Возможности Haskell архитектуры

Вершилов Александр

Про меня

- Разработчик с 2008 года
- Программирую на Haskell с 2011 года
- Senior software developer @ Tweag I/O с 2014 года

https://github.com/qnikst

https://ru.linked.com/in/qnikst

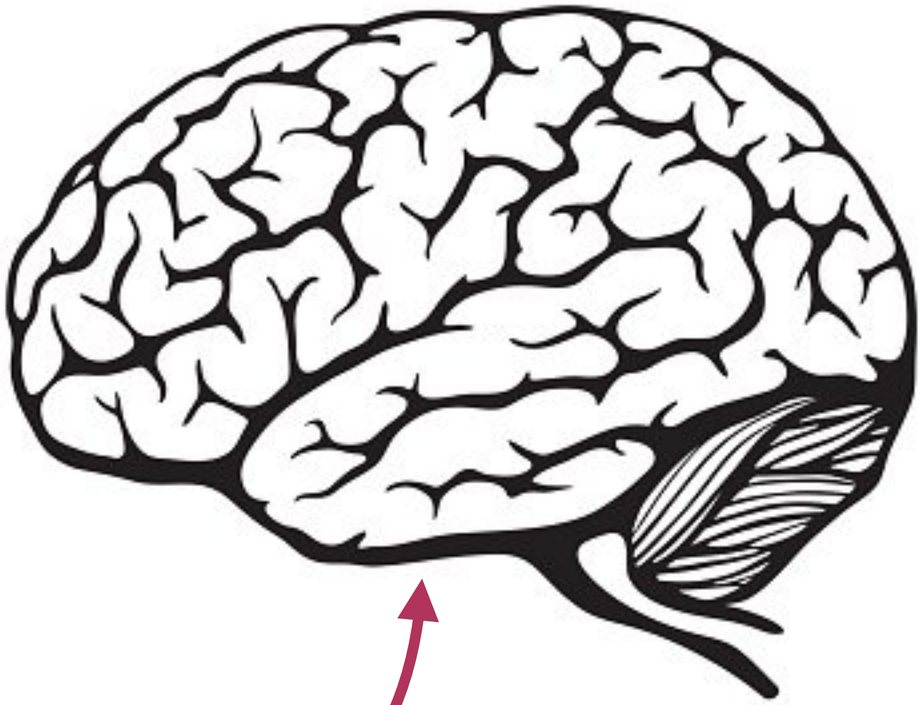
Конец эры первопроходцев

Доклады про архитектуру и паттерны

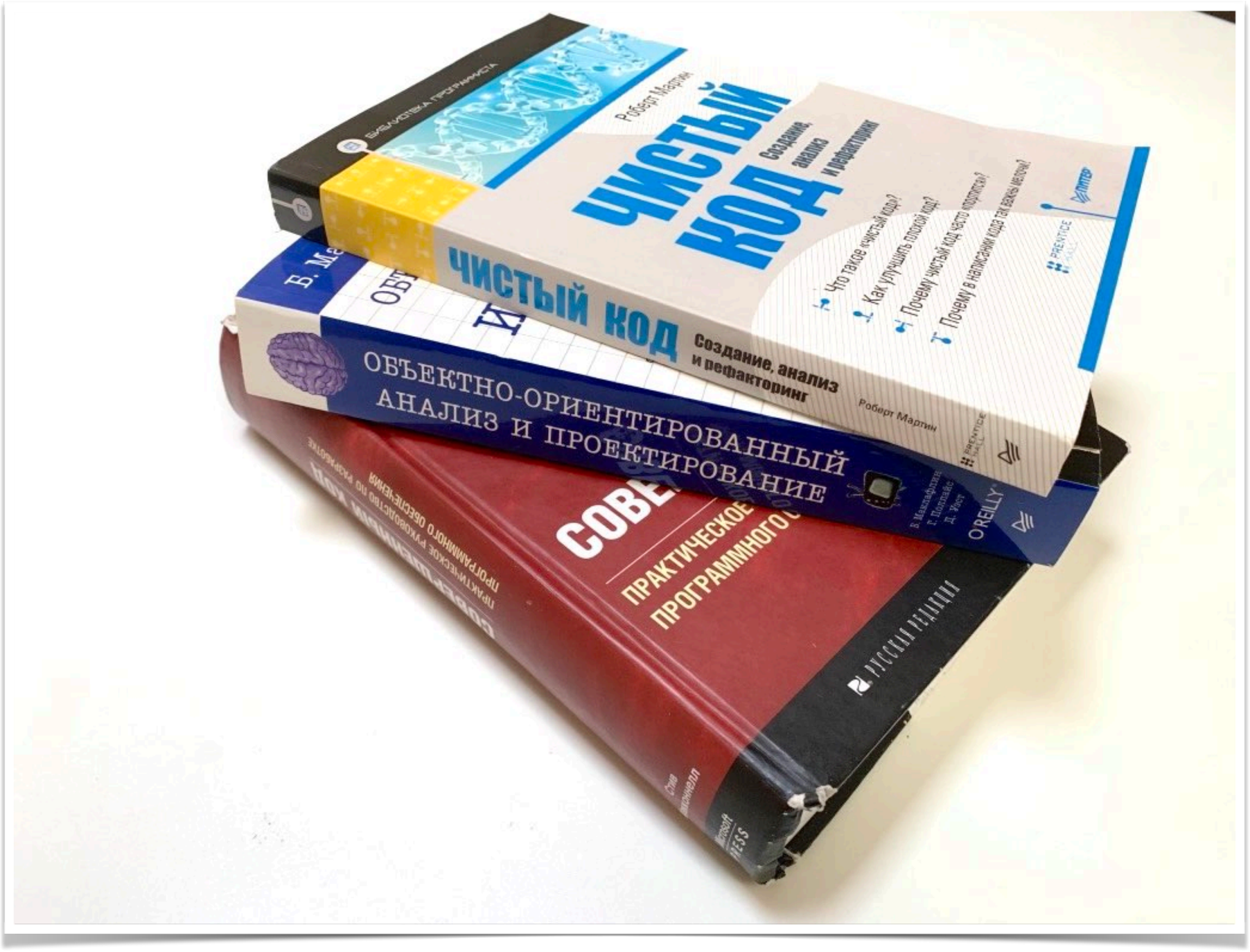
- Алексей Пирогов
  “Функциональный дизайн и паттерны ФП”
  (11:00-11:45 зал “Пушкин”)
- Александр Гранин
  “Final Tagless vs Free Monad”
  (15:00 — 15:45 зал “Пушкин”)
- СТАНИСЛАВ ЧЕРНИЧКИН
  Низкоуровневая оптимизация программ на Haskell
  (17:15 - 18:00 зал “Достоевский”)

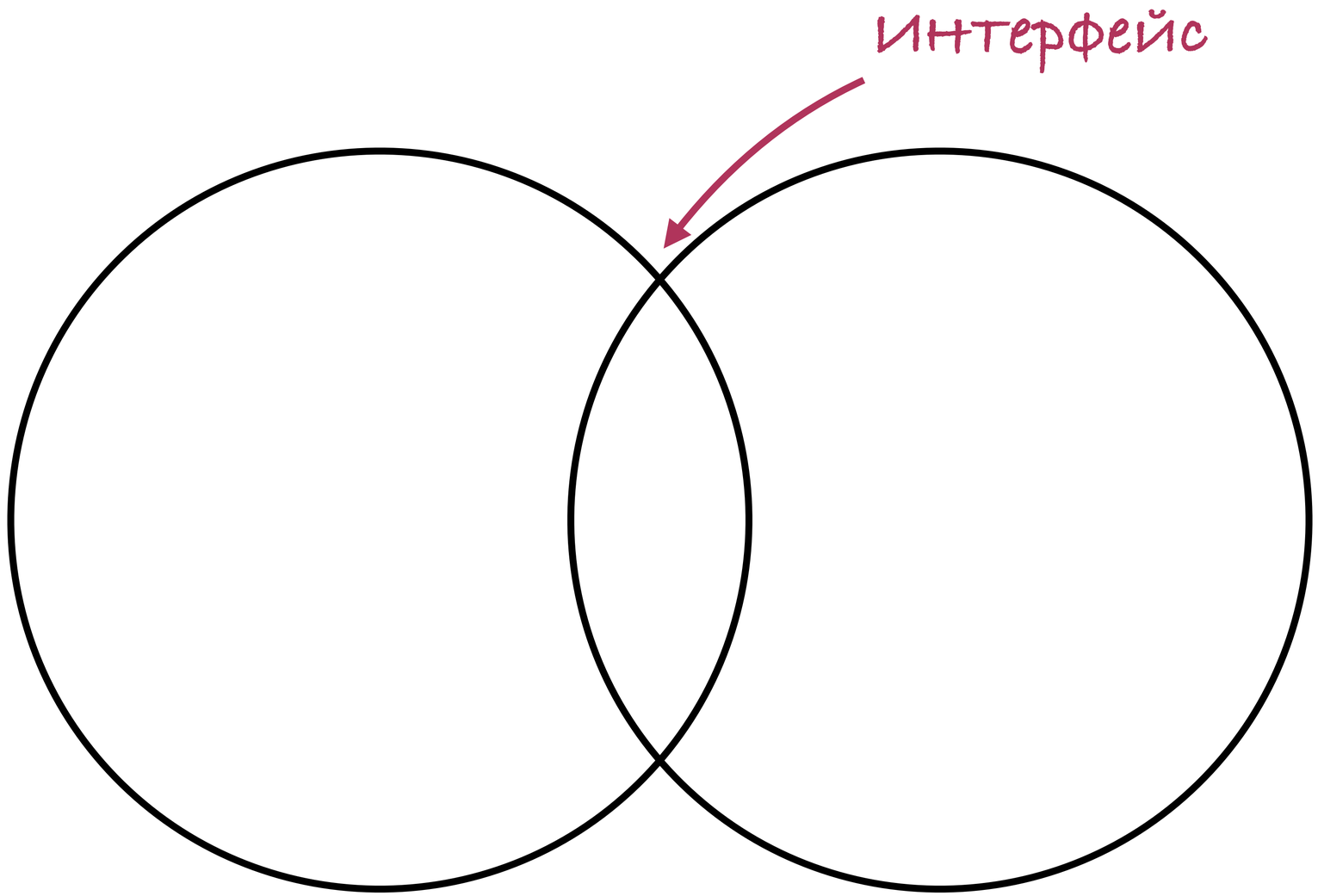
Как мы привыкли решать проблемы?

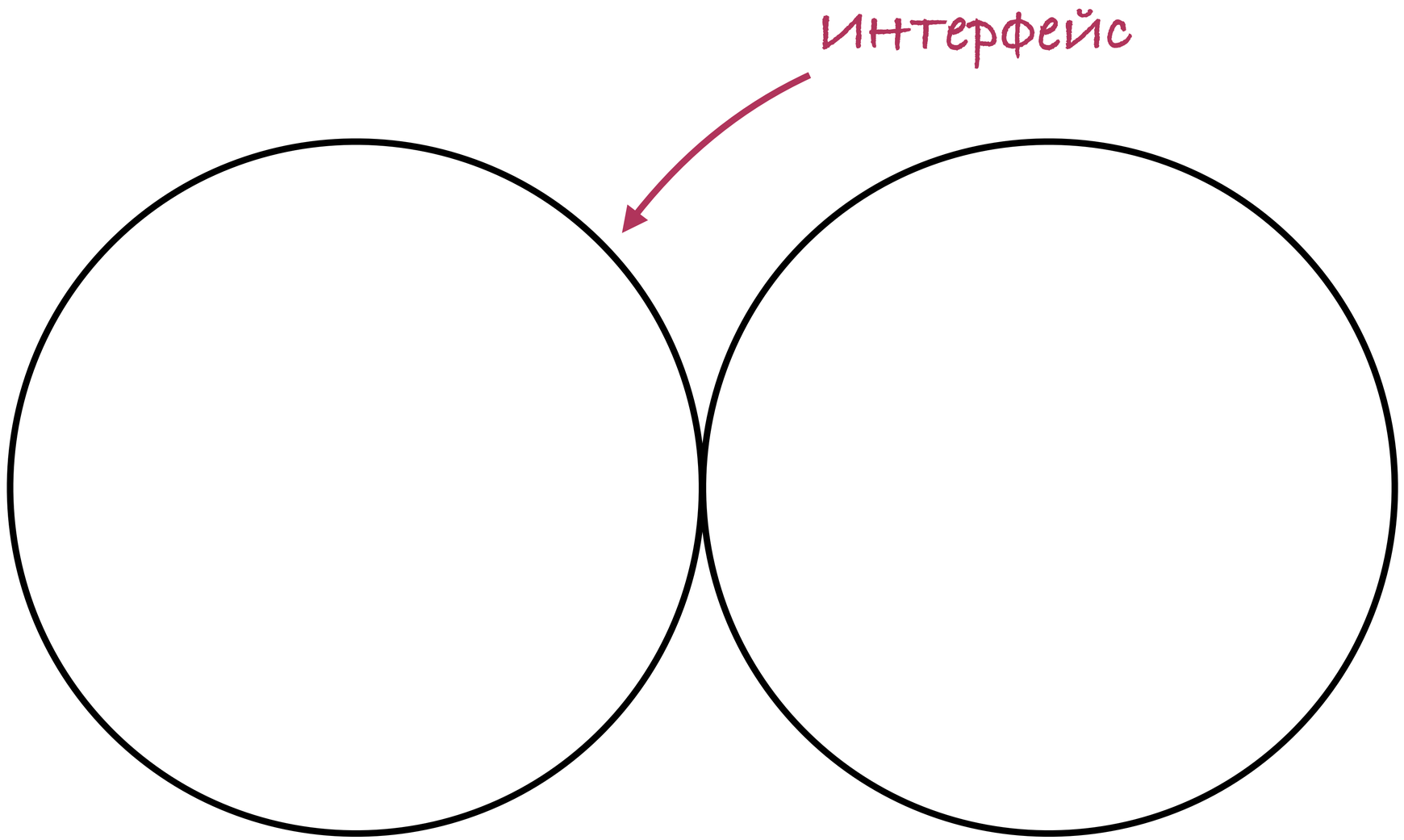
- Усердие и труд
- Паттерны
- Принципы



Это проблема

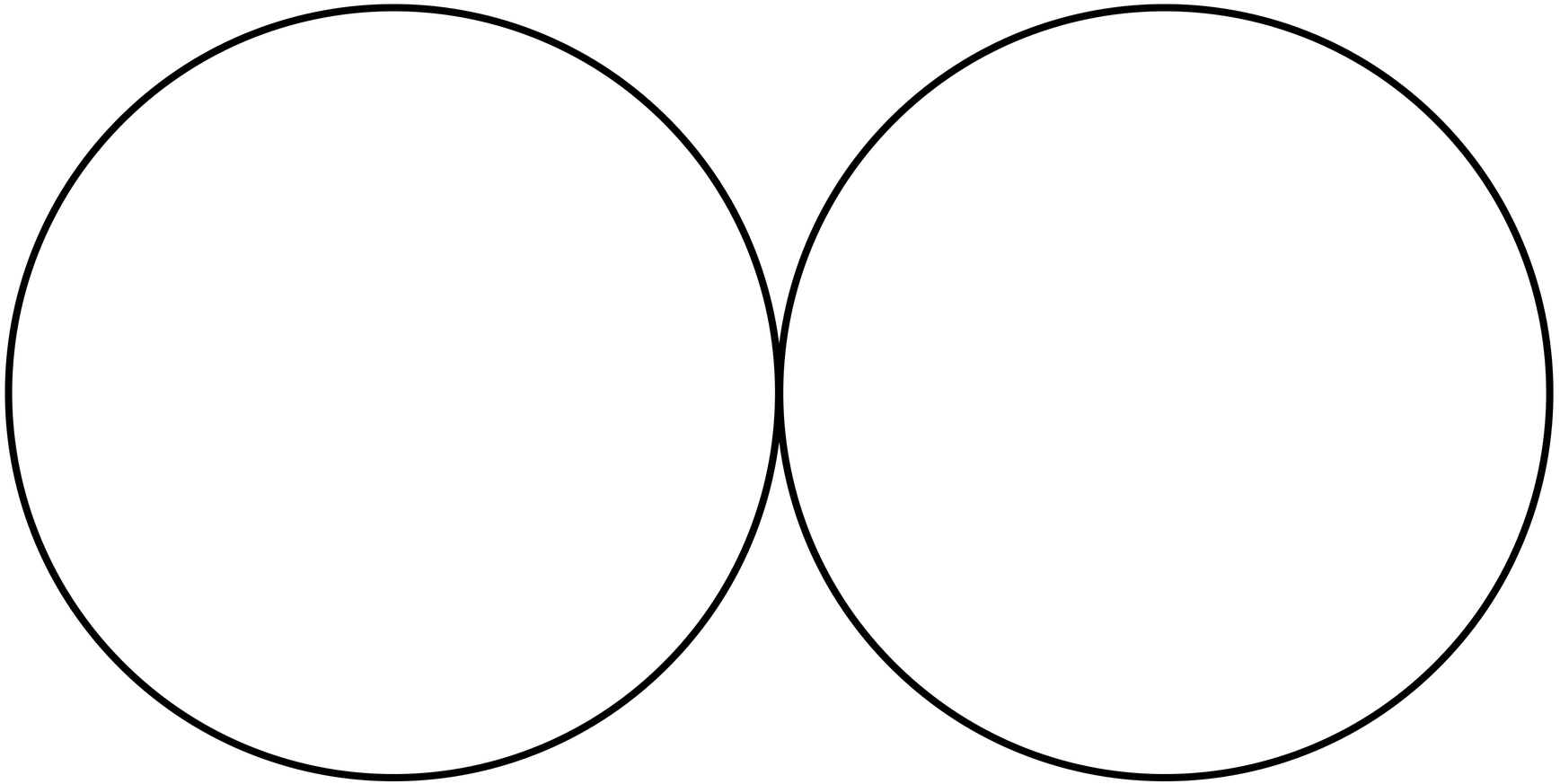

Роберт Мартин
ЧИСТЫЙ КОД
Создание, анализ и рефакторинг
Что такое «чистый код»?
Как улучшить плохой код?
Почему чистый код часто оказывается?
Почему в написании кода так важны мелочи?
ПИТЕР
ЧИСТЫЙ КОД
Создание, анализ и рефакторинг
Роберт Мартин
ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ
O'REILLY
ПРАКТИЧЕСКОЕ
ПРОГРАММНОГО
РУССКАЯ РЕДАКЦИЯ


Интерфейс


Интерфейс

Абстракция

Композиция



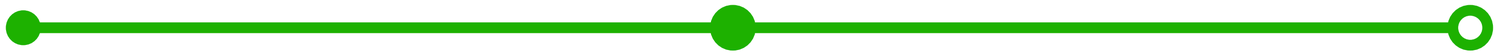
Чистые функции

$$f :: A \to B$$

$$g :: B \to C$$

$$f :: A \to B$$

$$g :: B \to C$$



$$g \circ f :: A \to C$$

Functional architecture
The pits of success

Mark Seemann

https://www.youtube.com/watch?v=US8QG9I1XW0

Основные положения

- Стабильность кода
- Разделение уровней ответственности
- Тестируемость

F#-подобный язык

```
check capacity getReservedSeats reservation =
  let reservedSeats = getReservedSeats
  if capacity < reservation.Quantity + reservedSeats
  then Failure CapacityExceeded
  else Success reservation
```

F#-подобный язык

```
check capacity getReservedSeats reservation =
  let reservedSeats = getReservedSeats
  if capacity < reservation.Quantity + reservedSeats
  then Failure CapacityExceeded
  else Success reservation
```

F#-подобный язык

```
check capacity getReservedSeats reservation =
  let reservedSeats = getReservedSeats
  if capacity < reservation.Quantity + reservedSeats
  then Failure CapacityExceeded
  else Success reservation
```

F#-подобный язык

```
check capacity getReservedSeats reservation =
  let reservedSeats = getReservedSeats
  if capacity < reservation.Quantity + reservedSeats
  then Failure CapacityExceeded
  else Success reservation
```

F#-подобный язык

```
check capacity getReservedSeats reservation =
  let reservedSeats = getReservedSeats
  if capacity < reservation.Quantity + reservedSeats
  then Failure CapacityExceeded
  else Success reservation
```

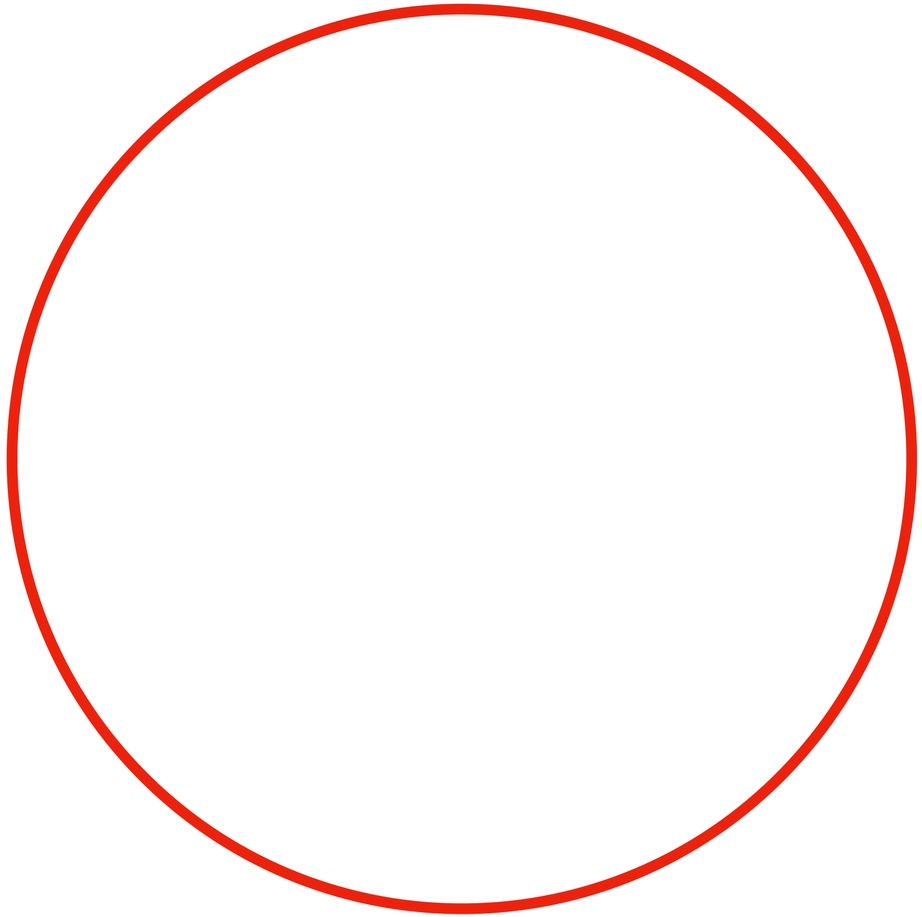
F#-подобный язык

```
check capacity getReservedSeats reservation =
  let reservedSeats = getReservedSeats
  if capacity < reservation.Quantity + reservedSeats
  then Failure CapacityExceeded
  else Success reservation
```

F#-подобный язык

```
check capacity getReservedSeats reservation =
  let reservedSeats = getReservedSeats
  if capacity < reservation.Quantity + reservedSeats
  then Failure CapacityExceeded
  else Success reservation
```

Контекст



```
check
  :: Int
  -> m Int
  -> Reservation
  -> m (Result Reservation)
check capacity getReservedSeats reservation = do
  reservedSeats <- getReservedSeats
  if capacity < reservation.Quantity + reservedSeats
  then return $ Failure CapacityExceeded
  else return $ Success reservation
```

```
check
  :: Int
  -> m Int
  -> Reservation
  -> m (Result Reservation)
check capacity getReservedSeats reservation = do
  reservedSeats <- getReservedSeats
  if capacity < reservation.Quantity + reservedSeats
  then return $ Failure CapacityExceeded
  else return $ Success reservation
```

```
check
  :: Int
  -> m Int
  -> Reservation
  -> m (Result Reservation)
check capacity getReservedSeats reservation = do
  reservedSeats <- getReservedSeats
  if capacity < reservation.Quantity + reservedSeats
  then return $ Failure CapacityExceeded
  else return $ Success reservation
```

m

```
check
  :: Int
  -> m Int
  -> Reservation
  -> m (Result Reservation)
checkCapacity getReservedSeats reservation = do
  reservedSeats <- getReservedSeats
  if capacity < reservation.Quantity + reservedSeats
  then return $ Failure CapacityExceeded
  else return $ Success reservation
```

Identity → **IO**

```
postReservation
  :: ReservationRendition
  -> IO (HttpResult ())
postReservation candidate = toHttpResult $ runEitherT $ do
   r <- hoistEither $ validateReservation candidate
   let i = liftIO $ getReservationSeatsFromDb connStr
                  $ date r
   log DEBUG $ “checking capacity for “ <> show r
   hoistEither $ checkCapacity 10 i r >>=
     liftIO . saveReservation connStr
```

```
postReservation
  :: ReservationRendition
  -> IO (HttpResult ())
postReservation candidate = toHttpResult $ runEitherT $ do
    r <- hoistEither $ validateReservation candidate
    let i = liftIO $ getReservationSeatsFromDb connStr
                   $ date r
    log DEBUG $ “checking capacity for“ <> show r
    hoistEither $ checkCapacity 10 i r >>=
      liftIO . saveReservation connStr
```

```
postReservation
  :: ReservationRendition
  -> IO (HttpResult ())
postReservation candidate = toHttpResult $ runEitherT $ do
    r <- hoistEither $ validateReservation candidate
    let i = liftIO $ getReservationSeatsFromDb connStr
                   $ date r
    log DEBUG $ “checking capacity for“ <> show r
    hoistEither $ checkCapacity 10 i r >>=
      liftIO . saveReservation connStr
```

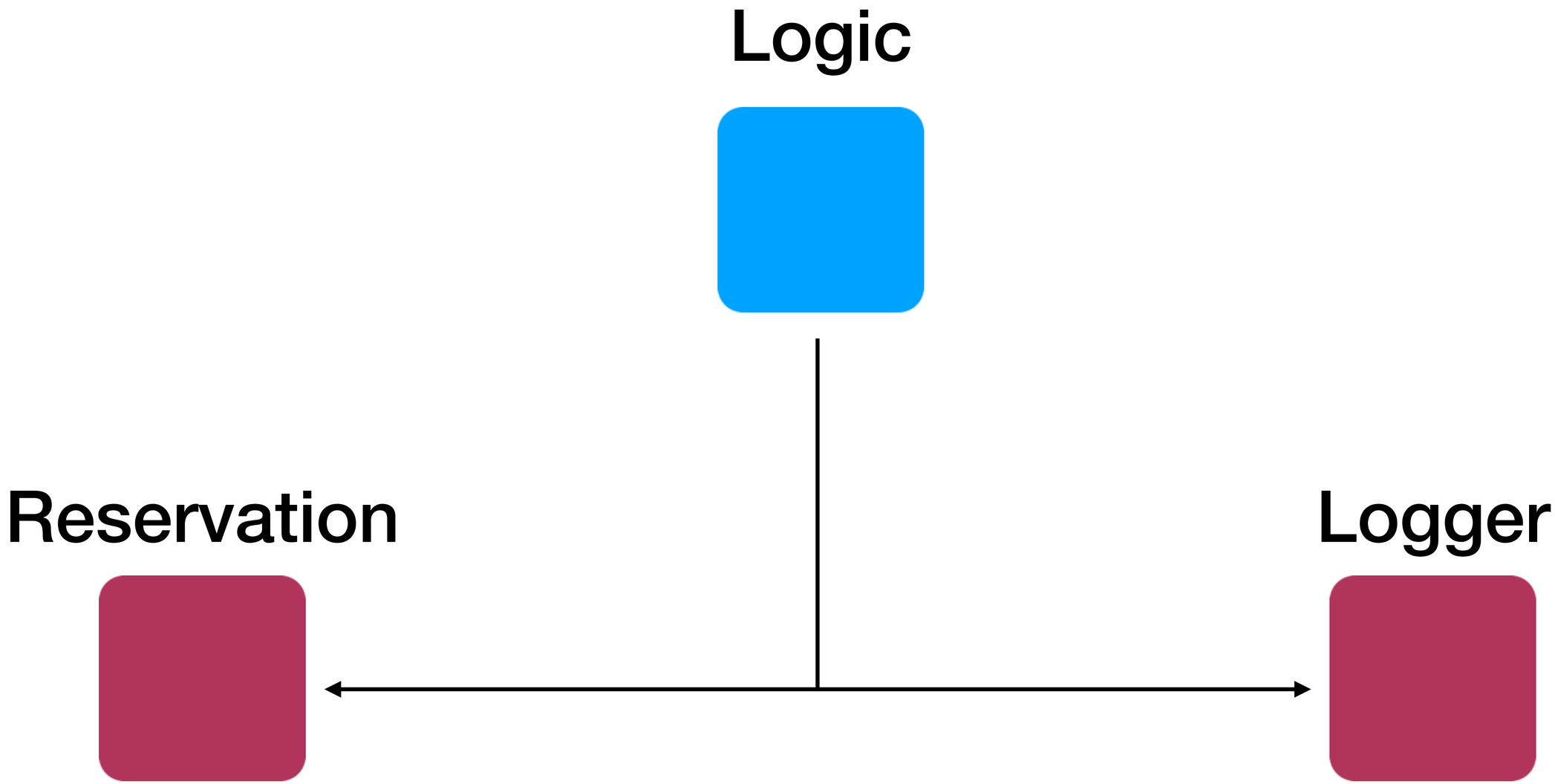
```
postReservation
  :: ReservationRendition
  -> IO (HttpResult ())
postReservation candidate = toHttpResult $ runEitherT $ do
   r <- hoistEither $ validateReservation candidate
   let i = liftIO $ getReservationSeatsFromDb connStr
                  $ date r
   log DEBUG $ “checking capacity for“ <> show r
   hoistEither $ checkCapacity 10 i r >>=
     liftIO . saveReservation connStr
```

```
postReservation
  :: ReservationRendition
  -> IO (HttpResult ())
postReservation candidate = toHttpResult $ runEitherT $ do
   r <- hoistEither $ validateReservation candidate
   let i = liftIO $ getReservationSeatsFromDb connStr
                  $ date r
   log DEBUG $ “checking capacity for“ <> show r
   hoistEither $ checkCapacity 10 i r >>=
     liftIO . saveReservation connStr
```

```
postReservation
  :: ReservationRendition
  -> IO (HttpResult ())
postReservation candidate = toHttpResult $ runEitherT $ do
   r <- hoistEither $ validateReservation candidate
   let i = liftIO $ getReservationSeatsFromDb connStr
                  $ date r
   log DEBUG $ "checking capacity for" <> show r
   hoistEither $ checkCapacity 10 i r >>=
     liftIO . saveReservation connStr
```

```
postReservation
  :: ReservationRendition
  -> IO (HttpResult ())
postReservation candidate = toHttpResult $ runEitherT $ do
   r <- hoistEither $ validateReservation candidate
   i <- liftIO $ getReservationSeatsFromDb connStr
               $ date r
   log DEBUG $ "checking capacity for" <> show r
   hoistEither $ checkCapacity 10 i r >>=
     liftIO . saveReservation connStr
```

```
data Logic = Logic
    { getReservationSeatsFromDb
       :: ConnStr -> Date -> IO Int
    , saveReservation
       :: ConnStr -> Date -> Reservation -> IO ()
    , log :: LogLevel -> String -> IO ()
    }
```


Logic
Reservation
Logger

Getting things done in Haskell

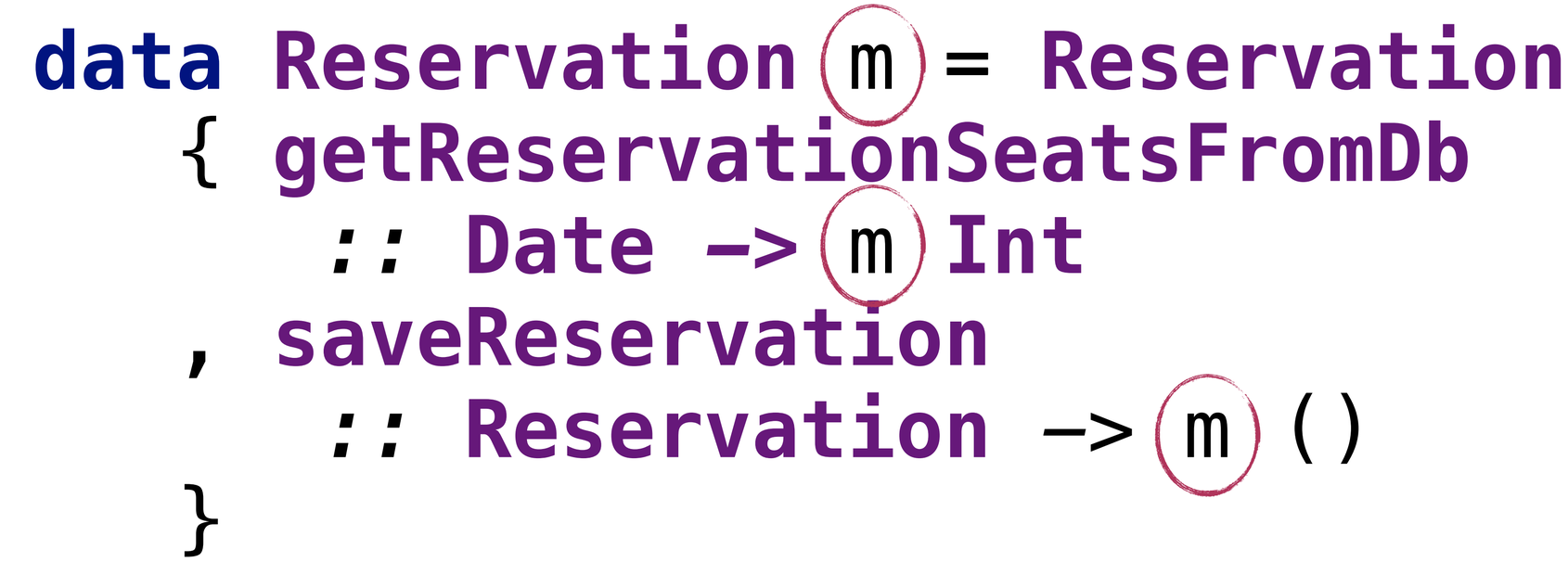
Jasper Van der Jeugt

https://www.youtube.com/watch?v=-X1vrxQUETM

```
transaction :: Db.Service -> (TransactionHandle -> IO a) -> IO a
```

```
transaction dbService $ \handle -> return handle
```

```
transaction :: Monad m
  => Db.Service
  -> ReaderT TransactionHandle m a
  -> m a
```


data Reservation m = Reservation
{ getReservationSeatsFromDb
:: Date -> m Int
, saveReservation
:: Reservation -> m ()
}

```
Эффекты
transaction db $ do
    Эффекты + возможности DB
```

```
logic
 :: (… => m)
 => Reservation.Service m
 -> Logger.Service m
 -> UserInfo.Service m
 -> ReservationRendition
 -> IO (HttpResult ())
```

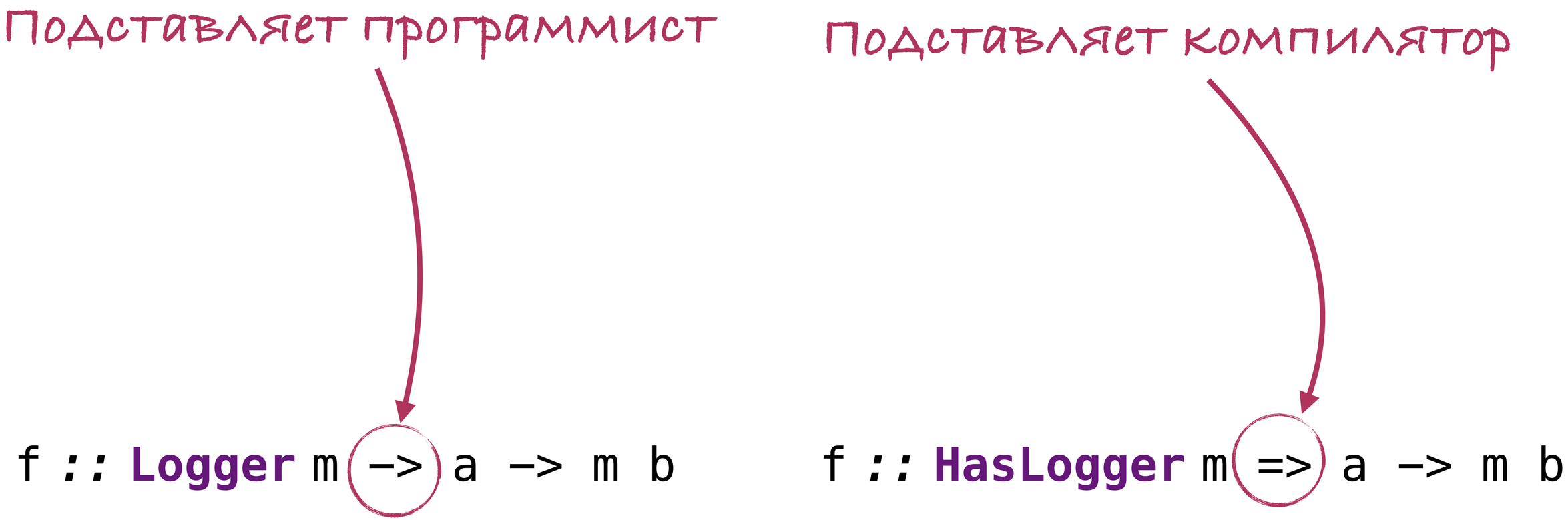
Стиль MTL

```haskell
data Logger m = Logger
  { log :: LogLevel
        -> String
        -> m ()
  }
```

```haskell
class HasLogger m where
  log :: LogLevel
      -> String
      -> m ()
```

```haskell
f :: Logger m -> a -> m b
```

```haskell
f :: HasLogger m => a -> m b
```


Подставляет программист
Подставляет компилятор
f :: Logger m -> a -> m b
f :: HasLogger m => a -> m b

Стиль MTL

Упрощенный Final Tagless

- Дополнительный язык (eDSL) встраивается в основной
- Инстанс класса типов - интерпретатор языка

`HasLogger m`

`HasReservation m`

`MonadThrow E m`

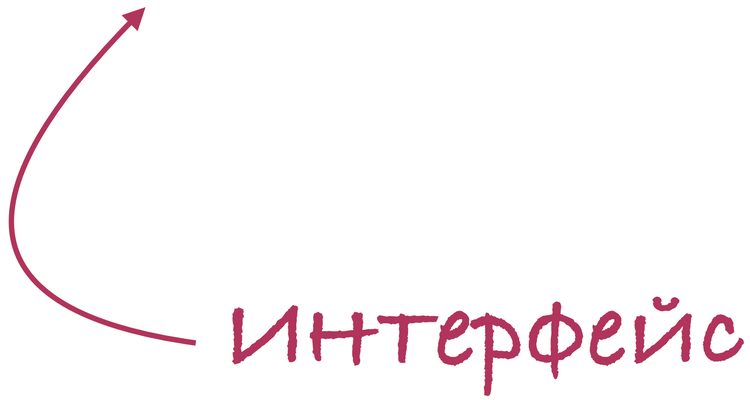
`HasLogger m`

`HasReservation m`

`MonadThrow E m`

```
f :: (HasLogger m, HasReservation m, MonadThrow E m) => m ()
```


```
class HasLogger m where
    log ::  LogLevel -> String -> m ()
```



```haskell
class HasLogger m where
    log ::  LogLevel -> String -> m ()
```

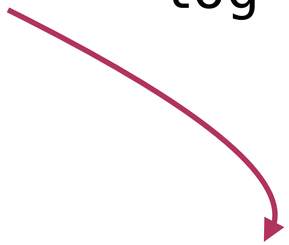
Используем интерфейс

```haskell
businessLogic :: HasLogger m => …
businessLogic = do
    log DEBUG “Hello, FPure-2019!”
```

Используем интерфейс

```haskell
class HasLogger m where
    log ::  LogLevel -> String -> m ()

businessLogic :: HasLogger m => …
businessLogic = do
   log DEBUG “Hello, FPure-2019!”
```



```haskell
newtype MyLogger a = MyLogger
    { runMyLogger ::  ReaderT IO Handle a }
```

```haskell
class HasLogger m where
    log ::  LogLevel -> String -> m ()

businessLogic :: HasLogger m => …
businessLogic = do
   log DEBUG “Hello, FPure-2019!”
```

интерпретатор

```haskell
newtype MyLogger a = MyLogger
    { runMyLogger ::  ReaderT IO Handle a }

instance HasLogger MyLogger where
  log lvl msg = MyLogger $
    ask >>= doLog lvl msg
```

```haskell
class HasLogger m where
    log ::  LogLevel -> String -> m ()

businessLogic :: HasLogger m => …
businessLogic = do
   log DEBUG “Hello, FPure-2019!”
```

```haskell
newtype MyLogger a = MyLogger
    { runMyLogger ::  ReaderT IO Handle a }

instance HasLogger MyLogger where
  log lvl msg = MyLogger $
    ask >>= doLog lvl msg
```

```haskell
class HasLogger m where
    log ::  LogLevel -> String -> m ()

businessLogic :: HasLogger m => …
businessLogic = do
   log DEBUG “Hello, FPure-2019!”
```

```haskell
newtype MyLogger a = MyLogger
    { runMyLogger ::  ReaderT IO Handle a }
```

```haskell
instance HasLogger MyLogger where
  log lvl msg = MyLogger $
    ask >>= doLog lvl msg
```

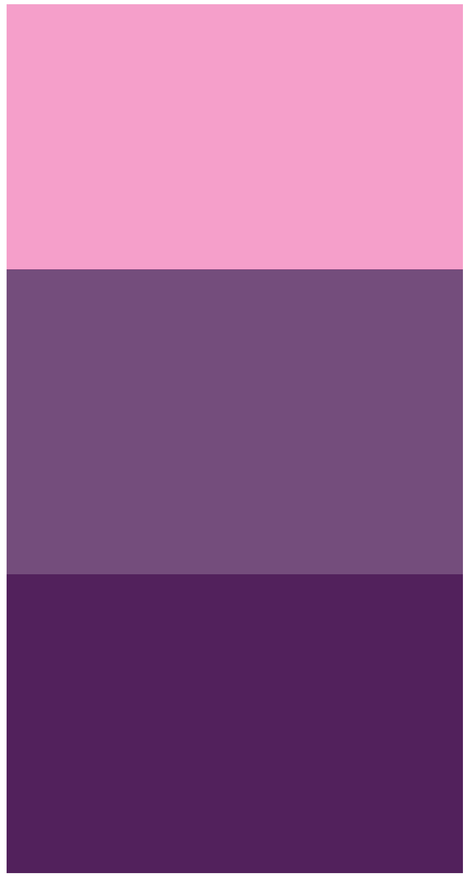
начинаются проблемы

```haskell
runReaderT (runMyLogger businessLogic) handle
```

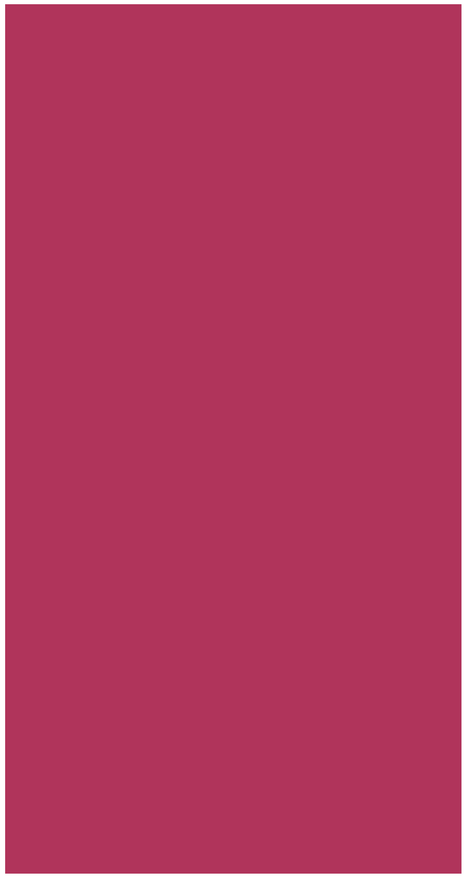
lift (lift ask) :: WriterT w (StateT s (Reader Env)) Env

lift ask :: StateT a (Reader Env) Env

ask :: Reader Env



ask :: MonadReader Env m => m Env



ask :: MonadReader Env m => m Env

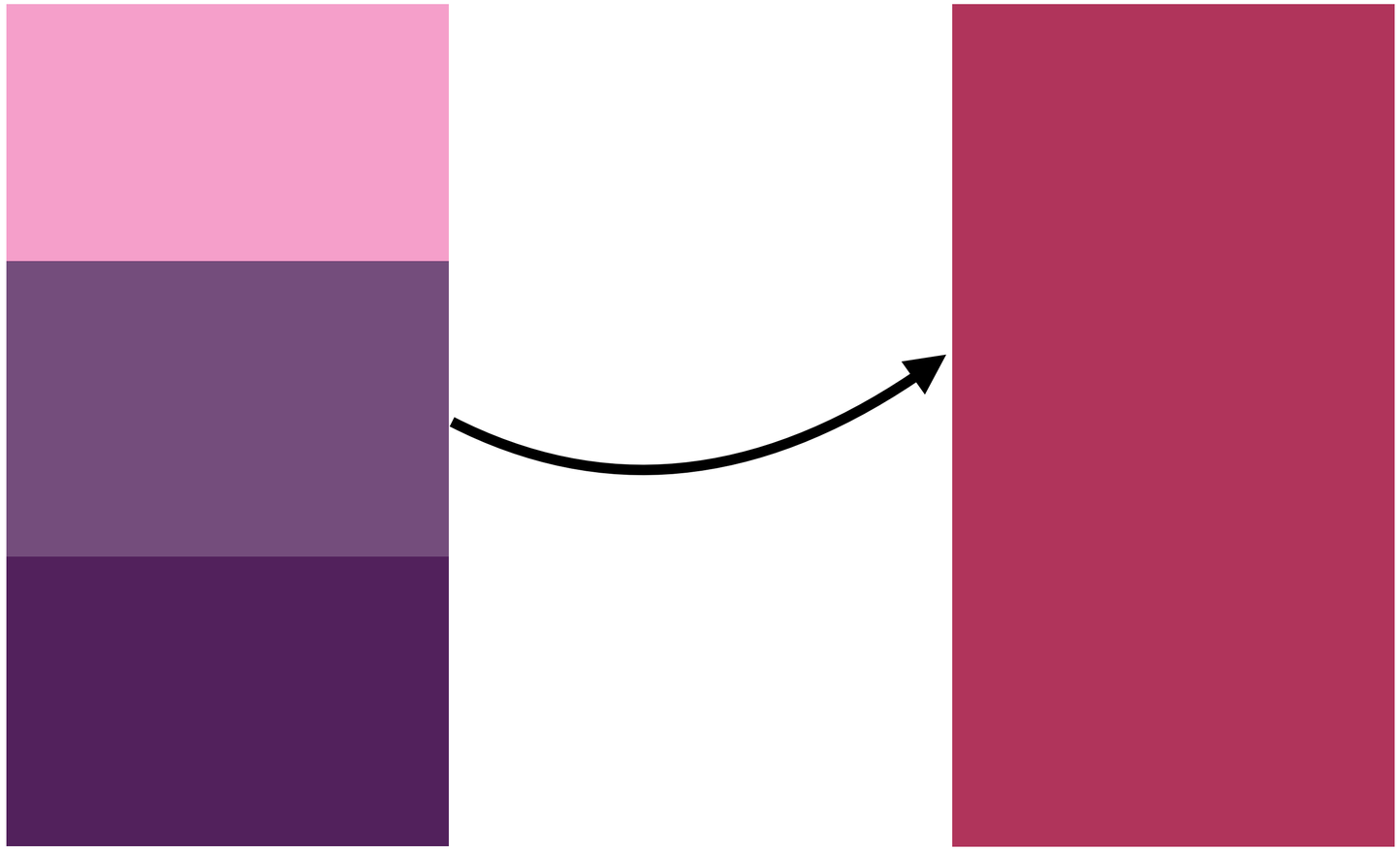
```haskell
instance MonadReader t m => MonadReader (StateT a m)
  where
    local f m = StateT $ (local f) . runStateT m
```

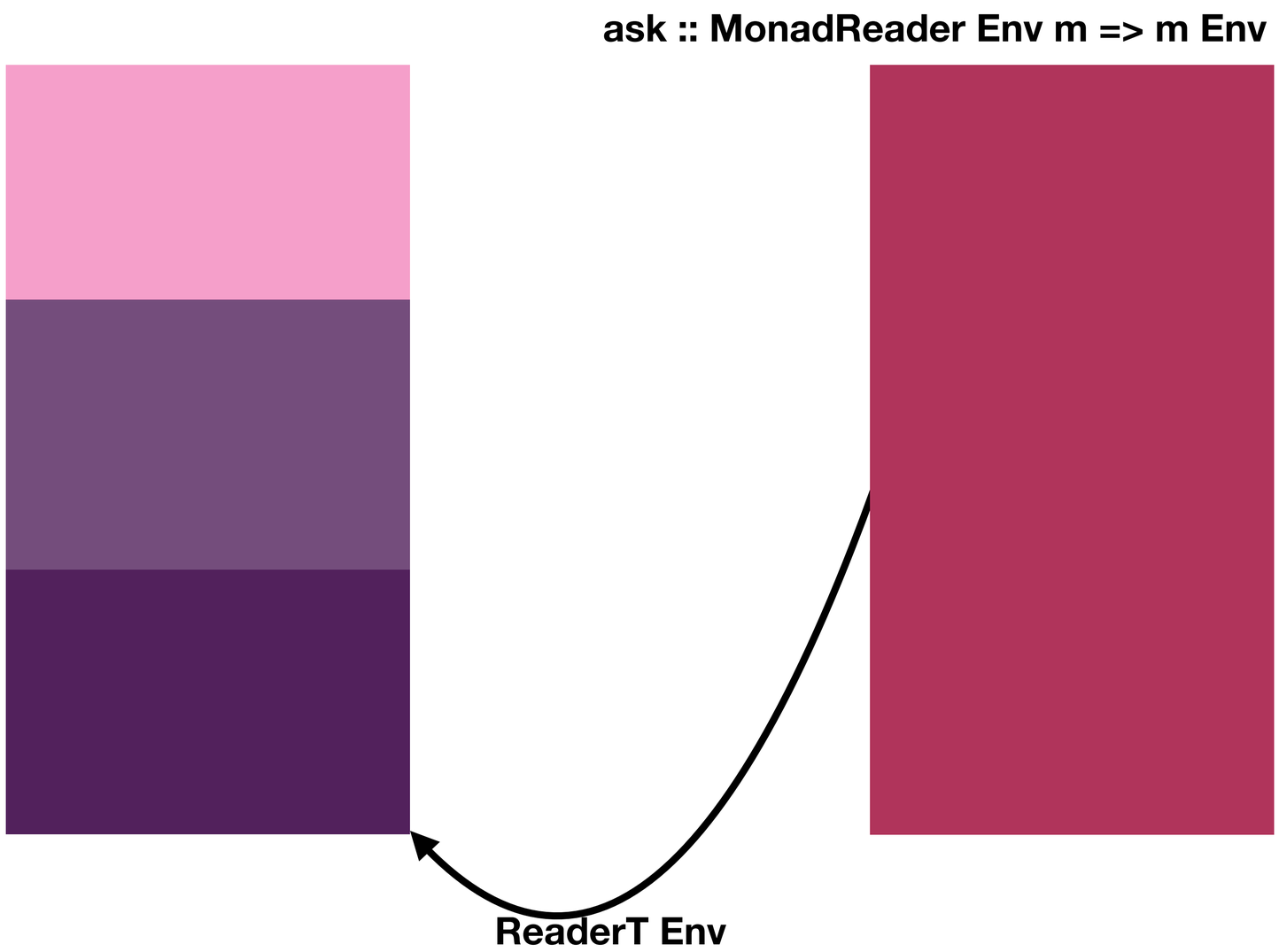
```haskell
instance MonadReader t m => MonadReader (WriterT a m)
  where
    local = mapWriterT . Local
```

$$O(n^2)$$

ReaderT

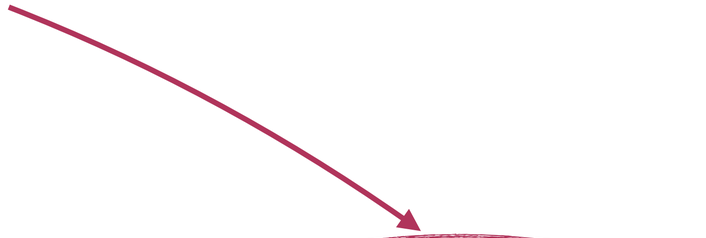
ask :: MonadReader Env m => m Env



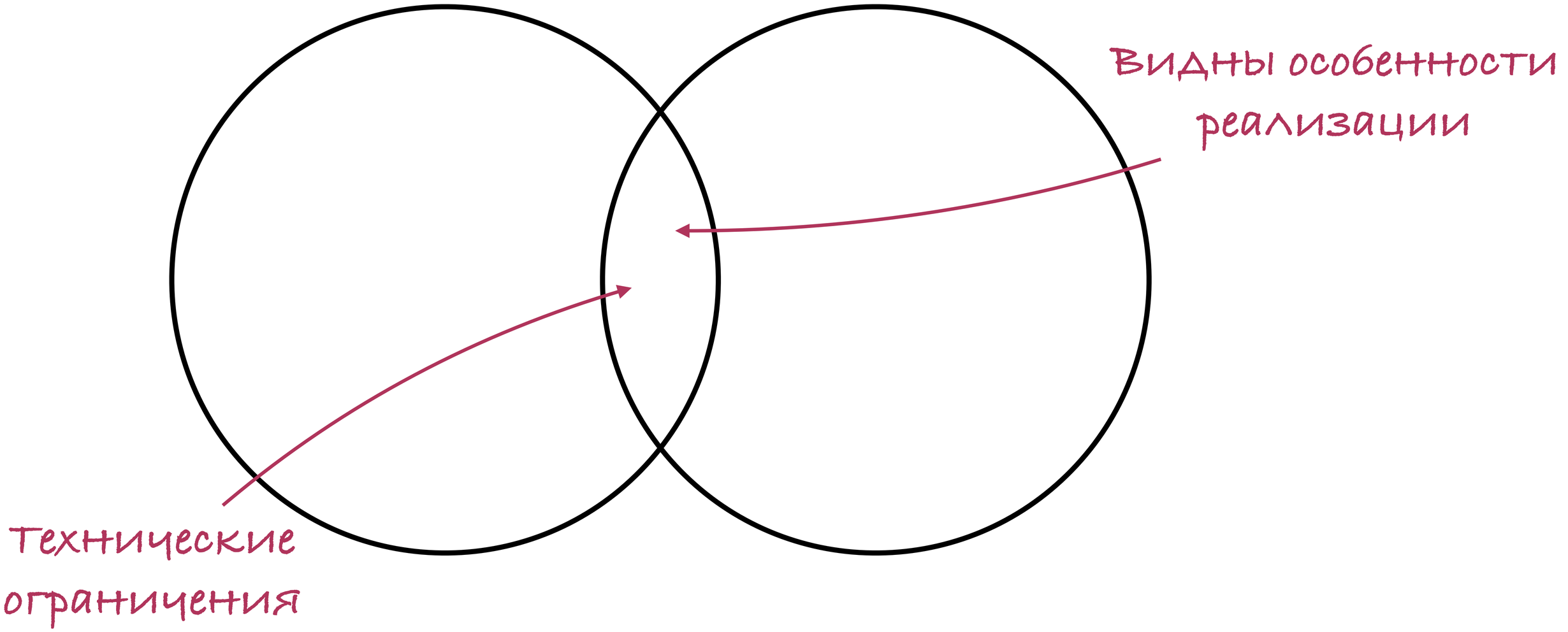

ask :: MonadReader Env m => m Env
ReaderT Env

Если мы знаем t,

то мы знаем m



```
class MonadReader t m | m -> t where
    local :: (t -> t) -> m a -> m a
```


Видны особенности
реализации
Технические
ограничения

“If you don't use the mtl, the mtl-style is fine.”

– Arnaud Spiwack

Capability

http://hackage.haskell.org/package/capability

Лицензия: BSD-3

Поддерживается Tweag I/O

Не используем автоматический вывод реализаций классов типов

- Кодируем возможности eDSL
- Описываем свойства классами
- Комбинируем свойства через Constraints

Как структурировать вычисления?

с `IO` эффектами

```
class HasState tag m s | tag m -> s where
    put_ :: Proxy# tag -> s -> m ()
    get_ :: Proxy# tag -> m s
```

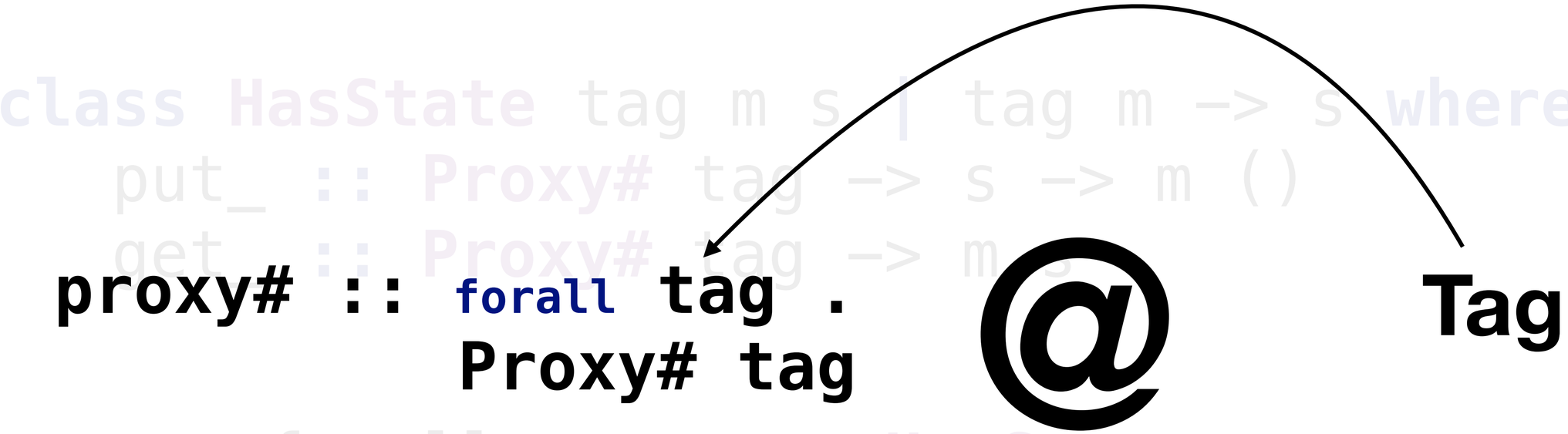
```
class HasState tag m s | tag m -> s where
    put_ :: Proxy# tag -> s -> m ()
    get_ :: Proxy# tag -> m s
```

```
class HasState tag m s | tag m -> s where
    put_ :: Proxy# tag -> s -> m ()
    get_ :: Proxy# tag -> m s
```

Proxy# ::Proxy *tag*

```haskell
class HasState tag m s | tag m -> s where
    put_ :: Proxy# tag -> s -> m ()
    get_ :: Proxy# tag -> m s


put :: forall tag m s. HasState tag m s
    => s -> m ()
put = put_ (proxy# @tag)
```



Proxy# Tag

```
countWordsAndLetters
  :: ( HasState “words” Int m
     , HasState “chars” Int m)
  -> T.Text -> m ()
countWordsAndLetters line =
   for_ (T.words line) $ \word -> do
     modify @“words” (+1)
     modify @“char” (+(T.length word))
```

```
countWordsAndLetters
  :: ( HasState “words” Int m
     , HasState “chars” Int m)
  -> m ()
countWordsAndLetters = do
    line <- getLine
    for_ (T.words line) $ \word -> do
      modify @“words” (+1)
      modify @“char” (+(T.length word))
```

```
countWordsAndLetters
  :: ( HasState “words” Int m
     , HasState “chars” Int m)
  -> m ()
countWordsAndLetters = do
    line <- getLine
    for_ (T.words line) $ \word -> do
      modify @“words” (+1)
      modify @“char” (+(T.length word))
```

Примитивы в библиотеке

- Состояние (HasState)
- Окружение (HasReader)
- Потоки (HasStream)
- Поддержка исключений (HasThrow, HasCatch)

Как использовать созданные примитивы?

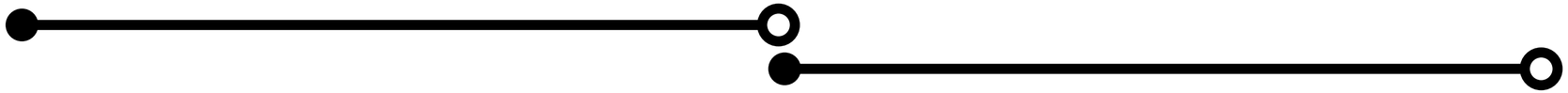
Состояние (HasState)

Пришло исключение

игнорируем

восстанавливаем состояние

храним состояние



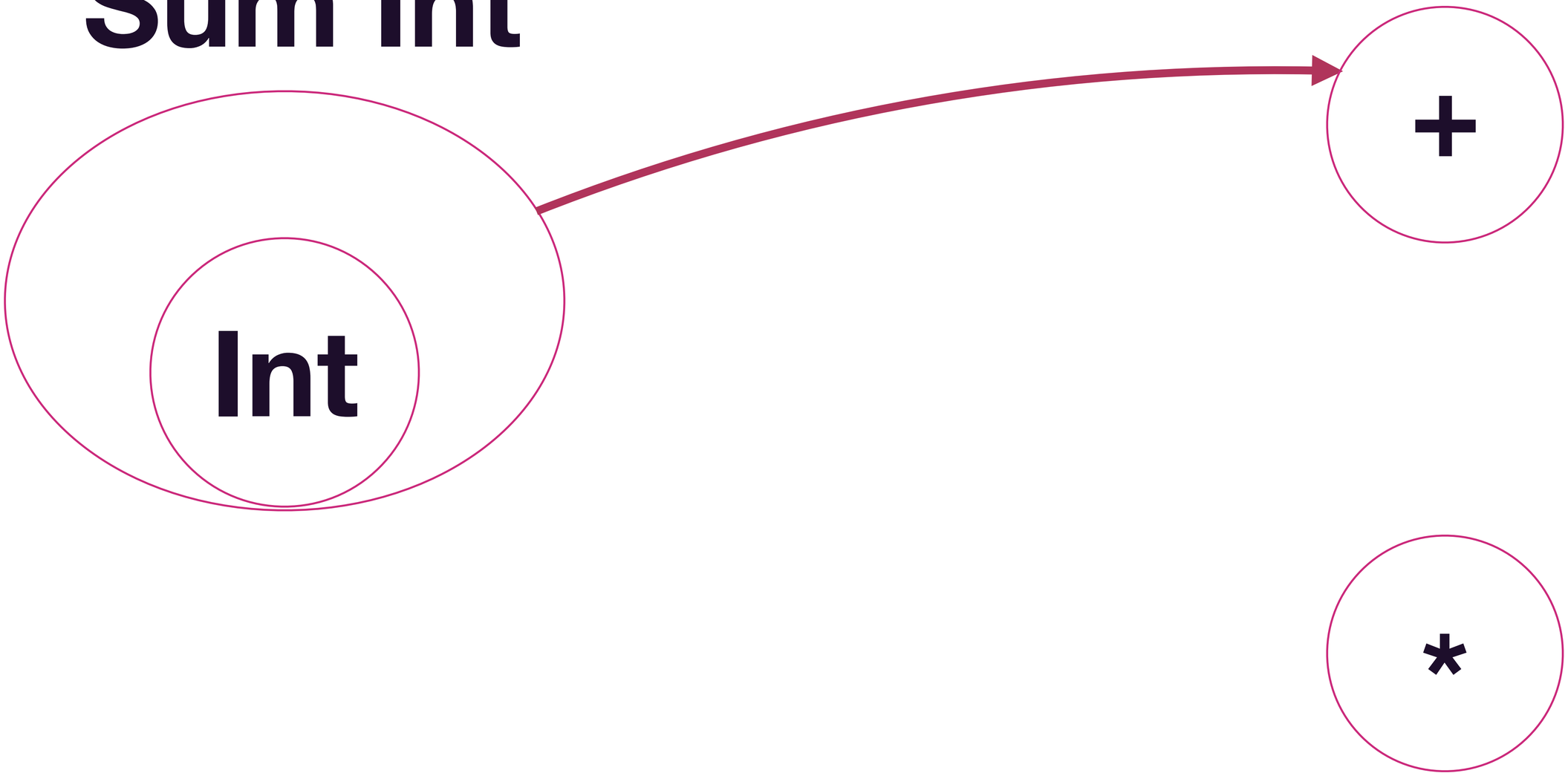
Int

+

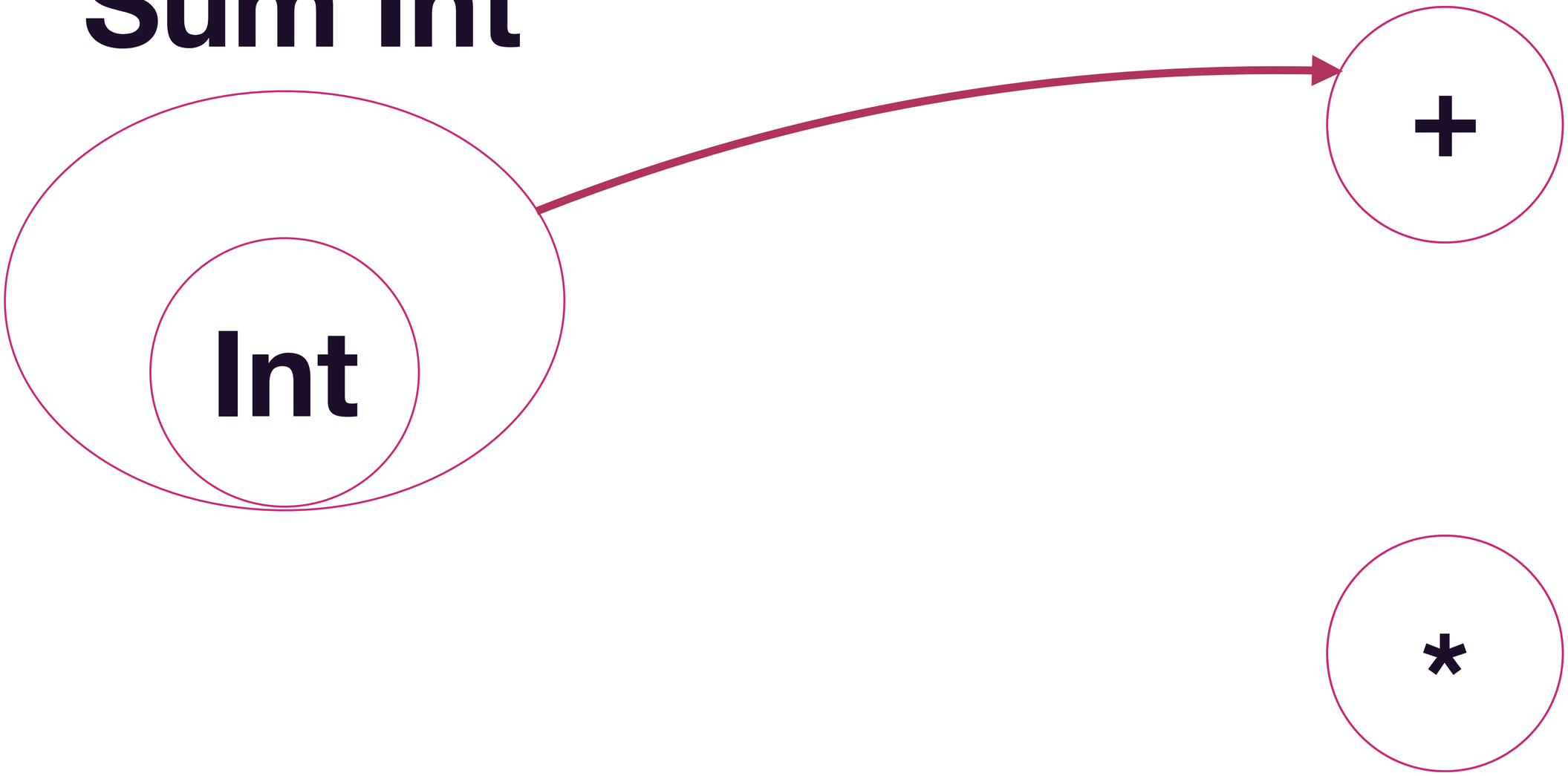
Классы типов когерентны

*

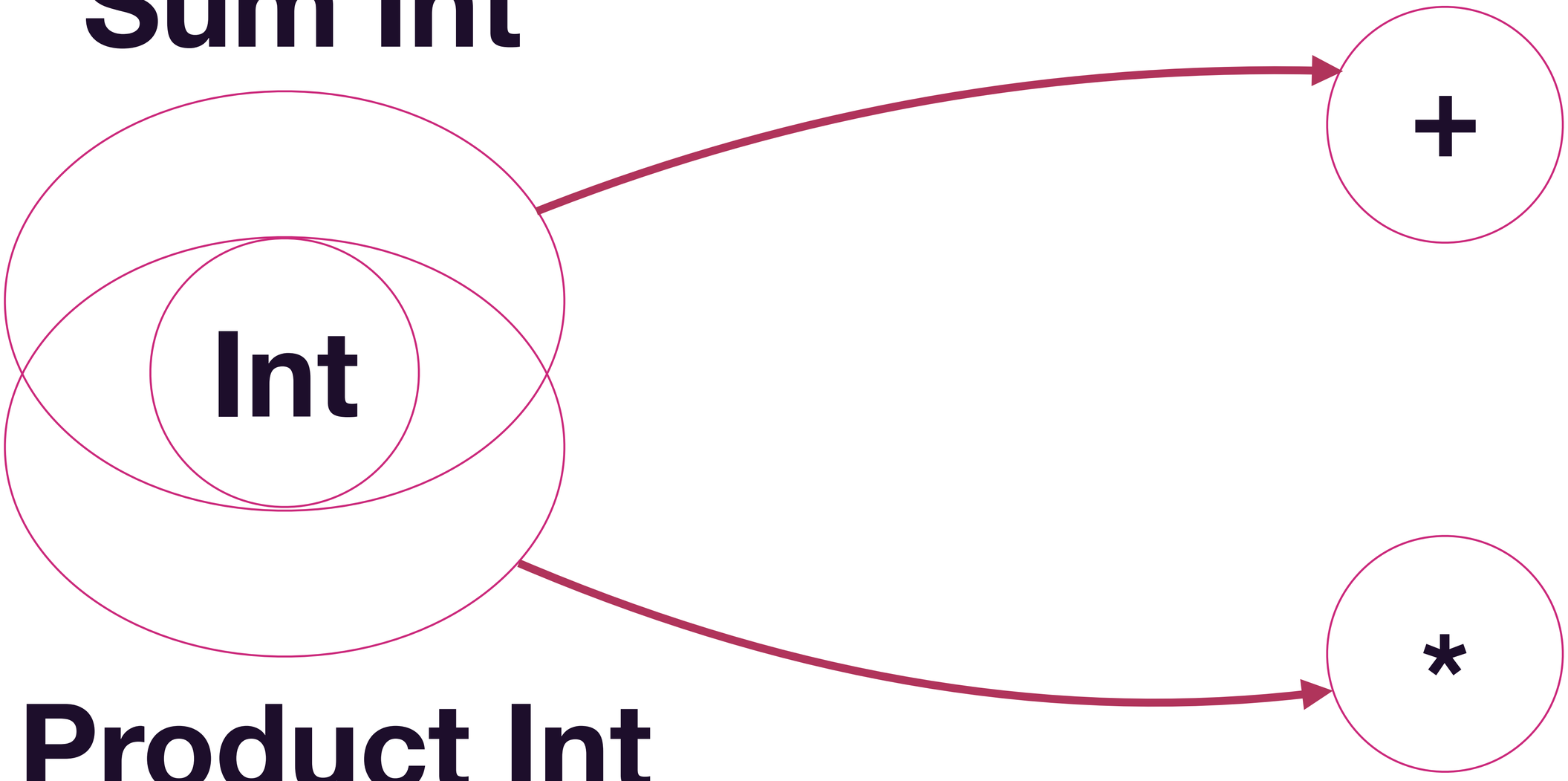
Sum Int



Sum Int

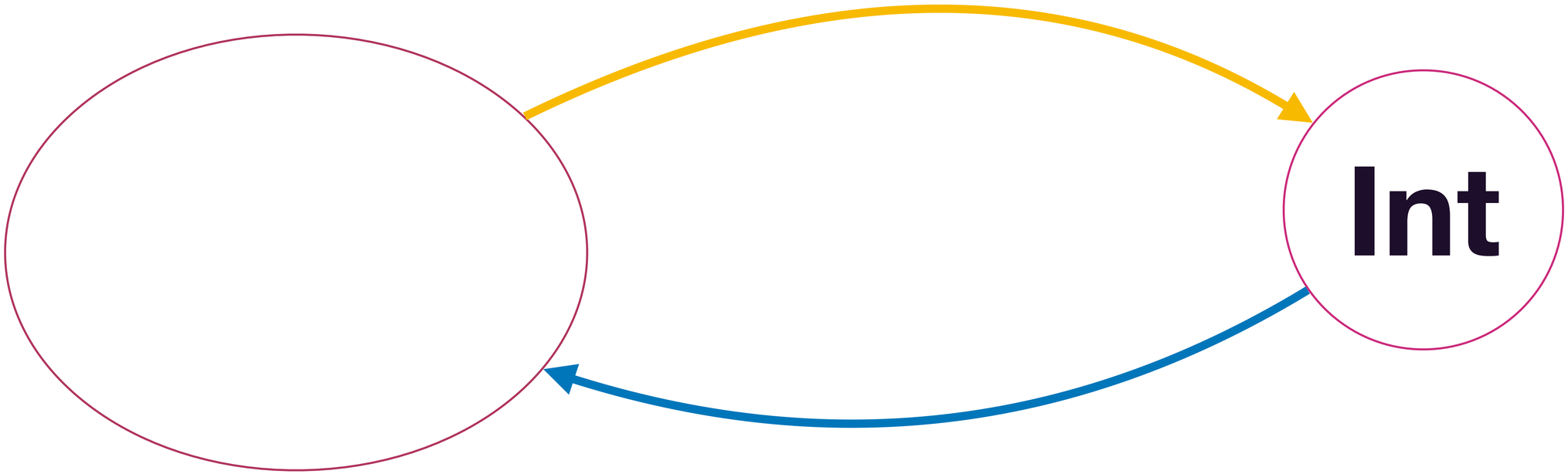


Sum Int



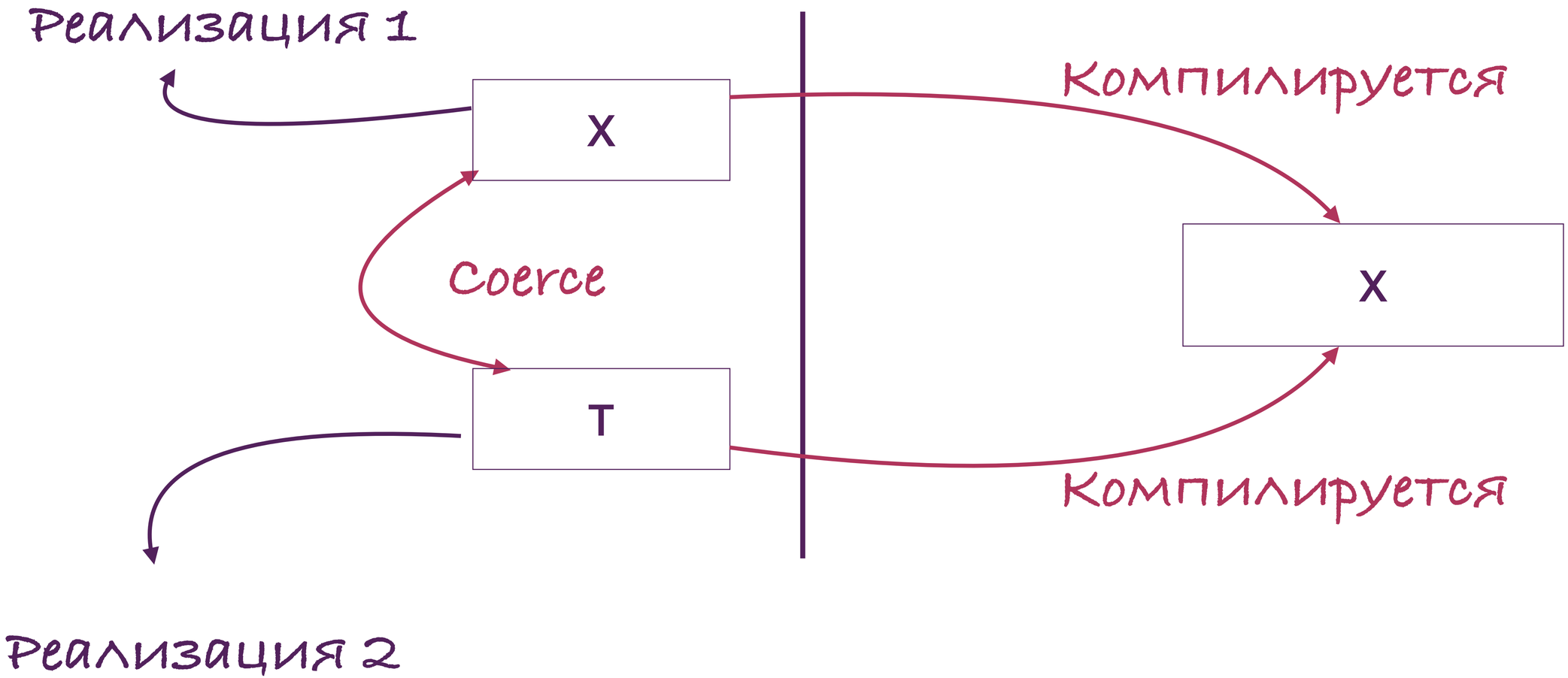
Product Int

Sum Int

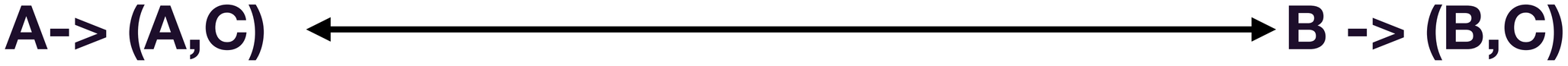


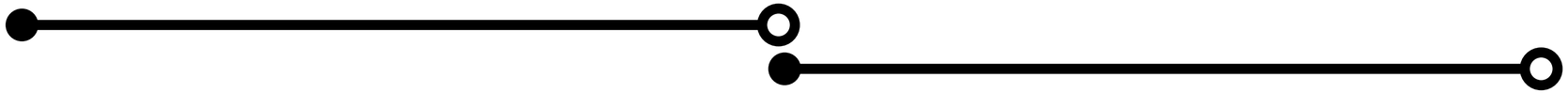
sumA + sumB

newtype T = T X

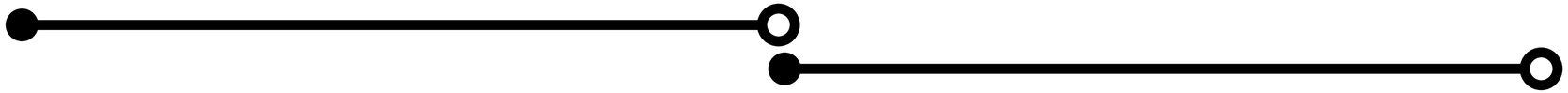

Реализация 1
X
Компилируется
Coerce
X
T
Компилируется
Реализация 2

```
coerce :: Coercible a b => a -> b
```

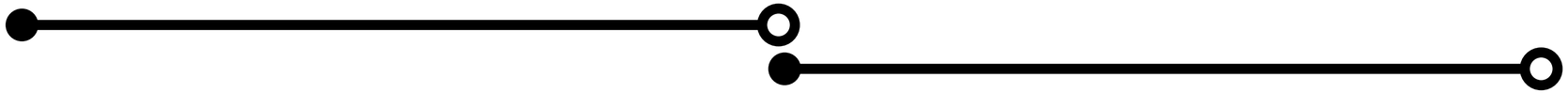




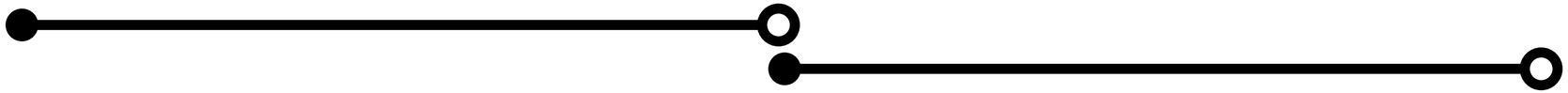
```haskell
instance Monoid A where
  mappend :: A -> A -> A
  mappend = Data.Coerce.coerce
    @(Sum a -> Sum a -> Sum a)
    @(A -> A -> A)
    mappend
```



```haskell
instance Monoid A where
  mappend :: A -> A -> A
  mappend = Data.Coerce.coerce
    @(Sum a -> Sum a -> Sum a)
    @(A -> A -> A)
    mappend
```



```haskell
instance Monoid A where
  mappend :: A -> A -> A
  mappend = Data.Coerce.coerce
    @(Sum a -> Sum a -> Sum a)
    @(A -> A -> A)
    mappend
```



```haskell
instance Monoid A where
  mappend :: A -> A -> A
  mappend = Data.Coerce.coerce
    @(Sum a -> Sum a -> Sum a)
    @(A -> A -> A)
    mappend
```

DerivingVia

Используем свой тип

```
data Foo a = Foo { unFoo :: a }
  deriving Monoid via (Sum a)
```

Обертка только для инстанса

```haskell
newtype ReadStatePure (m :: * -> *) (a :: *) = ReadStatePure (m a)
  deriving (Functor, Applicative, Monad)
```

```haskell
newtype ReadStatePure (m :: * -> *) (a :: *) = ReadStatePure (m a)
  deriving (Functor, Applicative, Monad)

instance
  HasState tag r m
  => HasReader tag r (ReadStatePure m) where
  local_ :: forall a.
    Proxy# tag
    -> (r -> r)
    -> ReadStatePure m a
    -> ReadStatePure m a
  local_ _ f = coerce @(m a -> m a) $ \m -> do
    r <- state @tag $ \r -> (r, f r)
    m <* put @tag r
```

```haskell
newtype ReadStatePure (m :: * -> *) (a :: *) = ReadStatePure (m a)
  deriving (Functor, Applicative, Monad)

instance
  HasState tag r m
  => HasReader tag r (ReadStatePure m) where
  local_ :: forall a.
    Proxy# tag
    -> (r -> r)
    -> ReadStatePure m a
    -> ReadStatePure m a
  local_ _ f = coerce @(m a -> m a) $ \m -> do
    r <- state @tag $ \r -> (r, f r)
    m <* put @tag r
```

```haskell
newtype ReadStatePure (m :: * -> *) (a :: *) = ReadStatePure (m a)
  deriving (Functor, Applicative, Monad)

instance
  HasState tag r m
  => HasReader tag r (ReadStatePure m) where
  local_ :: forall a.
    Proxy# tag
    -> (r -> r)
    -> ReadStatePure m a
    -> ReadStatePure m a
  local_ _ f = coerce @(m a -> m a) $ \m -> do
    r <- state @tag $ \r -> (r, f r)
    m <* put @tag r
```

```haskell
newtype ReadStatePure (m :: * -> *) (a :: *) = ReadStatePure (m a)
  deriving (Functor, Applicative, Monad)

instance
  HasState tag r m
  => HasReader tag r (ReadStatePure m) where
  local_ :: forall a.
    Proxy# tag
    -> (r -> r)
    -> ReadStatePure m a
    -> ReadStatePure m a
  local_ _ f = coerce @(m a -> m a) $ \m -> do
    r <- state @tag $ \r -> (r, f r)
    m <* put @tag r
```

```haskell
newtype ReadStatePure (m :: * -> *) (a :: *) = ReadStatePure (m a)
  deriving (Functor, Applicative, Monad)

instance
  HasState tag r m
  => HasReader tag r (ReadStatePure m) where
  local_ :: forall a.
    Proxy# tag
    -> (r -> r)
    -> ReadStatePure m a
    -> ReadStatePure m a
  local_ _ f = coerce @(m a -> m a) $ \m -> do
    r <- state @tag $ \r -> (r, f r)
    m <* put @tag r
```

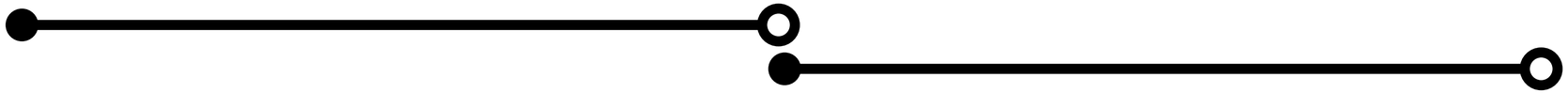
```haskell
newtype ReadStatePure (m :: * -> *) (a :: *) = ReadStatePure (m a)
  deriving (Functor, Applicative, Monad)

instance
  HasState tag r m
  => HasReader tag r (ReadStatePure m) where
  local_ :: forall a.
    Proxy# tag
    -> (r -> r)
    -> ReadStatePure m a
    -> ReadStatePure m a
  local_ _ f = coerce @(m a -> m a) $ \m -> do
    r <- state @tag $ \r -> (r, f r)
    m <* put @tag r
```

```haskell
newtype ReadStatePure (m :: * -> *) (a :: *) = ReadStatePure (m a)
  deriving (Functor, Applicative, Monad)

instance
  HasState tag r m
  => HasReader tag r (ReadStatePure m) where
  local_ :: forall a.
    Proxy# tag
    -> (r -> r)
    -> ReadStatePure m a
    -> ReadStatePure m a
  local_ _ f = coerce @(m a -> m a) $ \m -> do
    r <- state @tag $ \r -> (r, f r)
    m <* put @tag r
```

Исключение!

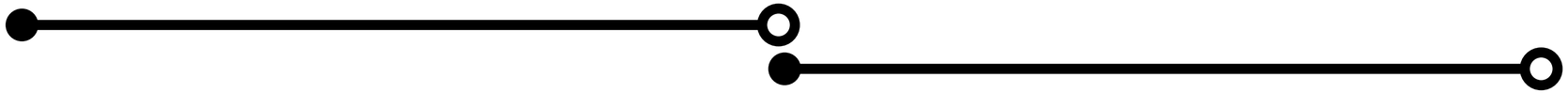


```haskell
newtype ReadState (m :: * -> *) (a :: *) = ReadState (m a)
  deriving (Functor, Applicative, Monad, MonadIO, PrimMonad)

instance
  (HasState tag r m, MonadMask m)
  => HasReader tag r (ReadState m) where
  local_ :: forall a.
    Proxy# tag
    -> (r -> r)
    -> ReadState m a
    -> ReadState m a
  local_ _ f = coerce @(m a -> m a) $ \action ->
      let
        setAndSave = state @tag $ \r -> (r, f r)
        restore r = put @tag r
      in
      bracket setAndSave restore $ \_ -> action
```

```haskell
newtype ReadState (m :: * -> *) (a :: *) = ReadState (m a)
  deriving (Functor, Applicative, Monad, MonadIO, PrimMonad)

instance
  (HasState tag r m, MonadMask m)
  => HasReader tag r (ReadState m) where
  local_ :: forall a.
    Proxy# tag
    -> (r -> r)
    -> ReadState m a
    -> ReadState m a
  local_ _ f = coerce @(m a -> m a) $ \action ->
      let
        setAndSave = state @tag $ \r -> (r, f r)
        restore r = put @tag r
      in
      bracket setAndSave restore $ \_ -> action
```



Стратегии для других библиотек

- SafeExceptions
- MonadUnliftIO
- MonadError

Стратегии вычисления

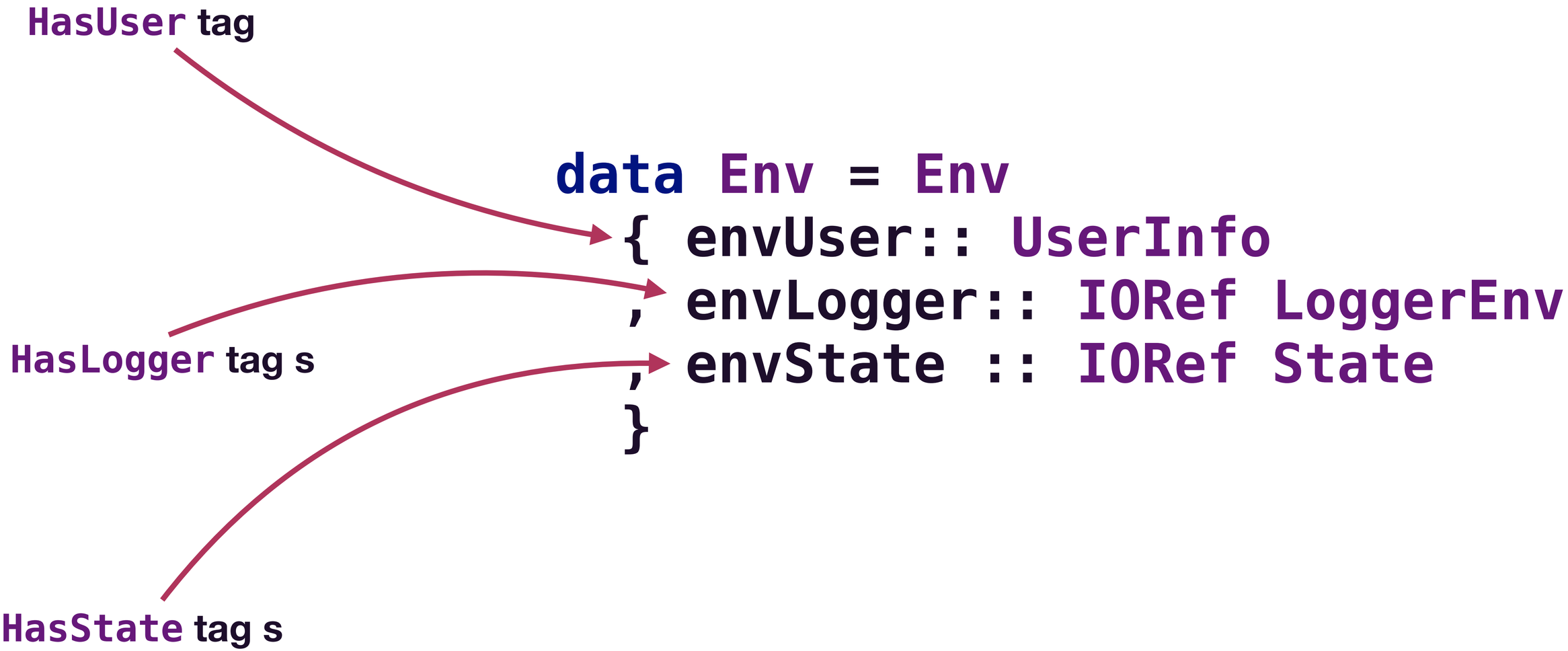
- StreamStack
- StreamDList
- StreamLog

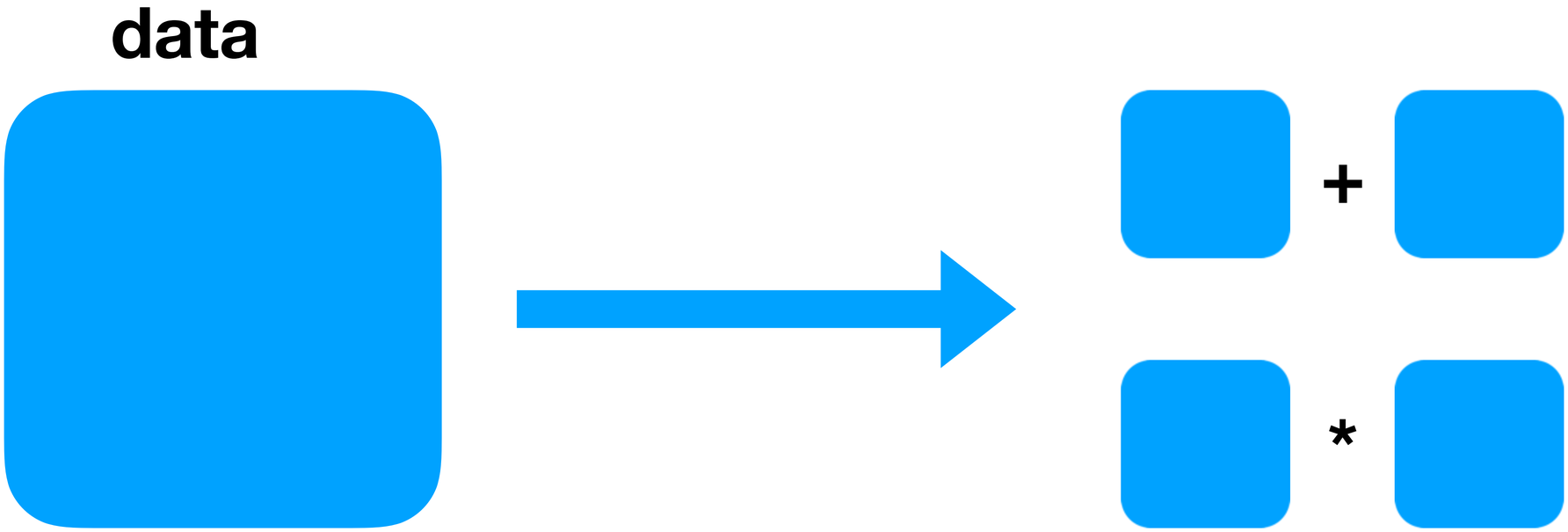
Как запускать?

```
newtype App = App { ... }
```

Transformers

ReaderT IO


HasUser tag
HasLogger tag s
HasState tag s
data Env = Env
{ envUser:: UserInfo
, envLogger:: IORef LoggerEnv
, envState :: IORef State
}


data
+
*

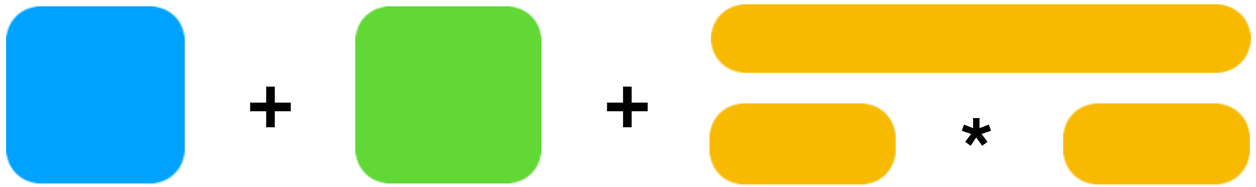
```
data Env
  = ErrorCrypto CryptoError
  | ErrorTransport TransportError
  | Running
      { _buffer :: IORef Buffer
      , _clients :: Map Address State
      }
```

```
data Env
  = ErrorCrypto CryptoError
  | ErrorTransport TransportError
  | Running
      { _buffer :: IORef Buffer
      , _clients :: Map Address State
      }
```

```
data Env
  = ErrorCrypto CryptoError
  | ErrorTransport TransportError
  | Running
      { _buffer :: IORef Buffer
      , _clients :: Map Address State
      }
```

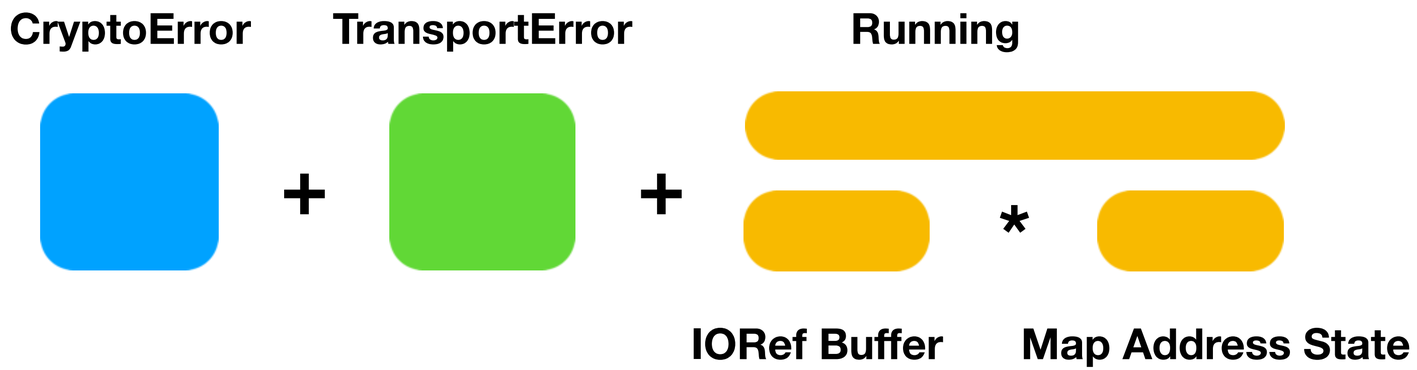
```
data Env
  = ErrorCrypto CryptoError
  | ErrorTransport TransportError
  | Running
    { _buffer :: IORef Buffer
    , _clients :: Map Address State
    }
```

```
data Env
  = ErrorCrypto CryptoError
  | ErrorTransport TransportError
  | Running
    { _buffer :: IORef Buffer
    , _clients :: Map Address State
    }
```



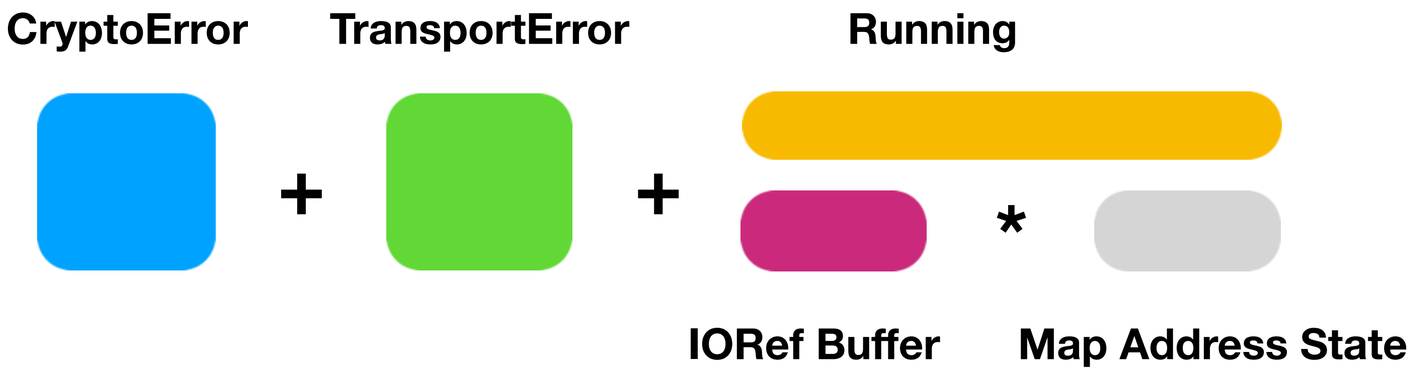
```
data Env
  = ErrorCrypto CryptoError
  | ErrorTransport TransportError
  | Running
      { _buffer :: IORef Buffer
      , _clients :: Map Address State
      }
```

Обобщенное представление



generic-lens

```
_Ctor @“CryptoError"
_Ctor @“TransportError"
_Ctor @“Running”. field @ “_buffer”
```



- Доступ к конструктору **`_Ctor`**
- Доступ к полю по имени **`field`**
- Доступ к полю по номеру **`pos`**

```haskell
newtype Field (field :: Symbol) (oldtag :: k) m
(a :: *) = Field (m a)
  deriving (Functor, Applicative, Monad)
```

```haskell
instance
  ( tag ~ field, HasField' field record v, HasReader oldtag record m )
  => HasReader tag v (Field field oldtag m)
  where
    local_ :: forall a.
      Proxy# tag
      -> (v -> v)
      -> Field field oldtag m a
      -> Field field oldtag m a
    local_ _ = coerce @((v -> v) -> m a -> m a) $
      local @oldtag . over (Generic.field' @field)
```

```haskell
newtype Field (field :: Symbol) (oldtag :: k) m
(a :: *) = Field (m a)
  deriving (Functor, Applicative, Monad)
```

```haskell
instance
  ( tag ~ field, HasField' field record v, HasReader oldtag record m )
  => HasReader tag v (Field field oldtag m)
  where
    local_ :: forall a.
      Proxy# tag
      -> (v -> v)
      -> Field field oldtag m a
      -> Field field oldtag m a
    local_ _ = coerce @((v -> v) -> m a -> m a) $
      local @oldtag . over (Generic.field' @field)
```

```
newtype Field (field :: Symbol) (oldtag :: k) m
(a :: *) = Field (m a)
  deriving (Functor, Applicative, Monad)
```

```
instance
  ( tag ~ field, HasField' field record v, HasReader oldtag record m )
  => HasReader tag v (Field field oldtag m)
  where
    local_ :: forall a.
      Proxy# tag
      -> (v -> v)
      -> Field field oldtag m a
      -> Field field oldtag m a
    local_ _ = coerce @((v -> v) -> m a -> m a) $
      local @oldtag . over (Generic.field' @field)
```

```haskell
newtype Field (field :: Symbol) (oldtag :: k) m
(a :: *) = Field (m a)
  deriving (Functor, Applicative, Monad)
```

```haskell
instance
  ( tag ~ field, HasField' field record v, HasReader oldtag record m )
  => HasReader tag v (Field field oldtag m)
  where
    local_ :: forall a.
      Proxy# tag
      -> (v -> v)
      -> Field field oldtag m a
      -> Field field oldtag m a
    local_ _ = coerce @((v -> v) -> m a -> m a) $
      local @oldtag . over (Generic.field' @field)
```

```haskell
newtype Field (field :: Symbol) (oldtag :: k) m
(a :: *) = Field (m a)
  deriving (Functor, Applicative, Monad)
```

```haskell
instance
  ( tag ~ field, HasField' field record v, HasReader oldtag record m )
  => HasReader tag v (Field field oldtag m)
  where
    local_ :: forall a.
      Proxy# tag
      -> (v -> v)
      -> Field field oldtag m a
      -> Field field oldtag m a
    local_ _ = coerce @((v -> v) -> m a -> m a) $
      local @oldtag . over (Generic.field' @field)
```

```
newtype Field (field :: Symbol) (oldtag :: k) m
(a :: *) = Field (m a)
  deriving (Functor, Applicative, Monad)
```

```
instance
  ( tag ~ field, HasField' field record v, HasReader oldtag record m )
  => HasReader tag v (Field field oldtag m)
  where
    local_ :: forall a.
      Proxy# tag
      -> (v -> v)
      -> Field field oldtag m a
      -> Field field oldtag m a
    local_ _ = coerce @((v -> v) -> m a -> m a) $
      local @oldtag . over (Generic.field' @field)
```

```
newtype Field (field :: Symbol) (oldtag :: k) m
(a :: *) = Field (m a)
  deriving (Functor, Applicative, Monad)
```

```
instance
  ( tag ~ field, HasField' field record v, HasReader oldtag record m )
  => HasReader tag v (Field field oldtag m)
  where
    local_ :: forall a.
      Proxy# tag
      -> (v -> v)
      -> Field field oldtag m a
      -> Field field oldtag m a
    local_ _ = coerce @((v -> v) -> m a -> m a) $
      local @oldtag . over (Generic.field' @field)
```

```haskell
newtype Field (field :: Symbol) (oldtag :: k) m
(a :: *) = Field (m a)
  deriving (Functor, Applicative, Monad)
```

```haskell
instance
  ( tag ~ field, HasField' field record v, HasReader oldtag record m )
  => HasReader tag v (Field field oldtag m)
  where
    local_ :: forall a.
      Proxy# tag
      -> (v -> v)
      -> Field field oldtag m a
      -> Field field oldtag m a
    local_ _ = coerce @((v -> v) -> m a -> m a) $
      local @oldtag . over (Generic.field' @field)
```

```haskell
data Foo = Foo { foo :: Int }

newtype MyReader a = MyReader (Reader Foo a)
  deriving (HasReader "foo" Int) via
    Field "foo" () (MonadReader (Reader Foo))
```

```haskell
data Foo = Foo { foo :: Int }

newtype MyReader a = MyReader (Reader Foo a)
  deriving (HasReader "foo" Int) via
    Field "foo" () (MonadReader (Reader Foo))
```

```haskell
data Foo = Foo { foo :: Int }

newtype MyReader a = MyReader (Reader Foo a)
  deriving (HasReader "foo" Int) via
    Field "foo" () (MonadReader (Reader Foo))
```

```haskell
data Foo = Foo { foo :: Int }

newtype MyReader a = MyReader (Reader Foo a)
  deriving (HasReader "foo" Int) via
    Field "foo" () (MonadReader (Reader Foo))
```

```haskell
data Foo = Foo { foo :: Int }

newtype MyReader a = MyReader (Reader Foo a)
  deriving (HasReader "foo" Int) via
    Field "foo" () (MonadReader (Reader Foo))
```

```
newtype Ctor (field :: Symbol) olgtag m (a :: *)

data MyError
  = ErrA String
  | ErrB String

newtype MyExcept a = MyExcept (Except MyError a)
  deriving (HasThrow ErrB String) via
     Ctor ErrB () (MonadError (Except MyError)
```

```
newtype Ctor (field :: Symbol) olgtag m (a :: *)

data MyError
  = ErrA String
  | ErrB String

newtype MyExcept a = MyExcept (Except MyError a)
  deriving (HasThrow ErrB String) via
     Ctor ErrB () (MonadError (Except MyError)
```

```
newtype Ctor (field :: Symbol) olgtag m (a :: *)

data MyError
  = ErrA String
  | ErrB String

newtype MyExcept a = MyExcept (Except MyError a)
  deriving (HasThrow ErrB String) via
     Ctor ErrB () (MonadError (Except MyError)
```

```
newtype Ctor (field :: Symbol) olgtag m (a :: *)

data MyError
  = ErrA String
  | ErrB String

newtype MyExcept a = MyExcept (Except MyError a)
  deriving (HasThrow ErrB String) via
     Ctor ErrB () (MonadError (Except MyError)
```

```
newtype Ctor (field :: Symbol) olgtag m (a :: *)

data MyError
  = ErrA String
  | ErrB String

newtype MyExcept a = MyExcept (Except MyError a)
  deriving (HasThrow ErrB String) via
     Ctor ErrB () (MonadError (Except MyError)
```

```
logic session = do
    context @Log (“session” .= session) $ do
      slots <- prioritise (request session)
                  <$> get @Reservation
      flip fix slots $ \next slots -> case
        [] -> send session NoSlots
        (x:xs) -> locking slot $ \s -> do
          context @Log (“slot” .= slot) $ do
            reply <- confirm session (ConfirmReq s)
            case reply of
              Confirm -> store session s
              Decline -> next xs
```

```
logic session = do
    context @Log (“session” .= session) $ do
      slots <- prioritise (request session)
                    <$> get @Reservation
      flip fix slots $ \next slots -> case
        [] -> send session NoSlots
        (x:xs) -> locking slot $ \s -> do
          context @Log (“slot” .= slot) $ do
            reply <- confirm session (ConfirmReq s)
            case reply of
              Confirm -> store session s
              Decline -> next xs
```

```
logic session = do
    context @Log (“session” .= session) $ do
      slots <- prioritise (request session)
                 <$> get @Reservation
      flip fix slots $ \next slots -> case
        [] -> send session NoSlots
        (x:xs) -> locking slot $ \s -> do
          context @Log (“slot” .= slot) $ do
            reply <- confirm session (ConfirmReq s)
            case reply of
              Confirm -> store session s
              Decline -> next xs
```

```
logic session = do
    context @Log (“session” .= session) $ do
      slots <- prioritise (request session)
                  <$> get @Reservation
      flip fix slots $ \next slots -> case
        [] -> send session NoSlots
        (x:xs) -> locking slot $ \s -> do
          context @Log (“slot” .= slot) $ do
            reply <- confirm session (ConfirmReq s)
            case reply of
              Confirm -> store session s
              Decline -> next xs
```

```
logic session = do
    context @Log (“session” .= session) $ do
      slots <- prioritise (request session)
                  <$> get @Reservation
      flip fix slots $ \next slots -> case
        [] -> send session NoSlots
        (x:xs) -> locking slot $ \s -> do
          context @Log (“slot” .= slot) $ do
            reply <- confirm session (ConfirmReq s)
            case reply of
              Confirm -> store session s
              Decline -> next xs
```

```
logic session = do
    context @Log (“session” .= session) $ do
      slots <- prioritise (request session)
                  <$> get @Reservation
      flip fix slots $ \next slots -> case
        [] -> send session NoSlots
        (x:xs) -> locking slot $ \s -> do
          context @Log (“slot” .= slot) $ do
            reply <- confirm session (ConfirmReq s)
            case reply of
              Confirm -> store session s
              Decline -> next xs
```

```
logic session = do
    context @Log (“session” .= session) $ do
      slots <- prioritise (request session)
                  <$> get @Reservation
      flip fix slots $ \next slots -> case
        [] -> send session NoSlots
        (x:xs) -> locking slot $ \s -> do
          context @Log (“slot” .= slot) $ do
            reply <- confirm session (ConfirmReq s)
            case reply of
              Confirm -> store session s
              Decline -> next xs
```

```
logic session = do
    context @Log (“session” .= session) $ do
      slots <- prioritise (request session)
                  <$> get @Reservation
      flip fix slots $ \next slots -> case
        [] -> send session NoSlots
        (x:xs) -> locking slot $ \s -> do
          context @Log (“slot” .= slot) $ do
            reply <- confirm session (ConfirmReq s)
            case reply of
              Confirm -> store session s
              Decline -> next xs
```

```
logic session = do
    context @Log (“session” .= session) $ do
      slots <- prioritise (request session)
                  <$> get @Reservation
      flip fix slots $ \next slots -> case
        [] -> send session NoSlots
        (x:xs) -> locking slot $ \s -> do
          context @Log (“slot” .= slot) $ do
            reply <- confirm session (ConfirmReq s)
            case reply of
              Confirm -> store session s
              Decline -> next xs
```

```
logic session = do
    context @Log ("session" .= session) $ do
      slots <- prioritise (request session)
                 <$> get @Reservation
      flip fix slots $ \next slots -> case
        [] -> send session NoSlots
        (x:xs) -> locking slot $ \s -> do
          context @Log ("slot" .= slot) $ do
            reply <- confirm session (ConfirmReq s)
            case reply of
              Confirm -> store session s
              Decline -> next xs
```

1. Определяем свойства

```
class Process tag protocol m where
   confirm  :: Proxy# tag
            -> Session protocol
            -> Request protocol
            -> m (Reply protocol)

class Reservation tag m where
   locking :: Proxy# tag -> Slot -> (ReservedSlot -> m ())
   order :: Proxy# tag -> ReservedSlot -> User -> m ()


class Logger tag m where
   log :: Proxy# tag -> LogLevel -> Message -> m ()
   context :: Proxy# tag -> ContextEntry -> m () -> m ()
```

2. Пишем стратегии

Определение работы через примитивы

Тестовая среда

Различные стратегии вычислений

3. Комбинируем интерпретатор

Весь бойлерплейт тут

```
data Env = Env
  { reservation :: IORef ReservationState
  , process :: ProcessHandle
  }
newtype M a = M { runM :: ReaderT IO Env a }
  deriving (Functor, Applicative, Monad)
  deriving Process “reservation”
    via HasProcess (Field “process” () (ReaderState Env))
  deriving Reservation Reservation
    via (ReservationState (Field “reservation” () (Reader Env))
```

3. Комбинируем интерпретатор

```
data Env = Env
  { reservation :: IORef ReservationState
  , process :: ProcessHandle
  }
newtype M a = M { runM :: ReaderT IO Env a }
  deriving (Functor, Applicative, Monad)
  deriving Process “reservation”
    via HasProcess (Field “process”  () (ReaderState Env))
  deriving Reservation Reservation
    via (ReservationState (Field “reservation” () (Reader Env))
```

3. Комбинируем интерпретатор

```
data Env = Env
  { reservation :: IORef ReservationState
  , process :: ProcessHandle
  }
newtype M a = M { runM :: ReaderT IO Env a }
  deriving (Functor, Applicative, Monad)
  deriving Process “reservation”
    via HasProcess (Field “process” () (ReaderState Env))
  deriving Reservation Reservation
    via (ReservationState (Field “reservation” () (Reader Env))
```

3. Комбинируем интерпретатор

```
data Env = Env
  { reservation :: IORef ReservationState
  , process :: ProcessHandle
  }
newtype M a = M { runM :: ReaderT IO Env a }
  deriving (Functor, Applicative, Monad)
  deriving Process “reservation”
    via HasProcess (Field “process” () (ReaderState Env))
  deriving Reservation Reservation
    via (ReservationState (Field “reservation” () (Reader Env))
```

3. Комбинируем интерпретатор

```
data Env = Env
  { reservation :: IORef ReservationState
  , process :: ProcessHandle
  }
newtype M a = M { runM :: ReaderT IO Env a }
  deriving (Functor, Applicative, Monad)
  deriving Process “reservation”
    via HasProcess (Field “process” () (ReaderState Env))
  deriving Reservation Reservation
    via (ReservationState (Field “reservation” () (Reader Env))
```

3. Комбинируем интерпретатор

```
data Env = Env
  { reservation :: IORef ReservationState
  , process :: ProcessHandle
  }
newtype M a = M { runM :: ReaderT IO Env a }
  deriving (Functor, Applicative, Monad)
  deriving Process “reservation”
    via HasProcess (Field “process” () (ReaderState Env))
  deriving Reservation Reservation
    via (ReservationState (Field “reservation” () (Reader Env))
```

3. Комбинируем интерпретатор

```
data Env = Env
  { reservation :: IORef ReservationState
  , process :: ProcessHandle
  }
newtype M a = M { runM :: ReaderT IO Env a }
  deriving (Functor, Applicative, Monad)
  deriving Process “reservation”
    via HasProcess (Field “process” ()) (ReaderState Env))
  deriving Reservation Reservation
    via (ReservationState (Field “reservation” () (Reader Env))
```

Capability

- Реализация MTL с использованием современных средств
- Ещё одно направление в пространстве поиска решений