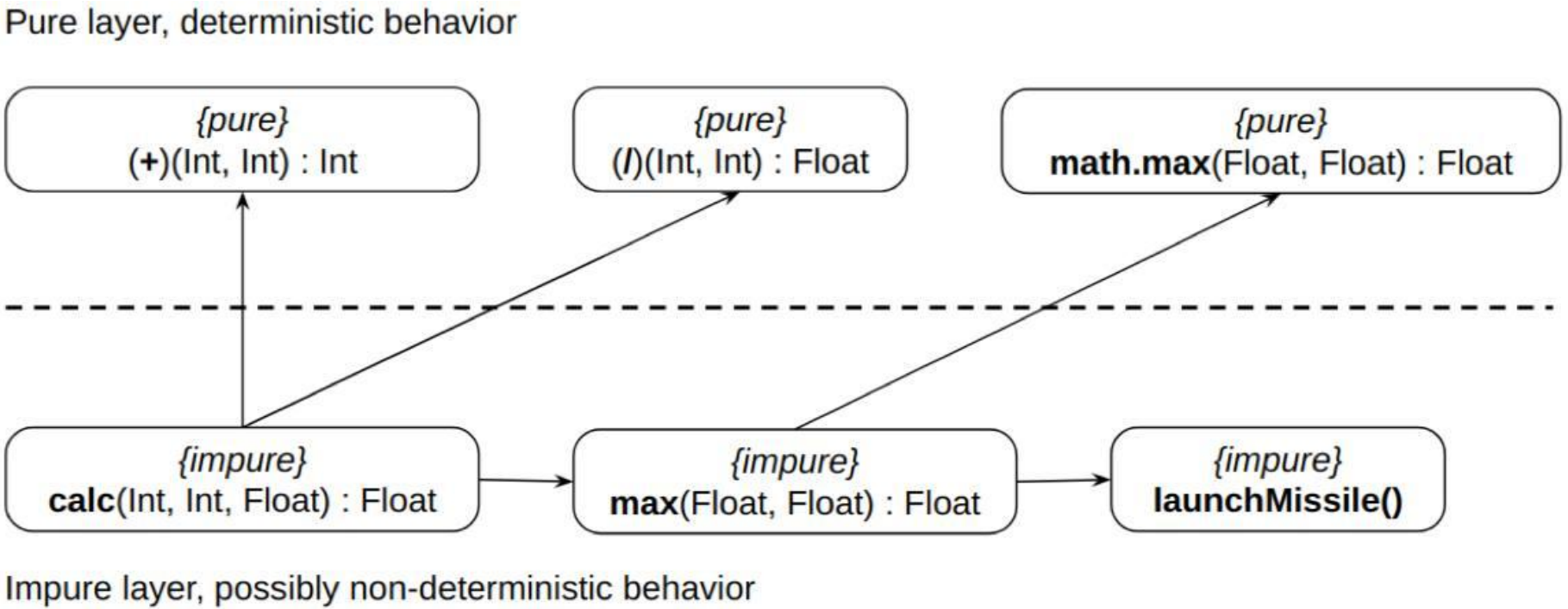
Functional declarative design

Is a part of Functional Programming with the concepts of:

- Assignments are bad
- Pure functions and pyramid of them
- Pure and impure layers



Pure and impure layers

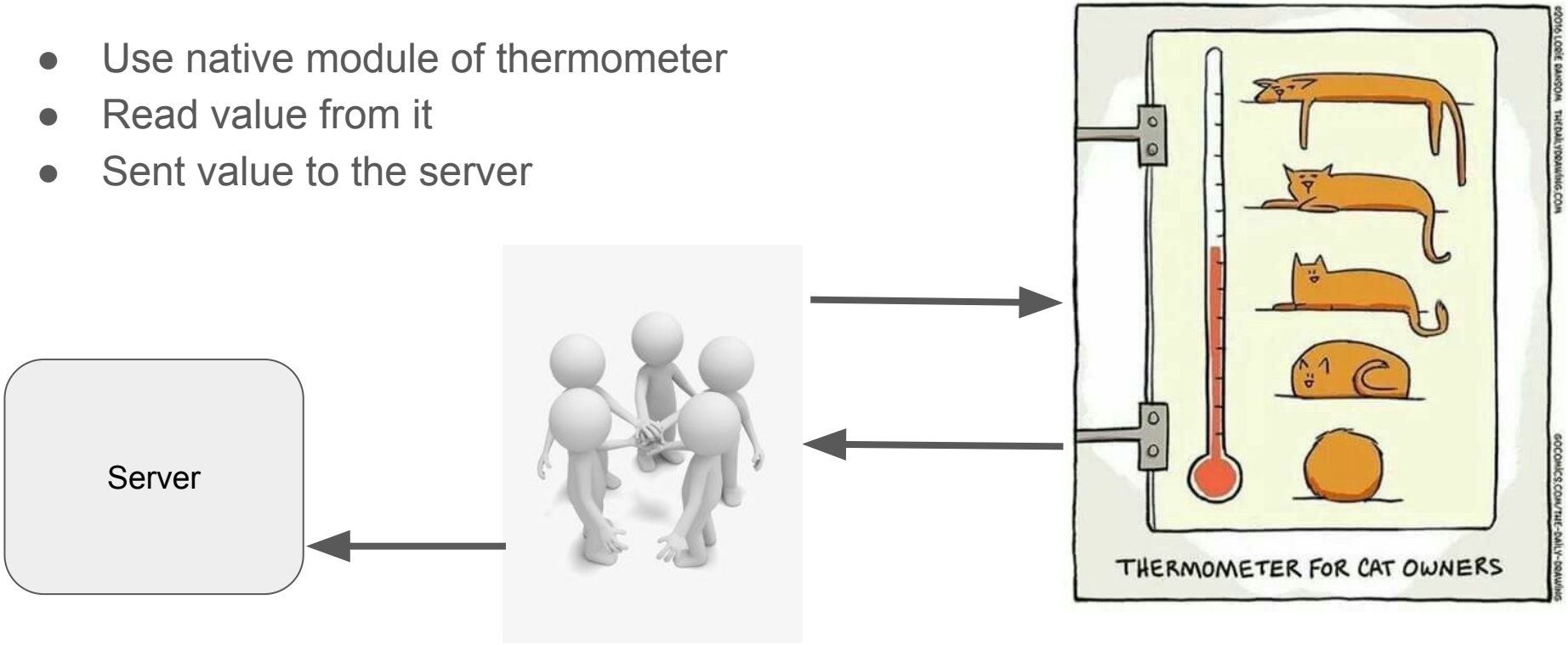


DSL and Interpreter. Basic

1. Basic approach
2. Common idea

Example task

- Use native module of thermometer
- Read value from it
- Sent value to the server



Functional code v1

```
import qualified Native.Core.Thermometer as T
import qualified ServerContext.Connection as C

readThermometer :: String -> IO T.Temperature
readThermometer name = T.read name

sendTemperature :: String -> Float -> IO ()
sendTemperature name t = C.send "temperature" name t

readTemperature :: IO Float
readTemperature = do
  t1 <- readThermometer "T-201A"
  return $ case t1 of
    T.Kelvin  v -> 273.15 – v
    T.Celsius v -> v

readAndSend :: IO ()
readAndSend = do
  t1 <- readTemperature
  let t2 = t1 - 12.5
  sendTemperature "T-201A" t2
```

Functional code v1

```
import qualified Native.Core.Thermometer as T
import qualified ServerContext.Connection as C

readThermometer :: String -> IO T.Temperature
readThermometer name = T.read name

sendTemperature :: String -> Float -> IO ()
sendTemperature name t = C.send "temperature" name t

readTemperature :: IO Float
readTemperature = do
  t1 <- readThermometer "T-201A"
  return $ case t1 of
    T.Kelvin  v -> 273.15 - v
    T.Celsius v -> v

readAndSend :: IO ()
readAndSend = do
  t1 <- readTemperature
  let t2 = t1 - 12.5
  sendTemperature "T-201A" t2
```

Impure call to thermometer, use of native library

Functional code v1

```
import qualified Native.Core.Thermometer as T
import qualified ServerContext.Connection as C

readThermometer :: String -> IO T.Temperature
readThermometer name = T.read name

sendTemperature :: String -> Float -> IO ()
sendTemperature name t = C.send "temperature" name t

readTemperature :: IO Float
readTemperature = do
  t1 <- readThermometer "T-201A"
  return $ case t1 of
    T.Kelvin  v -> 273.15 - v
    T.Celsius v -> v

readAndSend :: IO ()
readAndSend = do
  t1 <- readTemperature
  let t2 = t1 - 12.5
  sendTemperature "T-201A" t2
```

Impure call to server, use of native library

Functional code v1

```
import qualified Native.Core.Thermometer as T
import qualified ServerContext.Connection as C

readThermometer :: String -> IO T.Temperature
readThermometer name = T.read name

sendTemperature :: String -> Float -> IO ()
sendTemperature name t = C.send "temperature" name t

readTemperature :: IO Float
readTemperature = do
  t1 <- readThermometer "T-201A"
  return $ case t1 of
    T.Kelvin  v -> 273.15 - v
    T.Celsius v -> v

readAndSend :: IO ()
readAndSend = do
  t1 <- readTemperature
  let t2 = t1 - 12.5
  sendTemperature "T-201A" t2
```

Impure action, depending on native library call

Functional code v1

```
import qualified Native.Core.Thermometer as T
import qualified ServerContext.Connection as C

readThermometer :: String -> IO T.Temperature
readThermometer name = T.read name

sendTemperature :: String -> Float -> IO ()
sendTemperature name t = C.send "temperature" name t

readTemperature :: IO Float
readTemperature = do
  t1 <- readThermometer "T-201A"
  return $ case t1 of
    T.Kelvin  v -> 273.15 - v
    T.Celsius v -> v

readAndSend :: IO ()
readAndSend = do
  t1 <- readTemperature
  let t2 = t1 - 12.5
  sendTemperature "T-201A" t2
```

Possible, highly coupled function, because it has a lot of dependencies

Functional code v1. Conclusion

- Has only impure level
- Cannot be tested
- Straightforward
- Possibly, high coupled

Functional code v2

```
type DeviceName = String
type DataType = String
type TransformF a = Float -> ActionDsl a

data ActionDsl a
  = ReadDevice DeviceName (a -> ActionDsl a)
  | Transform (a -> Float) a (TransformF a)
  | Correct (Float -> Float) Float (TransformF a)
  | Send DataType DeviceName Float

transform (T.Kelvin  v) = v - 273.15
transform (T.Celsius v) = v
correction v = v - 12.5
therm = Thermometer "T-800"

scenario :: ActionDsl T.Temperature
scenario =
  ReadDevice therm (\v ->
  Transform transform v (\v1 ->
  Correct correction v1 (\v2 ->
  Send temp therm v2)))

interpret :: ActionDsl T.Temperature -> IO ()
interpret (ReadDevice n a) = do
  v <- T.read n
  interpret (a v)
interpret (Transform f v a) = interpret (a (f v))
interpret (Correct f v a)   = interpret (a (f v))
interpret (Send t n v)      = C.send t n v

readAndSend :: IO ()
readAndSend = interpret scenario
```

```
type DeviceName = String
type DataType = String
type TransformF a = Float -> ActionDsl a

data ActionDsl a
  = ReadDevice DeviceName (a -> ActionDsl a)
  | Transform (a -> Float) a (TransformF a)
  | Correct (Float -> Float) Float (TransformF a)
  | Send DataType DeviceName Float

transform (T.Kelvin  v) = v - 273.15
transform (T.Celsius v) = v
correction v = v - 12.5
therm = Thermometer "T-800"

scenario :: ActionDsl T.Temperature
scenario =
  ReadDevice therm (\v ->
  Transform transform v (\v1 ->
  Correct correction v1 (\v2 ->
  Send temp therm v2)))

interpret :: ActionDsl T.Temperature -> IO ()
interpret (ReadDevice n a) = do
  v <- T.read n
  interpret (a v)
interpret (Transform f v a) = interpret (a (f v))
interpret (Correct f v a)   = interpret (a (f v))
interpret (Send t n v)      = C.send t n v

readAndSend :: IO ()
readAndSend = interpret scenario
```

Functional code v2

Definition of actions, pure

Functional code v2

```
type DeviceName = String
type DataType = String
type TransformF a = Float -> ActionDsl a

data ActionDsl a
  = ReadDevice DeviceName (a -> ActionDsl a)
  | Transform (a -> Float) a (TransformF a)
  | Correct (Float -> Float) Float (TransformF a)
  | Send DataType DeviceName Float

transform (T.Kelvin  v) = v - 273.15
transform (T.Celsius v) = v
correction v = v - 12.5
therm = Thermometer "T-800"

scenario :: ActionDsl T.Temperature
scenario =
  ReadDevice therm (\v ->
  Transform transform v (\v1 ->
  Correct correction v1 (\v2 ->
  Send temp therm v2)))

interpret :: ActionDsl T.Temperature -> IO ()
interpret (ReadDevice n a) = do
  v <- T.read n
  interpret (a v)
interpret (Transform f v a) = interpret (a (f v))
interpret (Correct f v a)   = interpret (a (f v))
interpret (Send t n v)      = C.send t n v

readAndSend :: IO ()
readAndSend = interpret scenario
```

Pure functions

Functional code v2

```
type DeviceName = String
type DataType = String
type TransformF a = Float -> ActionDsl a

data ActionDsl a
  = ReadDevice DeviceName (a -> ActionDsl a)
  | Transform (a -> Float) a (TransformF a)
  | Correct (Float -> Float) Float (TransformF a)
  | Send DataType DeviceName Float

transform (T.Kelvin  v) = v - 273.15
transform (T.Celsius v) = v
correction v = v - 12.5
therm = Thermometer "T-800"

scenario :: ActionDsl T.Temperature
scenario =
  ReadDevice therm (\v ->
  Transform transform v (\v1 ->
  Correct correction v1 (\v2 ->
  Send temp therm v2)))

interpret :: ActionDsl T.Temperature -> IO ()
interpret (ReadDevice n a) = do
  v <- T.read n
  interpret (a v)
interpret (Transform f v a) = interpret (a (f v))
interpret (Correct f v a)   = interpret (a (f v))
interpret (Send t n v)      = C.send t n v

readAndSend :: IO ()
readAndSend = interpret scenario
```

Pure scenario, actions to be done

Functional code v2

```
type DeviceName = String
type DataType = String
type TransformF a = Float -> ActionDsl a

data ActionDsl a
  = ReadDevice DeviceName (a -> ActionDsl a)
  | Transform (a -> Float) a (TransformF a)
  | Correct (Float -> Float) Float (TransformF a)
  | Send DataType DeviceName Float

transform (T.Kelvin  v) = v - 273.15
transform (T.Celsius v) = v
correction v = v - 12.5
therm = Thermometer "T-800"

scenario :: ActionDsl T.Temperature
scenario =
  ReadDevice therm (\v ->
  Transform transform v (\v1 ->
  Correct correction v1 (\v2 ->
  Send temp therm v2)))

interpret :: ActionDsl T.Temperature -> IO ()
interpret (ReadDevice n a) = do
  v <- T.read n
  interpret (a v)
interpret (Transform f v a) = interpret (a (f v))
interpret (Correct f v a)   = interpret (a (f v))
interpret (Send t n v)      = C.send t n v

readAndSend :: IO ()
readAndSend = interpret scenario
```

Impure interpreter:
pure DSL -> impure world

Functional code v2

```
type DeviceName = String
type DataType = String
type TransformF a = Float -> ActionDsl a

data ActionDsl a
  = ReadDevice DeviceName (a -> ActionDsl a)
  | Transform (a -> Float) a (TransformF a)
  | Correct (Float -> Float) Float (TransformF a)
  | Send DataType DeviceName Float

transform (T.Kelvin  v) = v - 273.15
transform (T.Celsius v) = v
correction v = v - 12.5
therm = Thermometer "T-800"

scenario :: ActionDsl T.Temperature
scenario =
  ReadDevice therm (\v ->
  Transform transform v (\v1 ->
  Correct correction v1 (\v2 ->
  Send temp therm v2)))

interpret :: ActionDsl T.Temperature -> IO ()
interpret (ReadDevice n a) = do
  v <- T.read n
  interpret (a v)
interpret (Transform f v a) = interpret (a (f v))
interpret (Correct f v a)   = interpret (a (f v))
interpret (Send t n v)      = C.send t n v

readAndSend :: IO ()
readAndSend = interpret scenario
```

Entry point

Functional code v2

```
type DeviceName = String
type DataType = String
type TransformF a = Float -> ActionDsl a

data ActionDsl a
  = ReadDevice DeviceName (a -> ActionDsl a)
  | Transform (a -> Float) a (TransformF a)
  | Correct (Float -> Float) Float (TransformF a)
  | Send DataType DeviceName Float
```

Definition of actions, pure

```
transform (T.Kelvin  v) = v - 273.15
transform (T.Celsius v) = v
correction v = v - 12.5
therm = Thermometer "T-800"
```

Pure functions

```
scenario :: ActionDsl T.Temperature
scenario =
  ReadDevice therm (\v ->
  Transform transform v (\v1 ->
  Correct correction v1 (\v2 ->
  Send temp therm v2)))
```

Pure scenario, actions to be done

```
interpret :: ActionDsl T.Temperature -> IO ()
interpret (ReadDevice n a) = do
  v <- T.read n
  interpret (a v)
interpret (Transform f v a) = interpret (a (f v))
interpret (Correct f v a)   = interpret (a (f v))
interpret (Send t n v)      = C.send t n v
```

Impure interpreter:
pure DSL -> impure world

```
readAndSend :: IO ()
readAndSend = interpret scenario
```

Entry point

Functional code v2. Conclusion

- Has only pure and impure level
- Can be tested
- Has pure straightforward scenario
- Possibly, low coupled

DSL Approach

```
data Language = ImpureAction1
              | ImpureAction2
              | ImpureAction3

type Interpreter = Language -> IO ()

simpleInterpreter :: Interpreter
mockInterpreter :: Interpreter
whateverInterpreter :: Interpreter

simpleInterpreter ImpureAction1 = impureCall1
simpleInterpreter ImpureAction2 = impureCall2
simpleInterpreter ImpureAction3 = impureCall3

mockInterpreter ImpureAction1 = doNothing
mockInterpreter ImpureAction2 = doNothing
mockInterpreter ImpureAction3 = doNothing

-- Run language against the interpreter
run :: Language -> Interpreter -> IO ()
run script interpreter = interpreter script
```

DSL Approach

```
data Language = ImpureAction1
              | ImpureAction2
              | ImpureAction3

type Interpreter = Language -> IO ()

simpleInterpreter :: Interpreter
mockInterpreter :: Interpreter
whateverInterpreter :: Interpreter

simpleInterpreter ImpureAction1 = impureCall1
simpleInterpreter ImpureAction2 = impureCall2
simpleInterpreter ImpureAction3 = impureCall3

mockInterpreter ImpureAction1 = doNothing
mockInterpreter ImpureAction2 = doNothing
mockInterpreter ImpureAction3 = doNothing

-- Run language against the interpreter
run :: Language -> Interpreter -> IO ()
run script interpreter = interpreter script
```

Actions in Domain, impure by nature, pure by definition

DSL Approach

```
data Language = ImpureAction1
              | ImpureAction2
              | ImpureAction3

type Interpreter = Language -> IO ()

simpleInterpreter :: Interpreter
mockInterpreter :: Interpreter
whateverInterpreter :: Interpreter

simpleInterpreter ImpureAction1 = impureCall1
simpleInterpreter ImpureAction2 = impureCall2
simpleInterpreter ImpureAction3 = impureCall3

mockInterpreter ImpureAction1 = doNothing
mockInterpreter ImpureAction2 = doNothing
mockInterpreter ImpureAction3 = doNothing

-- Run language against the interpreter
run :: Language -> Interpreter -> IO ()
run script interpreter = interpreter script
```

Interpreter 1

DSL Approach

```
data Language = ImpureAction1
              | ImpureAction2
              | ImpureAction3

type Interpreter = Language -> IO ()

simpleInterpreter :: Interpreter
mockInterpreter :: Interpreter
whateverInterpreter :: Interpreter

simpleInterpreter ImpureAction1 = impureCall1
simpleInterpreter ImpureAction2 = impureCall2
simpleInterpreter ImpureAction3 = impureCall3

mockInterpreter ImpureAction1 = doNothing
mockInterpreter ImpureAction2 = doNothing
mockInterpreter ImpureAction3 = doNothing

-- Run language against the interpreter
run :: Language -> Interpreter -> IO ()
run script interpreter = interpreter script
```

Interpreter 2

DSL Approach

```
data Language = ImpureAction1
              | ImpureAction2
              | ImpureAction3

type Interpreter = Language -> IO ()

simpleInterpreter :: Interpreter
mockInterpreter :: Interpreter
whateverInterpreter :: Interpreter

simpleInterpreter ImpureAction1 = impureCall1
simpleInterpreter ImpureAction2 = impureCall2
simpleInterpreter ImpureAction3 = impureCall3

mockInterpreter ImpureAction1 = doNothing
mockInterpreter ImpureAction2 = doNothing
mockInterpreter ImpureAction3 = doNothing

-- Run language against the interpreter
run :: Language -> Interpreter -> IO ()
run script interpreter = interpreter script
```

Configuration of the execution

DSL Approach

```
data Language = ImpureAction1
              | ImpureAction2
              | ImpureAction3

type Interpreter = Language -> IO ()

simpleInterpreter :: Interpreter
mockInterpreter :: Interpreter
whateverInterpreter :: Interpreter

simpleInterpreter ImpureAction1 = impureCall1
simpleInterpreter ImpureAction2 = impureCall2
simpleInterpreter ImpureAction3 = impureCall3

mockInterpreter ImpureAction1 = doNothing
mockInterpreter ImpureAction2 = doNothing
mockInterpreter ImpureAction3 = doNothing

-- Run language against the interpreter
run :: Language -> Interpreter -> IO ()
run script interpreter = interpreter script
```

Actions in Domain

Interpreter 1

Interpreter 2

Configuration of the execution

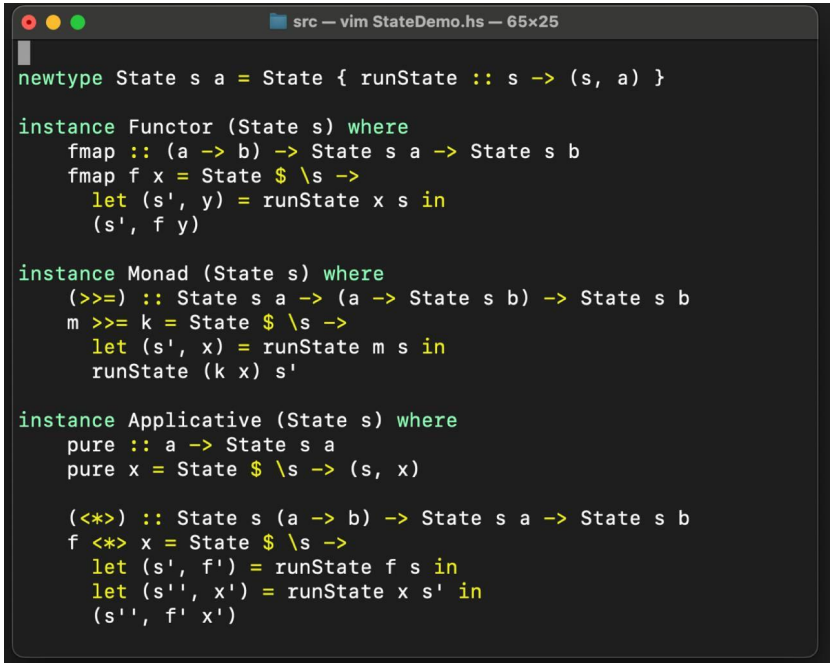
Demo 1

DSL and Interpreter. Details

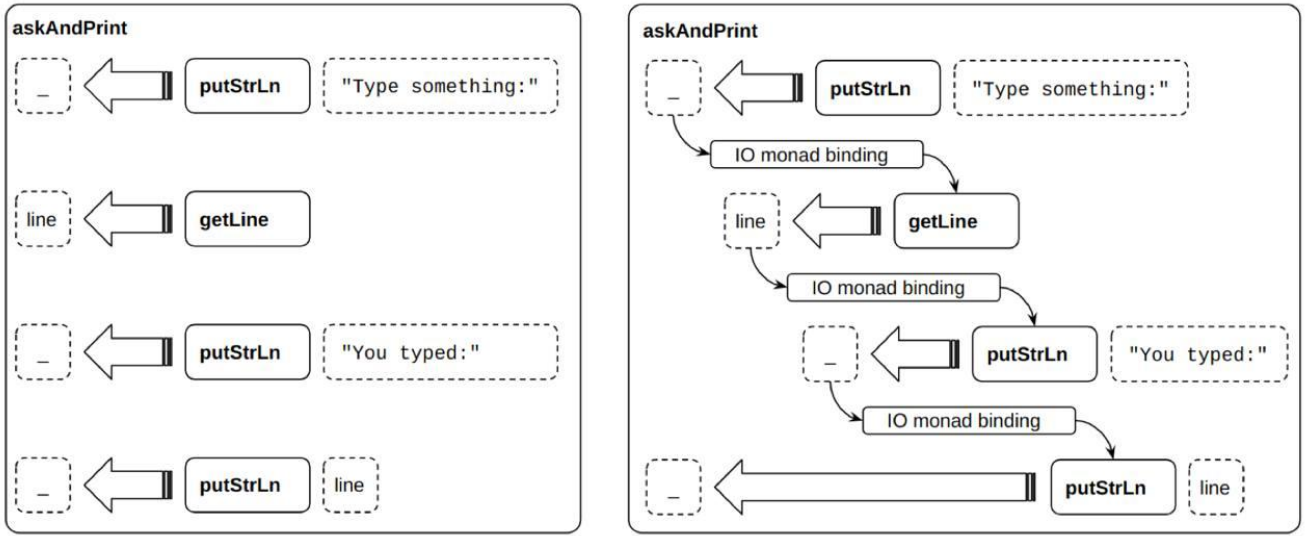
1. Monad / Monad stack as subsystem
2. Encapsulation
3. Interpreter and Translator

Monad as subsystem

- Running behavior:
  - To start calculation in the monad and get the result, we evaluate a function runX
- Treat monad as subsystem with effect:
  - Monadic calculations can be complex
  - Can operate with effects
  - Can store data, load data
  - Can combine formula evaluations
- Combine submodules with Monad Stack



Monad as subsystem

- 

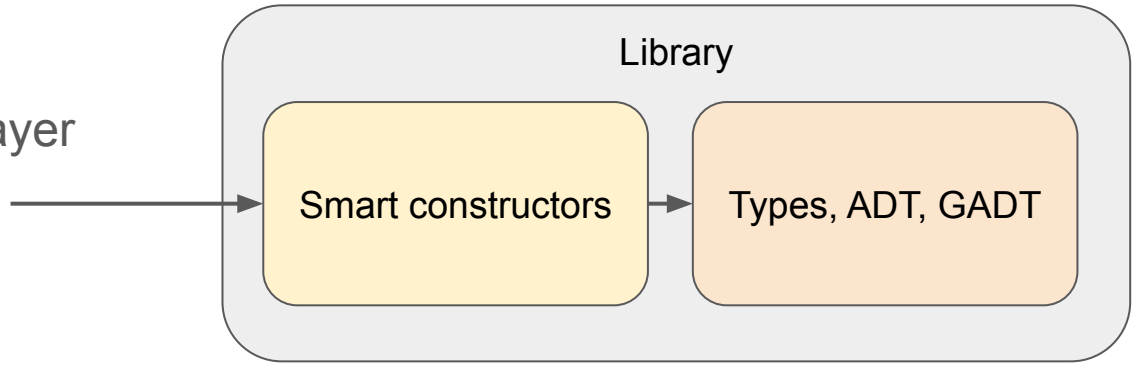
Encapsulation. Smart constructors

- Smart constructors is a function that **constructs a value of some type** without revealing the actual nature of the type
- They may even transform arguments to construct more complex value
- Smart constructors is an encapsulation way in FDD

Encapsulation. Smart constructors

- Smart constructors is a function that **constructs a value of some type** without revealing the actual nature of the type
- They may even transform arguments to construct more complex value
- Smart constructors is an encapsulation way in FDD

- One more abstraction layer



Encapsulation. Smart constructors

- Do you know any examples of smart constructors in Haskell?

Encapsulation. Smart constructors

- Do you know any examples of smart constructors in Haskell?

```
data Map k a = Bin Size k a (Map k a) (Map k a)
             | Tip
```

```
null :: Map k a -> Bool
size :: Map k a -> Int
lookup :: Ord k => k -> Map k a -> Maybe a
```

Encapsulation. Smart constructors

- Smart constructors allows us to switch between implementations of Map.Strict and Map.Lazy without refactoring the code

```
data Map k a = Bin Size k a (Map k a) (Map k a)
             | Tip
```

```
null :: Map k a -> Bool
size :: Map k a -> Int
lookup :: Ord k => k -> Map k a -> Maybe a
```

Encapsulation. Modules

- Module are the main tool of logical organization of the code
- Modules can be arranged hierarchically
- In Haskell, modules provide possibilities to export selectively

Encapsulation. Modules

- This is a SCADA software for spaceship control
- https://github.com/granina s/Andromeda

```
src\
    Andromeda
    Andromeda\
        Assets
        Calculations
        Common
        Hardware
        LogicControl
        Simulator
        Assets\
            Hardware\
                Components
        Calculations\
            Math\
            Physics\
        Common\
            Value
        Hardware\
            HDL
            HNDL
            Device
            Runtime
        LogicControl\
            Language
        Simulator\
            Language
            Runtime
```

Encapsulation. Modules

- This is a SCADA software for spaceship control
- https://github.com/granina s/Andromeda

```
src\
    Andromeda
    Andromeda\
        Assets
        Calculations
        Common
        Hardware
        LogicControl
        Simulator
        Assets\
            Hardware\
                Components
        Calculations\
            Math\
            Physics\
        Common\
            Value
        Hardware\
            HDL
            HNDL
            Device
            Runtime
        LogicControl\
            Language
        Simulator\
            Language
            Runtime
```

The module that exports the whole project as library

Encapsulation. Modules

- This is a SCADA software for spaceship control
- https://github.com/granina s/Andromeda

```
src\
    Andromeda
    Andromeda\
        Assets
        Calculations
        Common
        Hardware
        LogicControl
        Simulator
        Assets\
            Hardware\
                Components
        Calculations\
            Math\
            Physics\
        Common\
            Value
        Hardware\
            HDL
            HNDL
            Device
            Runtime
        LogicControl\
            Language
        Simulator\
            Language
            Runtime
```

] **Root of the project**

Encapsulation. Modules

- This is a SCADA software for spaceship control
- https://github.com/granin as/Andromeda

```
src\
    Andromeda
    Andromeda\
        Assets
        Calculations
        Common
        Hardware
        LogicControl
        Simulator
        Assets\
            Hardware\
                Components
        Calculations\
            Math\
            Physics\
        Common\
            Value
        Hardware\
            HDL
            HNDL
            Device
            Runtime
        LogicControl\
            Language
        Simulator\
            Language
            Runtime
```

Top modules; external code (such as test suites) should depend on these modules, not on the internal ones

Encapsulation. Modules

- This is a SCADA software for spaceship control
- https://github.com/granina s/Andromeda

```
src\
    Andromeda
    Andromeda\
        Assets
        Calculations
        Common
        Hardware
        LogicControl
        Simulator
        Assets\
            Hardware\
                Components
        Calculations\
            Math\
            Physics\
        Common\
            Value
        Hardware\
            HDL
            HNDL
            Device
            Runtime
        LogicControl\
            Language
        Simulator\
            Language
            Runtime
```

Predefined data such as default values, definitions, scenarios / scripts

Encapsulation. Modules

- This is a SCADA software for spaceship control
- https://github.com/granina s/Andromeda

```
src\
    Andromeda
    Andromeda\
        Assets
        Calculations
        Common
        Hardware
        LogicControl
        Simulator
        Assets\
            Hardware\
                Components
        Calculations\
            Math\
            Physics\
        Common\
            Value
        Hardware\
            HDL
            HNDL
            Device
            Runtime
        LogicControl\
            Language
        Simulator\
            Language
            Runtime
```

Separate library for math, physics

Encapsulation. Modules

- This is a SCADA software for spaceship control
- https://github.com/granina s/Andromeda

```
src\
    Andromeda
    Andromeda\
        Assets
        Calculations
        Common
        Hardware
        LogicControl
        Simulator
        Assets\
            Hardware\
                Components
        Calculations\
            Math\
            Physics\
        Common\
            Value
        Hardware\
            HDL
            HNDL
            Device
            Runtime
        LogicControl\
            Language
        Simulator\
            Language
            Runtime
```

Description languages with types, ADT and GADT

Encapsulation. Modules

- This is a SCADA software for spaceship control
- https://github.com/granina s/Andromeda

```
src\
    Andromeda
    Andromeda\
        Assets
        Calculations
        Common
        Hardware
        LogicControl
        Simulator
        Assets\
            Hardware\
                Components
        Calculations\
            Math\
            Physics\
        Common\
            Value
        Hardware\
            HDL
            HNDL
            Device
            Runtime
        LogicControl\
            Language
        Simulator\
            Language
            Runtime
```

Logic Control DLS (eDSL)

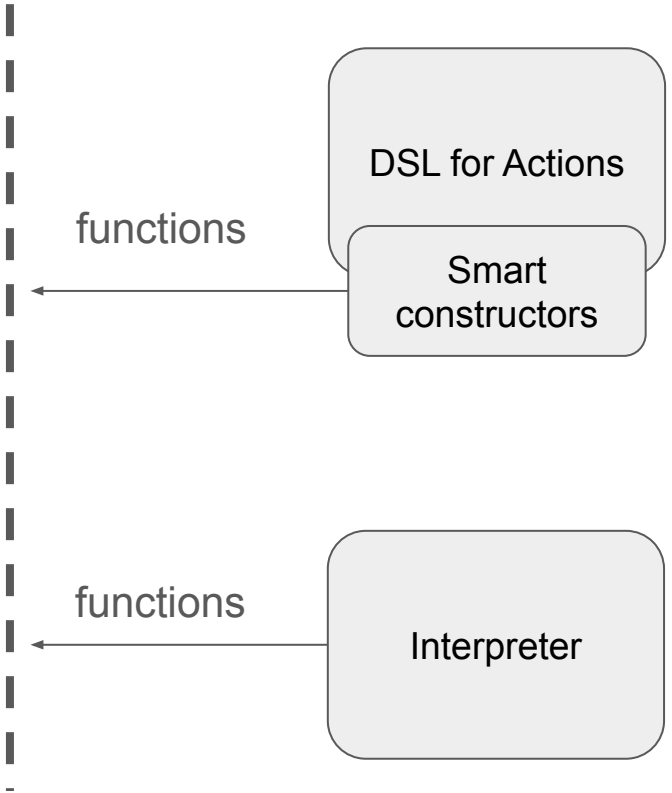
Encapsulation. Modules

- This is a SCADA software for spaceship control
- https://github.com/granina s/Andromeda

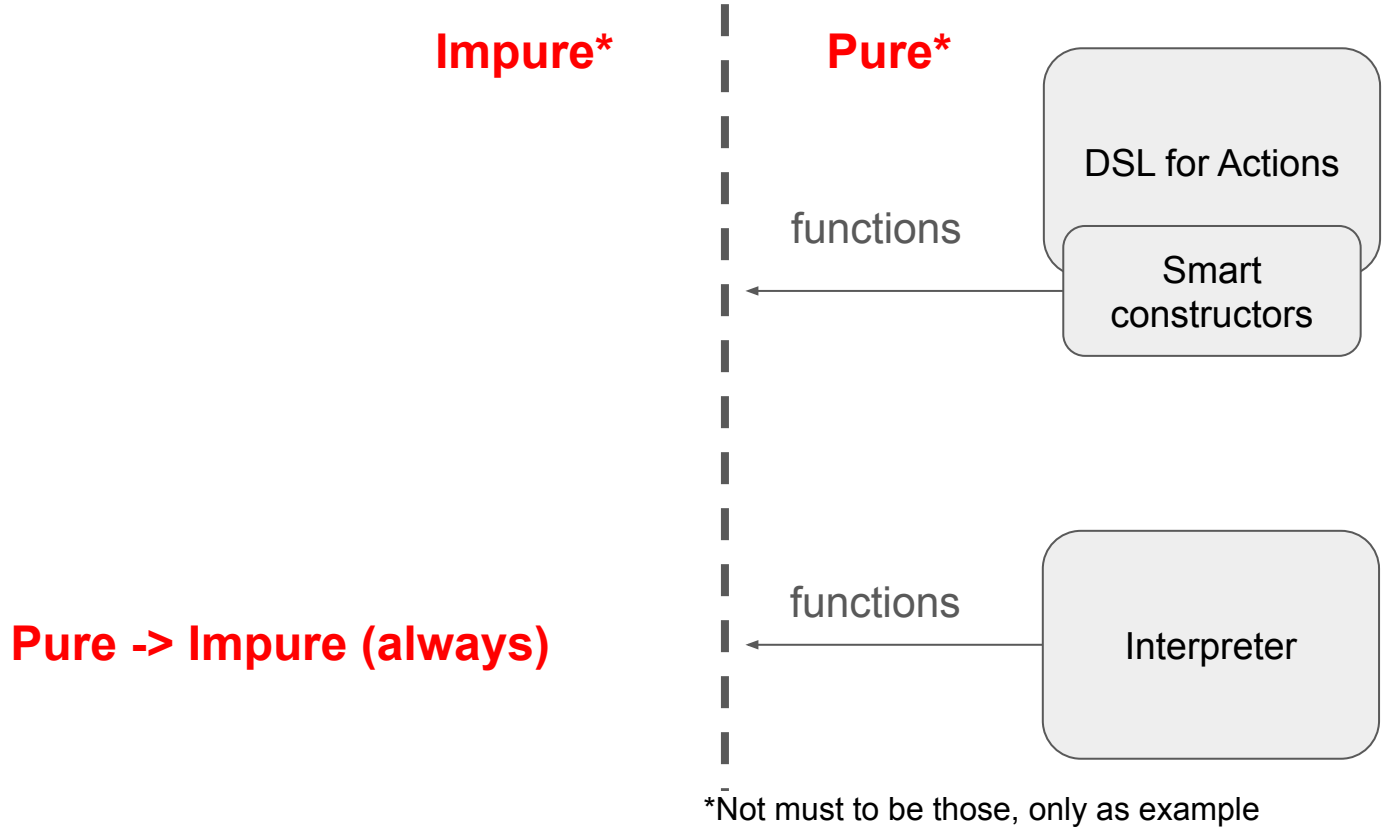
```
src\
    Andromeda
    Andromeda\
        Assets
        Calculations
        Common
        Hardware
        LogicControl
        Simulator
        Assets\
            Hardware\
                Components
        Calculations\
            Math\
            Physics\
        Common\
            Value
        Hardware\
            HDL
            HNDL
            Device
            Runtime
        LogicControl\
            Language
        Simulator\
            Language
            Runtime
```

Simulator subsystem, in future – separate project

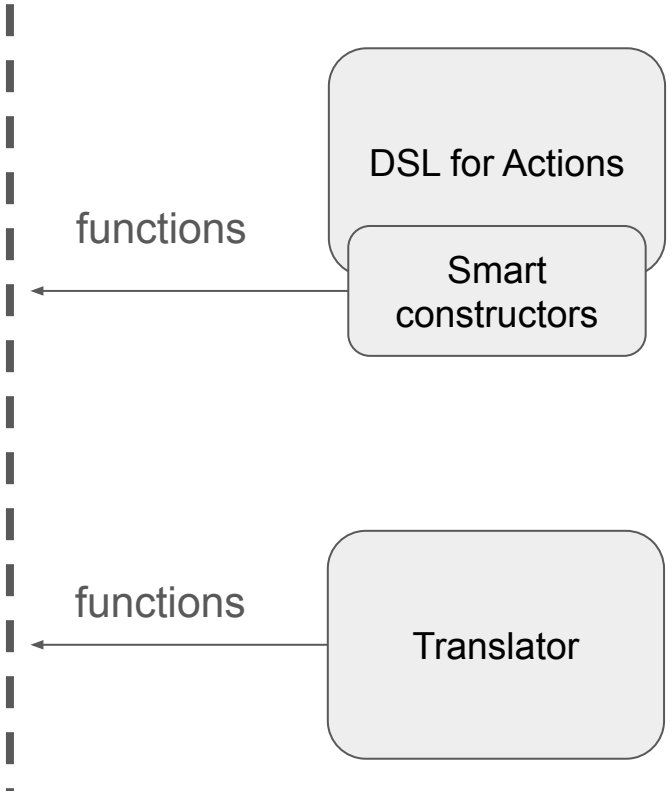
Interpreter and translator. Interpreter



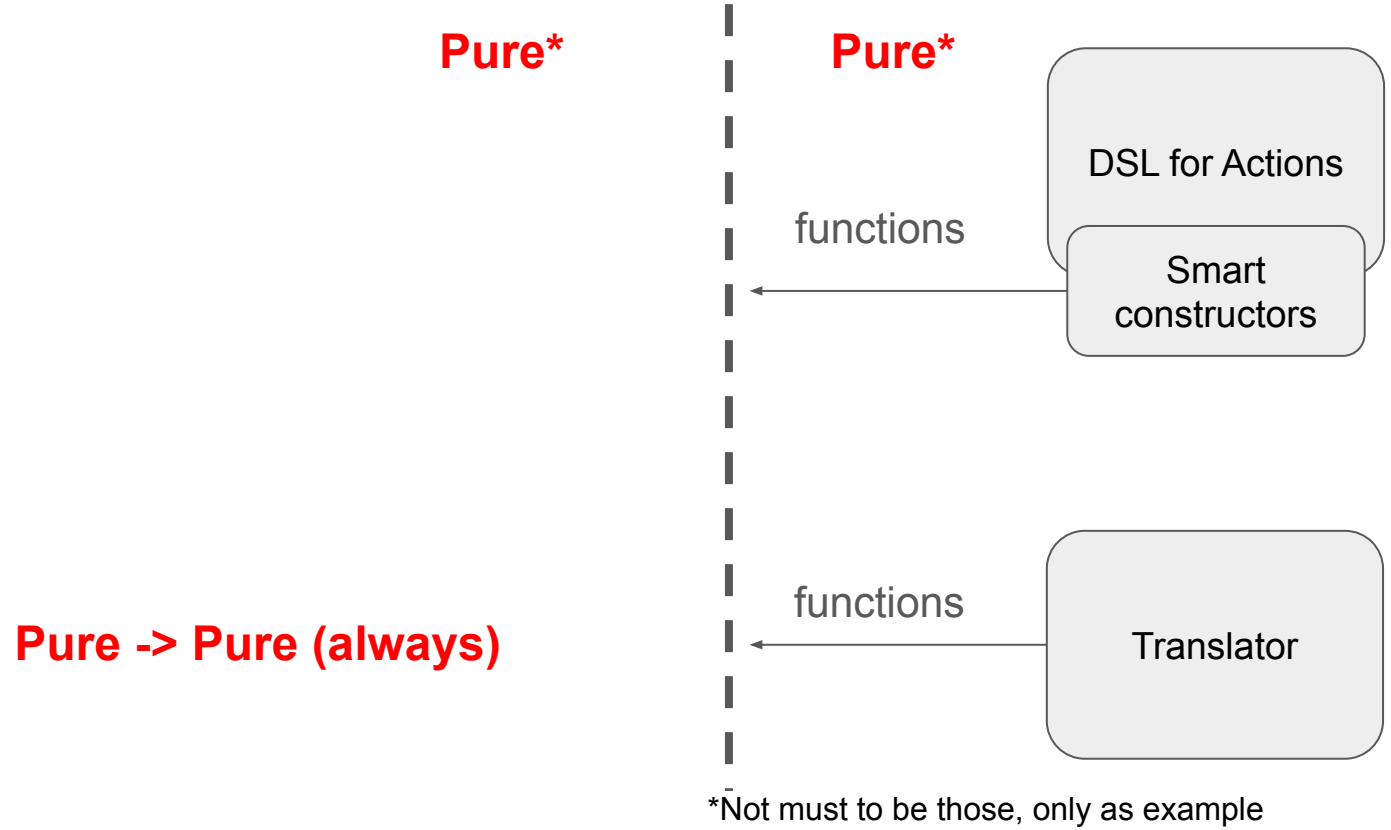
Interpreter and translator. Interpreter



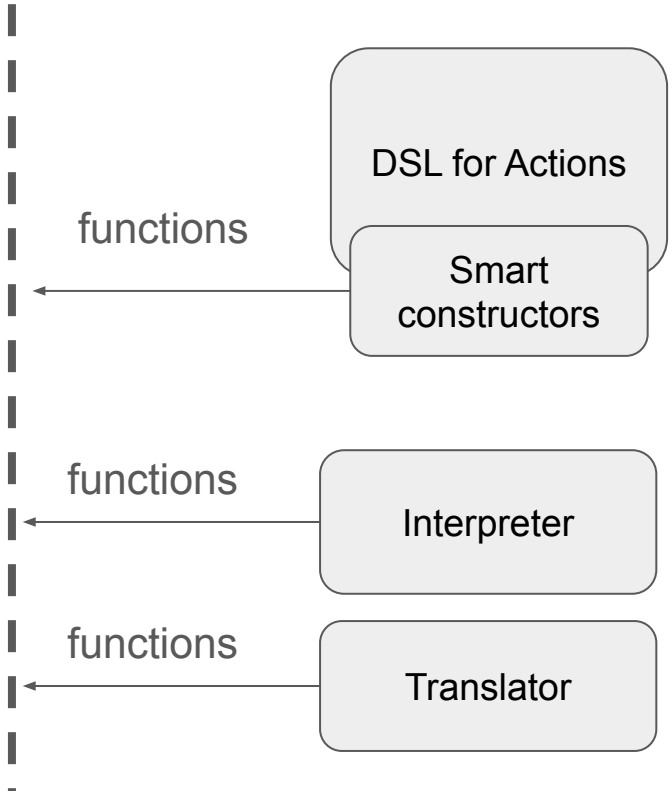
Interpreter and translator. Translator



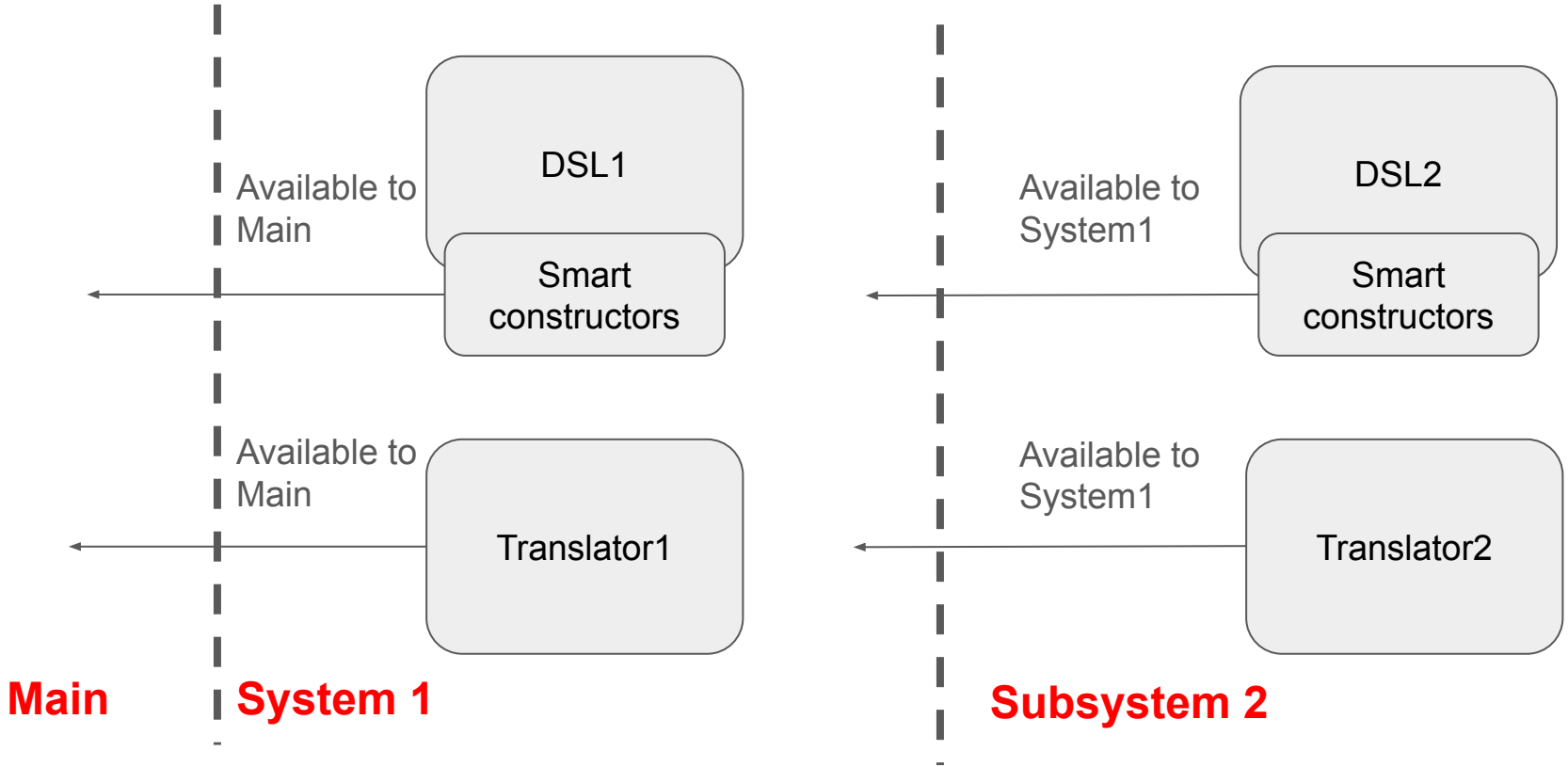
Interpreter and translator. Translator



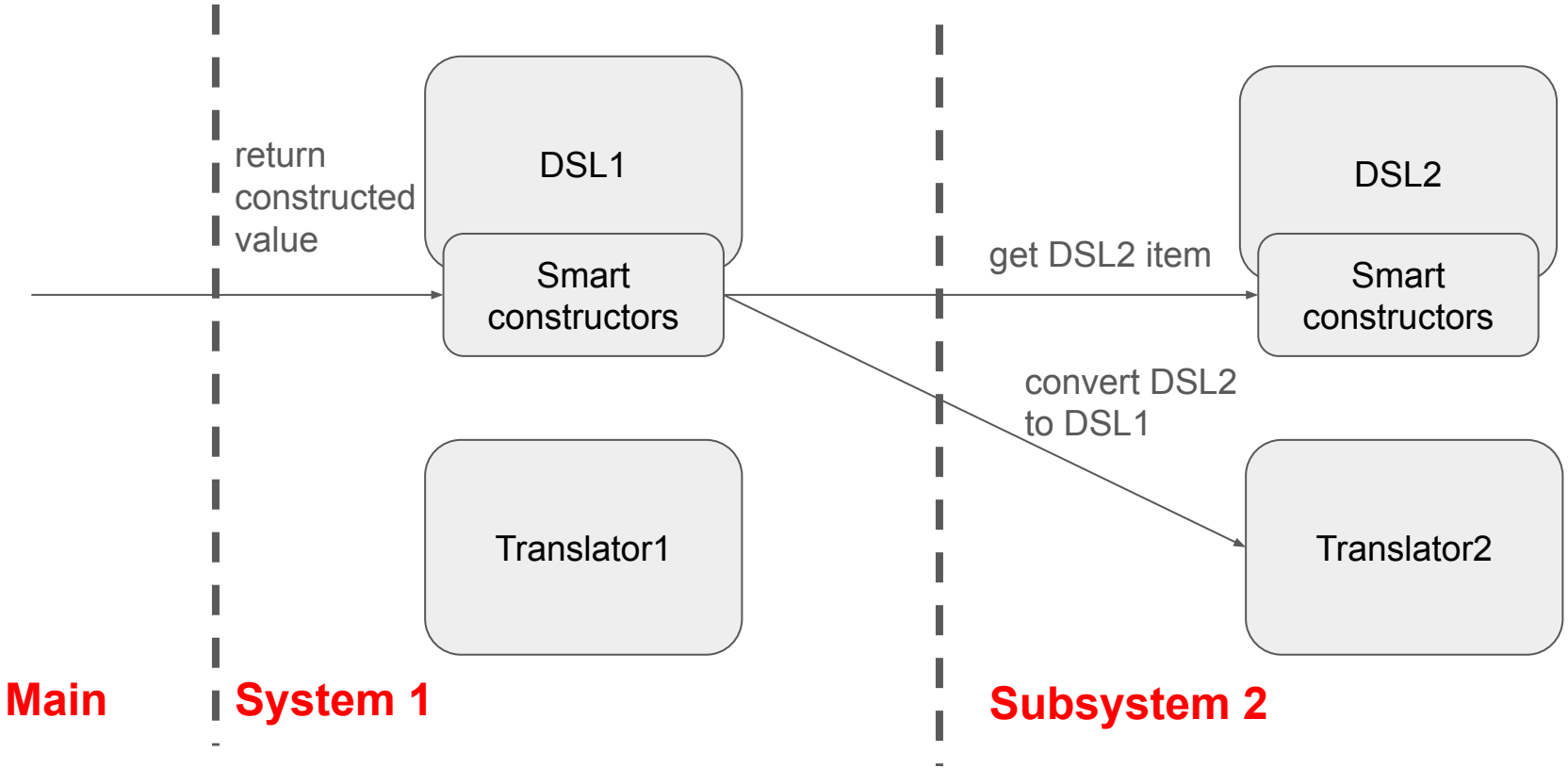
Interpreter and translator. Universal usage as a library



Interpreter and translator. Stacking systems



Interpreter and translator. Stacking systems



Practice

- Introduce smart constructors (based on your choice)
- Modify export policy

DSL and Interpreter. Advanced

- Free monad
- eDSL
- Updated interpreter

How to achieve DSL in Kotlin?

Free monad

- A way to get monad from a functor for free
- A free monad satisfies all the Monad laws, but does not do any collapsing (i.e., computation). It just builds up a nested series of contexts. The user who creates such a free monadic value is responsible for doing something with those nested contexts
- In our case, one action stores another action

```
data Free f a = Pure a | Roll (f (Free f a))
```

Free monad

```haskell
instance Functor UiItem where
    -- apply function for result of previous evaluation a
    fmap f (AddTitleLine level text a) = AddTitleLine level text (f a)
    fmap f (AddTextLine text a) = AddTextLine text (f a)
    fmap f (AddSubItem value text a) = AddSubItem value text (f a)
    fmap f (AddImageUrl url a) = AddImageUrl url (f a)
```

eDSL

- Dependency: free

```
uiComponents = do
  addTextLineElement "Simple text"
  addOrderedItem 1 "Carrot"
  addOrderedItem 2 "Apple"
```

```
addTitleLineElement :: String -> String -> Hdl ()
addTitleLineElement size title = case size of
    "big" -> Free (AddTitleLine H1 title (Pure ()))
    "small" -> Free (AddTitleLine H3 title (Pure ()))
    _ -> Free (AddTitleLine H2 title (Pure ()))
```

Updated interpreter

- We need to handle recursively scopes of Actions
- Iterate over such structure:

    Free OurDSLAction tail

- Example

```
52  -- interpreter --
53  writeToConsole :: Hdl () -> IO ()
54  writeToConsole (Pure _) = putStr ""
55  writeToConsole (Free val) = writeToConsole' val
56
57
58  writeToConsole' (AddTitleLine level text cont) = do
59      print $ case level of
60          H1 -> "# " ++ text
61          H2 -> "## " ++ text
62          H3 -> "### " ++ text
63      writeToConsole cont
```

Demo