Type-safe Embedded Domain-Specific Languages

Arthur Xavier

@arthurxavierx

1. Domain-specific languages

Domain modelling and DSLs
What makes a (good) DSL?
Examples

2. Language Oriented Programming

Playing with a real-world DSL
Discuss the techniques used
Language Oriented Programming

3. Type-safe embedding

Embedding techniques
Type-safety for DSLs
A DSL for validating business rules

4. Da Capo al Coda

A DSL for chatbots with indexed monads
Modifying the chatbot DSL
Wrap-up and conclusion

1

Domain-specific languages

Domains come in all shapes and colors

Testing	Infotaiment systems
Validation	Mobile app development
Financial services	Web forms
Storage	Data visualization
Database querying	Text documents
GUI development	3D graphics
Voice controllers	Architectural modelling
…	…

And it’s our job to translate all of this into code

Language is the essence of abstraction

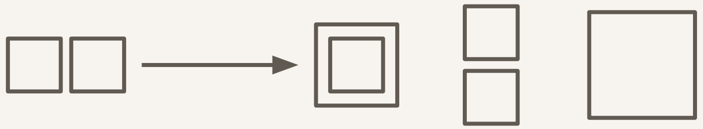
$$1 + 2 + 3 + 4$$

$$1 + 2 + 3 + \ldots$$

$$1 + 2 + 3 + 4 \longrightarrow \begin{array}{l} 1 + 2 + 3 + 4 \\ 10 \\ 3 + 3 + 4 \\ 1 + (2 + (3 + 4)) \\ \square \end{array}$$

$$1 + 2 + 3 + \ldots \longrightarrow \begin{array}{l} \infty \\ 10 \\ 6 \\ ((1 + 2) + 3) + \ldots \\ \square \end{array}$$

□□



$\text{Language} \stackrel{\text{def}}{=} \text{syntax} + \text{semantics}$

$\text{DSL} \overset{\text{def}}{=} \text{domain} + \text{language}$

$$\text{DSL} \overset{\text{def}}{=} \text{model} + \text{language}$$

$$\text{DSL} \stackrel{\text{def}}{=} \text{model} + \text{syntax} + \text{semantics}$$

DSL $\overset{\text{def}}{=}$ model + syntax + semantics

```
data Syntax = ...

semantics :: Syntax -> _
```

DSL $\stackrel{\text{def}}{=}$ model + syntax + semantics

```haskell
data Expr -- Abstract Syntax Tree
   = Val Bool
   | And Expr Expr
   | Or Expr Expr
   | Bla Expr

eval :: Expr -> Bool
eval (Val x) = x
eval (And a b) = eval a && eval b
eval (Or a b) = eval a || eval b
eval (Bla a) = not (eval a)
```

DSL $\stackrel{\text{def}}{=}$ model + syntax + semantics

```
data Expr
  = Val Bool
  | And Expr Expr
  | Or Expr Expr
  | Bla Expr
```

```
print :: Expr -> String
print (Val x) = show x
print (And a b) = "(" ++ print a ++ " ∧ " ++ print b ++ ")"
print (Or a b) = "(" ++ print a ++ " ∨ " ++ print b ++ ")"
print (Bla a) = "¬" ++ print a
```

$$\text{DSL} \overset{\text{def}}{=} \text{primitives} + \text{composition} + \text{interpretation}$$

DSL $\overset{\text{def}}{=}$ primitives + composition + interpretation

```
data Primitives = ...

combinator :: _ -> Primitives -> Primitives

interpreter :: Primitives -> _
```

DSL $\overset{\text{def}}{=}$ primitives + composition + interpretation

```haskell
data Contract
  = Transfer Person Person Money DateTime
  | Sell Person Person Product DateTime
  | Sequence Contract Contract
  | Freeze Contract
  | Cancel Contract
  | ...

calculate :: Contract -> Money
perform :: Contract -> IO ()
simulate :: Contract -> World
validate :: Contract -> Maybe ContractError
```

The goal is to encode domain rules in both syntax and semantics

The goal is to encode invariants in both syntax and semantics

The human factor

Syntax plays a big role
(it’s what us humans manipulate)

Correctness by construction is important

A good DSL

Simple

Concise

Conforming

Composable

Correct

A good DSL

Expresses problems using a specific vocabulary

Gives us simple, composable words

Lets us build up larger and correct systems
“in our own words”

Why?

To make things simple

To make things pretty

To make things fast

To make things correct

Any combination of the above

Examples

External DSLs

HTML

CSS

SQL

LaTeX

Makefile

VimL

Elm

Dhall

Solidity

Examples

Embedded DSLs in Haskell

Parsec

HSpec

Persistent’s Entity Syntax

Esqueleto

Servant routes

```haskell
recipe :: Parser Recipe
recipe = do
    rn <- lexeme stringLike
    lexeme (syntacticSugar "is made with") *> string "\r\n"
    i <- many1 ingredient
    many1 (string "\r\n")
    lexeme (string "prepared by") *> string "\r\n"
    s <- many1 step
    return $ Recipe rn i s
```

https://www.schoolofhaskell.com/user/adinapoli/the-pragmatic-haskeller/episode-5-a-simple-dsl

```haskell
mySpec :: Spec
mySpec = do
  describe "Prelude.head" $ do
    it "returns the first element of a list" $ do
      head [23 ..] `shouldBe` (23 :: Int)

    it "returns the first element of an *arbitrary* list" $ do
      property $ \x xs ->
        head (x:xs) == (x :: Int)

    it "throws an exception if used with an empty list" $ do
      evaluate (head []) `shouldThrow` anyException
```

https://hspec.github.io/

```
share [mkPersist sqlSettings, mkMigrate "migrateAll"] [persist|

  Person
    name String
    age Int Maybe
    deriving Eq Show
  BlogPost
    title String
    authorId PersonId
    deriving Eq Show
  Follow
    follower PersonId
    followed PersonId
    deriving Eq Show

|]
```

https://www.yesodweb.com/book/persistent

```haskell
recentArticles :: SqlPersistT m [(Entity User, Entity Article)]
recentArticles =
  select . from $ \(users `InnerJoin` articles) -> do
    on (users ^.UserId ==. articles ^.ArticleAuthorId)
    orderBy [desc (articles ^.ArticlePublishedTime)]
    limit 10
    return (users, articles)
```

https://mmhaskell.com/web-skills-5

```haskell
type UserAPI
  = "users"
    :> ReqBody '[JSON] User
    :> Post '[JSON] User
  :<|>
    "users"
    :> Capture "userId" Integer
    :> ReqBody '[JSON] User
    :> Put '[JSON] User
```

https://docs.servant.dev/en/stable/tutorial/ApiType.html

```haskell
type UserAPI
  = "users"
    ( ReqBody '[JSON] User
      :> Post '[JSON] User
    :<|>
      Capture "userId" Integer
      :> ReqBody '[JSON] User
      :> Put '[JSON] User
    )
```

https://docs.servant.dev/en/stable/tutorial/ApiType.html

“If you have a set of things and a means of combining them, then it’s a language.”

https://parametri.city/blog/2018-12-23-language-oriented-software-engineering

Questions?

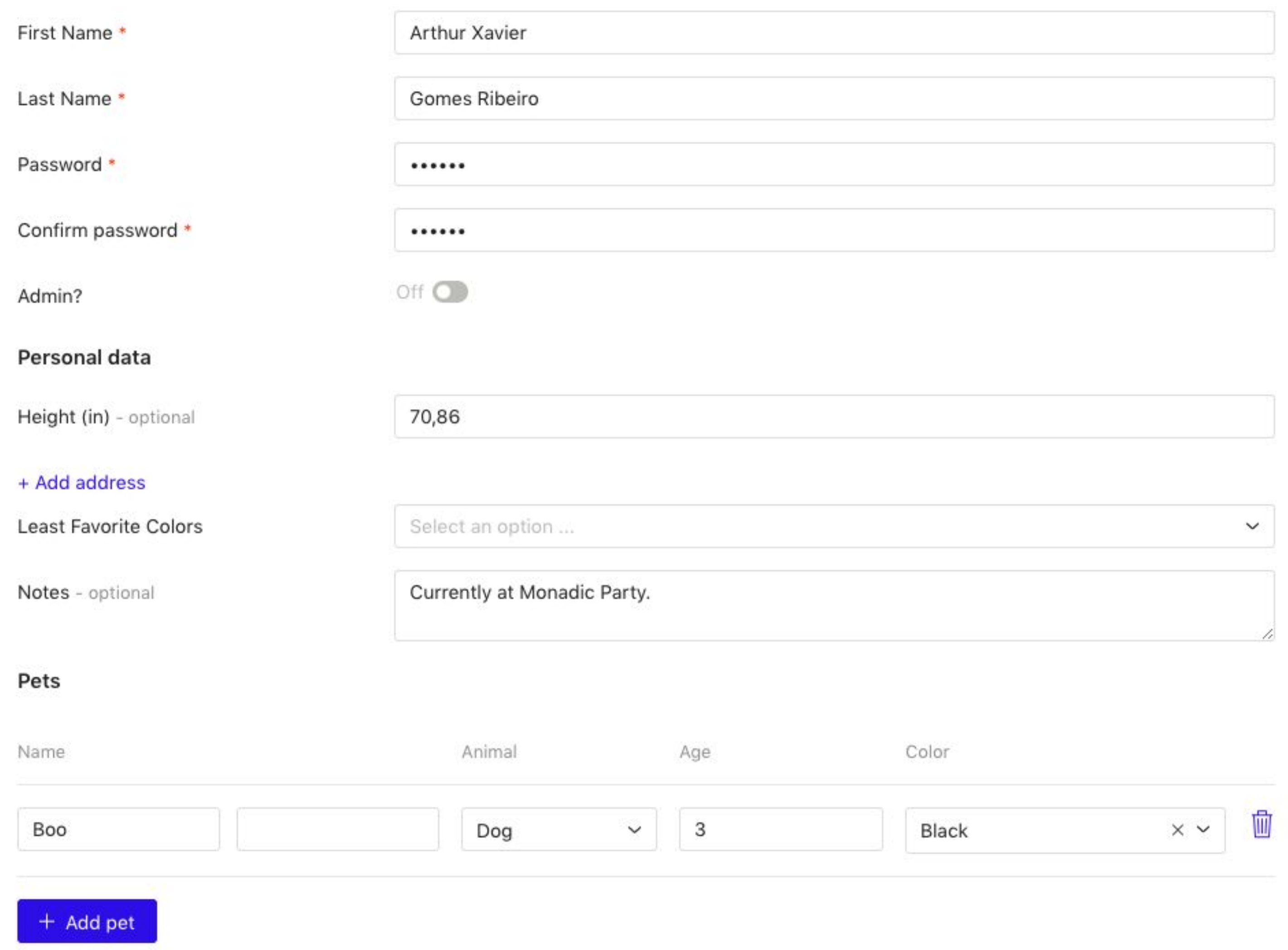
2

Language oriented programming

A DSL for forms

https://github.com/lumihq/purescript-lumi-components

https://lumihq.github.io/purescript-lumi-components/#/form



First Name *	Arthur Xavier
Last Name *	Gomes Ribeiro
Password *	••••••
Confirm password *	••••••
Admin?	Off

Personal data

Height (in) - optional	70,86

+ Add address

Least Favorite Colors	Select an option ...
Notes - optional	Currently at Monadic Party.

Pets

Name		Animal	Age	Color
Boo		Dog	3	Black

+ Add pet

```
$ git clone https://github.com/arthurxavierx/monadic-party-edsl.git
$ cd monadic-party-edsl/forms
$ make watch
```

```haskell
newtype Registration = Registration
  { email :: EmailAddress
  , password :: NonEmptyString
  }
```

```haskell
registrationForm :: FormBuilder _ RegistrationFormData Registration
registrationForm = ado
  email <-
    indent "Email" Required
    $ focus _email
    $ validated (isValidEmail "Email")
    $ validated (nonEmpty "Email")
    $ textbox
  password <-
    indent "Password" Required
    $ focus _password
    $ validated (nonEmpty "Password")
    $ passwordBox
  in
    Registration
      { email
      , password
      }
```

```
type RegistrationFormData =
  { email :: Validated String
  , password :: Validated String
  }

_email :: forall a r. Lens' { email :: a | r } a
_email = lens _.email _{ email = _ }

_password :: forall a r. Lens' { password :: a | r } a
_password = lens _.password _{ password = _ }

isValidEmail :: Validator String EmailAddress
-- isValidEmail :: String -> Either String EmailAddress
```

```
registrationForm :: FormBuilder _ RegistrationFormData Registration
registrationForm = ado
  email <-
    indent "Email" Required
    $ focus (lens _.email _{ email = _ })
    $ validated (isValidEmail "Email")
    $ validated (nonEmpty "Email")
    $ textbox
  password <-
    indent "Password" Required
    $ focus (lens _.password _{ password = _ })
    $ validated (nonEmpty "Password")
    $ passwordBox
  in
    Registration
      { email
      , password
      }
```

```
email <-
  indent "Email" Required
  $ focus (prop (SProxy :: _ "email"))
  $ validated (isValidEmail "Email")
  $ textbox
password <-
  indent "Password" Required
  $ focus (prop (SProxy :: _ "password"))
  $ validated (nonEmpty "Password")
  $ passwordBox
_ <- withValue \{ password } ->
  indent "Password confirmation" Required
  $ focus _passwordConfirmation
  $ validated (\pc ->
      if pc == fromValidated password then
        Right pc
      else
        Left "Passwords do not match."
    )
  $ passwordBox
in
  Registration
    { email
    , password
    }
```

How does it work?

```
newtype FormBuilder value result = FormBuilder
  ( value
    -> { edit :: ((value -> value) -> Effect Unit) -> UI
       , validate :: Maybe result
       }
  )

instance Applicative (FormBuilder props value)
```

```
newtype FormBuilder props value result = FormBuilder
  ( props
    -> value
    -> { edit :: ((value -> value) -> Effect Unit) -> UI
       , validate :: Maybe result
       }
  )

instance Applicative (FormBuilder props value)
```

```
type FormUI value = ((value -> value) -> Effect Unit) -> UI

newtype FormBuilder props value result =
  FormBuilder
    (ReaderT (Tuple props value) (WriterT FormUI Maybe) result)

instance Applicative (FormBuilder props value)
```

```
textbox :: forall props. FormBuilder props String String

passwordBox :: forall props. FormBuilder props String String

switch :: forall props. FormBuilder props Boolean Boolean
```

```haskell
indent
  :: forall props value result
   . String
  -> RequiredField
  -> FormBuilder props value result
  -> FormBuilder props value result

focus
  :: forall props s a result
   . Lens' s a
  -> FormBuilder props a result
  -> FormBuilder props s result
```

```
validated
  :: forall props value result_ result
   . (result_ -> Either String result)
  -> FormBuilder props value result_
  -> FormBuilder props (Validated value) result

withValue
  :: forall props value result
   . (value -> FormBuilder props value result)
  -> FormBuilder props value result
```

```
build
  :: forall props value result
   . FormBuilder props value result
  -> { value :: value
     , onChange :: (value -> value) -> Effect Unit
     | props
     }
  -> JSX

revalidate
  :: forall props value result
   . FormBuilder props value result
  -> props
  -> value
  -> Maybe result
```

Questions?

Multiple DSLs for building complex forms

```
newtype Wizard props value result =
  Wizard
    (Free (FormBuilder props value) result)

derive newtype instance Monad (Wizard props value)

step
  :: forall props value result
   . FormBuilder props value result
  -> Wizard props value result
```

```
newtype TableFormBuilder props value result = ...

instance Applicative (TableFormBuilder props value)

column
  :: forall props row result
   . String
  -> FormBuilder props row result
  -> TableFormBuilder props row result

table
  :: forall props row result
   . TableFormBuilder props row result
  -> FormBuilder props (Array row) (Array result)
```

Language oriented programming

```
interpreterA :: LanguageA -> LanguageB

interpreterB :: LanguageB -> LanguageC

interpreterC :: LanguageC -> LanguageD
```

...

Language oriented programming

Design a domain-specific language for the core application logic

Write the application in the DSL

Build interpreters to execute the DSL programs

Language oriented programming

Abstracting business problems as programming language problems, so that solutions are DSLs

How to abstract things ⇔ how to split things up and join them back together

Language building $\cong$ domain modelling

“[...] you cannot know what the DSL will be ahead of time, you have to evolve it alongside the concrete implementation.”

https://parametri.city/blog/2018-12-23-language-oriented-software-engineering

Questions?

3

Type-safe embedding

DSL $\stackrel{\text{def}}{=}$ model + syntax + semantics

```
data Syntax = ...

semantics :: Syntax -> _
```

DSL $\stackrel{\text{def}}{=}$ primitives + composition + interpretation

```
data Primitives = ...

combinator :: _ -> Primitives -> Primitives

interpreter :: Primitives -> _
```

Feasting on the host language

Expressions

Control flow

Data types

Effects

Recursion, unfortunately?

Shallow × deep embedding

Shallow × deep embedding

Who does the work, interpreters or constructors?

Extending: new interpretations or new constructors?

Deep is simple, but shallow is direct

```
newtype FormBuilder props value result = FormBuilder
  ( props
    -> value
    -> { edit :: ((value -> value) -> Effect Unit) -> UI
       , validate :: Maybe result
       }
  )
```

Shallow embedding

Shallow embedding

Syntax is defined in terms of the semantic domain

More flexibility for adding new combinators
(under a specific interpretation)

Adding a new interpreter might imply:

- adding some new set of constructors
- and/or refactoring the semantic domain to include the new interpretation, then rework all constructors

```haskell
data FormBuilder props value result where
  Textbox :: FormBuilder props String String
  Password :: FormBuilder props String String
  ...
  Focus
    :: Lens' s a
    -> FormBuilder props a result
    -> FormBuilder props s result
  ...
```

Deep embedding

Deep embedding

More flexibility for adding new interpreters
(for a specific set of constructors)

Adding new constructors might imply reworking all interpreters

“[…] The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety.”

Expression problem

https://en.wikipedia.org/wiki/Expression_problem

Monoids

```haskell
class Semigroup a where
  (<>) :: a -> a -> a

class Monoid a where
  mempty :: a
```

```haskell
data Document = Empty | Block [Block] | Inline [Inline]
data Block = Button String | Header String | Paragraph Inline | ...
data Inline = Text String | Strong String | Link URL String | ...

instance Semigroup Document where
  Empty <> Empty = Empty
  Empty <> Block b = b
  Block b <> Empty = b
  ...
  Block (Button l) <> Inline (Text s) = Block (Button (l <> s))
  ...

instance Monoid Document where
  mempty = Empty
```

```haskell
myDocument :: Document
myDocument =
  fold
    [ Block
        [ Header "Hello, world!"
        , Paragraph "Lorem ipsum dolor sit amet, consectetur ..."
        , Button "Click me"
        ]
    , Block
        [ Paragraph "Look! An image!"
        , Image "https://whatever.com/whatever.jpg"
        ]
    ]
```

```haskell
{-# LANGUAGE RebindableSyntax #-}

myDocument :: Document
myDocument = do
  Block
    [ Header "Hello, world!"
    , Paragraph "Lorem ipsum dolor sit amet, consectetur ..."
    , Button "Click me"
    ]
  Block
    [ Paragraph "Look! An image!"
    , Image "https://whatever.com/whatever.jpg"
    ]
  where
    (>>) = (<>)
```

Applicative functors

```haskell
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  --    :: f a         -> f b -> f (a, b)
```

```haskell
data AppConfig = AppConfig
  { hostname :: String
  , port :: Int
  , emailKey :: String
  , emailPassword :: String
  }

appConfig :: EnvConfig AppConfig
appConfig =
  AppConfig
  <$> string "hostname"
  <*> int "port"
  <*> string "emailKey"
  <*> string "emailPassword"
```

```haskell
{-# LANGUAGE ApplicativeDo #-}

appConfig :: EnvConfig AppConfig
appConfig = do
  hostname <- string "hostname"
  port <- int "port"
  emailKey <- string "emailKey"
  emailPassword <- string "emailPassword"
  pure $
    AppConfig
      { hostname = hostname
      , port = port
      , emailKey = emailKey
      , emailPassword = emailPassword
      }
```

Monads

```
class Applicative m => Monad m where
  bind :: m a -> (a -> m b) -> m b
```

```haskell
data AppConfig = AppConfig
  { hostname :: String
  , port :: Int
  , emailKey :: String
  , emailPassword :: String
  , emailDefaultFrom :: Maybe String
  }
```

```haskell
appConfig :: EnvConfig AppConfig
appConfig = do
  hostname <- string "hostname"
  port <- int "port"
  emailKey <- string "emailKey"
  emailPassword <- string "emailPassword"
  emailDefaultFrom <-
    if isJust emailKey then
      Just <$> string "emailDefaultFrom"
    else
      pure Nothing
  pure $
    AppConfig
      { hostname = hostname
      , port = port
      , emailKey = emailKey
      , emailPassword = emailPassword
      , emailDefaultFrom = emailDefaultFrom
      }
```

“[…] The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety.”

Expression problem

https://en.wikipedia.org/wiki/Expression_problem

Tagless final

Tagless final

Typeclasses as syntax

Instances as semantics

```haskell
class Monoid d => MathDoc d where
  text :: String -> d
  (-) :: d -> d -> d    -- subscripting
  (^) :: d -> d -> d    -- superscripting
  (/) :: d -> d -> d    -- fraction
```

http://okmij.org/ftp/tagless-final/MathLayout.hs

```haskell
data LaTeX = LaTeX String

instance Monoid LaTeX ...

instance MathDoc LaTeX where
  text = LaTeX
  LaTeX a - LaTeX b = LaTeX (a ++ "_{" ++ b ++ "}")
  LaTeX a ^ LaTeX b = LaTeX (a ++ "^{" ++ b ++ "}")
  LaTeX a / LaTeX b = LaTeX ("\\frac{" ++ a ++ "}{" ++ b ++ "}")
```

http://okmij.org/ftp/tagless-final/MathLayout.hs

Tagless final

Vertical extensibility: adding new interpreters

Horizontal extensibility: adding new terms

```haskell
data LaTeX = LaTeX String

instance MathDoc LaTeX where
  text = LaTeX
  LaTeX a - LaTeX b = LaTeX (a ++ "_{" ++ b ++ "}")
  LaTeX a ^ LaTeX b = LaTeX (a ++ "^{" ++ b ++ "}")
  LaTeX a / LaTeX b = LaTeX ("\\frac{" ++ a ++ "}{" ++ b ++ "}")

instance MathDoc String where
  text = id
  a - b = a ++ "_" ++ b
  a ^ b = a ++ "^" ++ b
  a / b = a ++ "/" ++ b
```

http://okmij.org/ftp/tagless-final/MathLayout.hs

```haskell
class MathDoc d where
  text :: String -> d
  (-) :: d -> d -> d    -- subscripting
  (^) :: d -> d -> d    -- superscripting
  (/) :: d -> d -> d    -- fraction

data GreekLetter d = GreekLetter Char
alpha = greek $ GreekLetter 'α'

class MathDoc d => Greek d where
  greek :: GreekLetter -> d

class MathDoc d => Circle d where
  circle :: d
```

http://okmij.org/ftp/tagless-final/MathLayout.hs

```haskell
-- doc1 :: (Circle d, Greek d) => d
doc1 = alpha - (text "1") ^ circle

doc1_string :: String
doc1_string = doc1
-- > "α_1^∘"

doc1_latex :: LaTeX
doc1_latex = doc1
-- > LaTeX "\\alpha_1^\\circ"
```

http://okmij.org/ftp/tagless-final/MathLayout.hs

Tagless final

Composing constraints ≅ defining capabilities

MTL uses tagless final for expressing effects

Type classes can be problematic sometimes

Example

Business rules and validation

Goals

- Validate events in an environment
  - Purchase, registration, product viewing, etc.
- Perform operations after events
  - Sending emails, updating the database, etc.

```
data Registration = Registration
  { firstName :: Text
  , lastName :: Text
  , email :: EmailAddress
  , password :: Password
  , country :: Country
  }
```

```
data User = ...

data BillingInfo = ...

data Purchase = ...

data Product = ...
```

```haskell
data Env = Env
  { envUser :: Maybe User
  ...
  }
```

```haskell
businessRules =
  fold
    [ to buyProduct validatePurchase
    , to register validateRegistration
    , to viewProduct validateViewProduct
    , ...
    , to buyProduct validateUserBilling
    , after buyProduct sendPurchaseEmail
    , ...
    ]
  where
    validatePurchase :: Purchase -> _
    ...
```

```haskell
{-# LANGUAGE RebindableSyntax #-}

businessRules = do
  to buyProduct validatePurchase
  to register validateRegistration
  to viewProduct validateViewProduct
  ...
  to buyProduct validateUserBilling
  after buyProduct sendPurchaseEmail
  ...
  where
    (>>) = (<>)

    validatePurchase :: Purchase -> _
    ...
```

```haskell
validatePurchase purchase = do
  user <- authenticate
  is
    (available
      (orderedQuantity `of_` purchase)
      (orderedProduct `of_` purchase))
  exists (price `of_` orderedProduct `of_` purchase)
  exists (country `of_` user
  done
  where
    available q product = quantity `of_` product >= q

validateUserBilling _ = do
  user <- authenticate
  exists (billingInfo `of_` user)
  done

authenticate = do
  env <- getEnv
  exists (envUser env)
```

What is a possible solution?

Let’s first think about the constraints

1. Validation rules depend on an environment.
2. Validation rules are sequentially composable.
3. Business rules can be combined.
4. The input to a validation rule or to an effect after an event must match the event’s contents.
5. Dispatching an event to a set of business rules must give us back (maybe) an effect.

```
data Event
  = BuyProduct Purchase
  | Register Registration
  | ViewProduct Product
```

```haskell
type Match f a = f -> Maybe a

to :: Match event e -> (e -> Rules) -> Rules

after :: Match event e -> (env -> e -> m a) -> Rules

dispatch :: Applicative m => Rules -> env -> event -> Maybe (m ())
```

```haskell
is :: Bool -> Validation ()

exists :: Maybe a -> Validation a

getEnv :: Validation env

runValidation :: Validation a -> env -> Maybe a


-- Syntax sugar
infixr 9 `of_`
of_ :: (a -> b) -> a -> b
of_ = ($)

done :: Monad m => m ()
done = return ()
```

Exercises

What if we wanted to depend on a database to perform validations?

What if we wanted to add proper error messages?

Try expressing rules with both deep and shallow embedding.

What if we wanted to have a pipeline instead?

That is, feeding validated outputs to `after` rules.
Tip: one possible solution requires having only a single type of rule instead of the original two: `to` and `after`.

Food for thought

Can we refactor `Validation` to be written in terms of classic monad transformers?

How could we statically render validation rules as documentation text?

How can we restrict the type of effects that can be performed `after` each event?

Tip: you can either use records of effects or `ConstraintKinds`.

Questions?

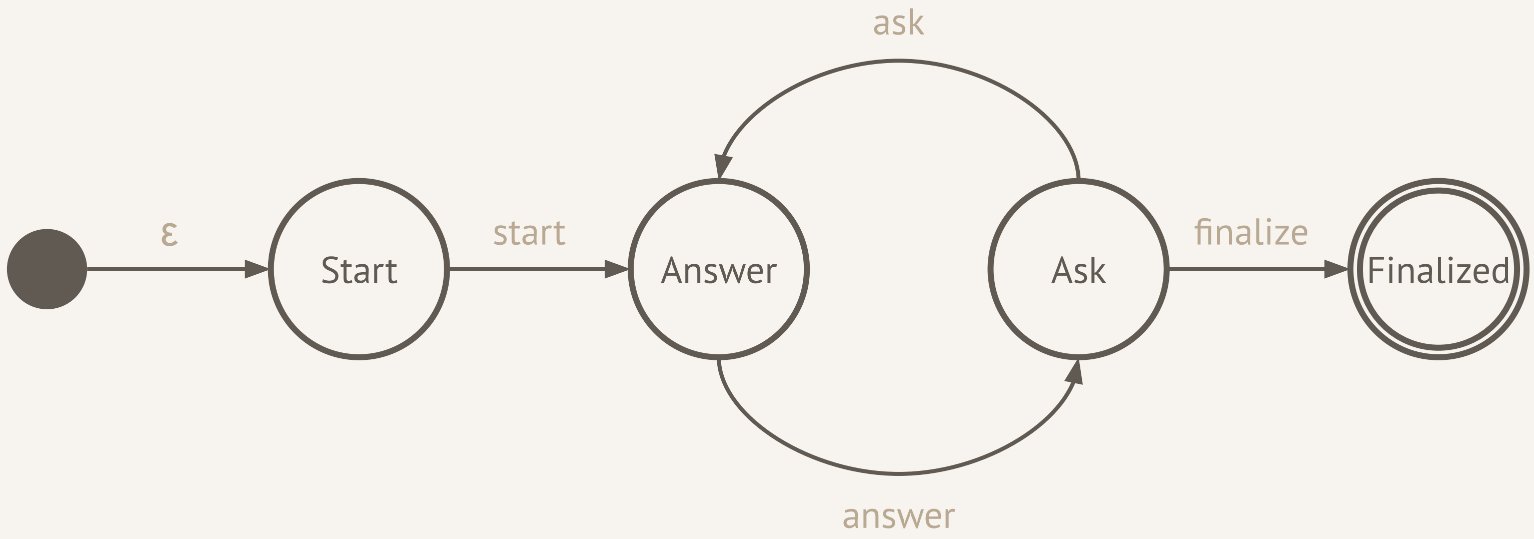
4

Da Capo al Coda

Example

Chatbots with indexed monads

```
$ git clone https://github.com/arthurxavierx/monadic-party-edsl.git
$ cd monadic-party-edsl/chatbot
$ npm ci
$ make run
```


ask
ε
Start
start
Answer
Ask
finalize
Finalized
answer

```haskell
class IxFunctor f where
  imap :: (a -> b) -> f x y a -> f x y b

class IxFunctor m => IxApplicative m where
  iapply :: m x y (a -> b) -> m y z a -> m x z b
         :: m x y a          -> m y z b -> m x z (a, b)
  ipure :: a -> m x x a

class IxApplicative m => IxMonad m where
  ibind :: m x y a -> (a -> m y z b) -> m x z b
```

```haskell
preferredLanguage = Ix.do
  start any
  answer "Hey there!"
  language <- askPreferences
  loop assertPreferences language
  answer "Nice!"
  finalize
```

```purescript
preferredLanguage = Ix.do
  start any
  answer "Which one do you prefer, Haskell or PureScript?"
  language <- ask parseLanguage
  loop language \lang -> Ix.do
    if lang == PureScript then
      break unit
    else Ix.do
      answer "Errrr... Which one do you really prefer?"
      lang' <- ask parseLanguage
      continue lang'
  answer "Nice!"
  finalize
  where
    parseLanguage = do
      prefers <-
        (Haskell <$ match "haskell") <|>
        (PureScript <$ match "purescript")
      pure { prefers }
```

Questions?

Exercises

How can we allow multiple consecutive answers and questions while keeping the initial and final transitions?

How could we make the DSL strictly applicative?
And what implications does this change effectively have?
Tip: we'll need to add two new combinators.

Food for thought

Can you think of a simpler way to express the same DSL?

How would you improve the domain modelling?
How would your changes affect the DSL?

Wrapping up

Wrapping up

Language is the essence of abstraction.

A language can be seen as a pair syntax + semantics.

We can write tiny languages embedded in Haskell that correctly model a domain.

Wrapping up

We can make use of fundamental typeclasses such as `Monoid`, `Applicative` and `Monad` to embed a language more ergonomically and safely.

We can define the syntax of a DSL in Haskell in terms of its abstract syntax or of its semantic domain.
(deep × shallow embedding)

Wrapping up

We can use typeclasses and instances to embed extendable DSLs in Haskell. (tagless final style)

We can leverage Haskell's powerful type system to make our DSLs more expressive and safe.

Creativity is key, but respecting laws is important.

The good

Separation of concerns

High development productivity

Highly maintainable design

Less lines of less complex code
⇒ more maintainable code.

Highly portable design

Depending on the choices regarding embedding techniques.

The good

Opportunities for reuse

User enhanceable systems

Fewer bugs

Improved adaptability

The bad

A hard design problem

Up-front cost

A tendency do use multiple languages

Language cacophony

Maintenance can sometimes be painful

The ugly

Further reading

Monad transformers

Free applicatives & free monads

`Foldable`, `Traversable`, `Alternative`

Type-level programming

Functor combinators: `Sum`, `Product`, `Compose`, `Day`, `Yoneda`, `Coyoneda`, `Alt`

Abstract interpretation

Share your thoughts and conclusions!

References

Ward, M. P. (1994). Language Oriented Programming.

Van Deursen, A. et al. (2000). Domain-specific languages: An annotated bibliography.

Kiselyov, O. (2012). Typed Tagless Final Interpreters.

Gibbons, J., & Wu, N. (2014). Folding Domain-Specific Languages: Deep and Shallow Embeddings.