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

$$1 + 2 + 3 + 4$$

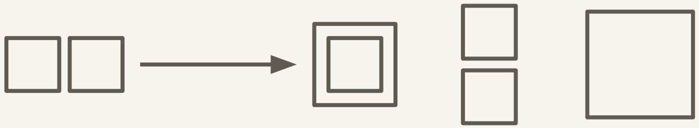
$$1 + 2 + 3 + \ldots$$

1 + 2 + 3 + 4 ⟶

1 + 2 + 3 + 4
10
3 + 3 + 4
1 + (2 + (3 + 4))
□

1 + 2 + 3 + ... ⟶

∞
10
6
((1 + 2) + 3) + ...
□



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

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

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

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

DSL $\stackrel{\text{def}}{=}$ 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

```haskell
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
```

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

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

```
data Primitives = ...

combinator :: _ -> Primitives -> Primitives

interpreter :: Primitives -> _
```

DSL $\stackrel{\text{def}}{=}$ 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

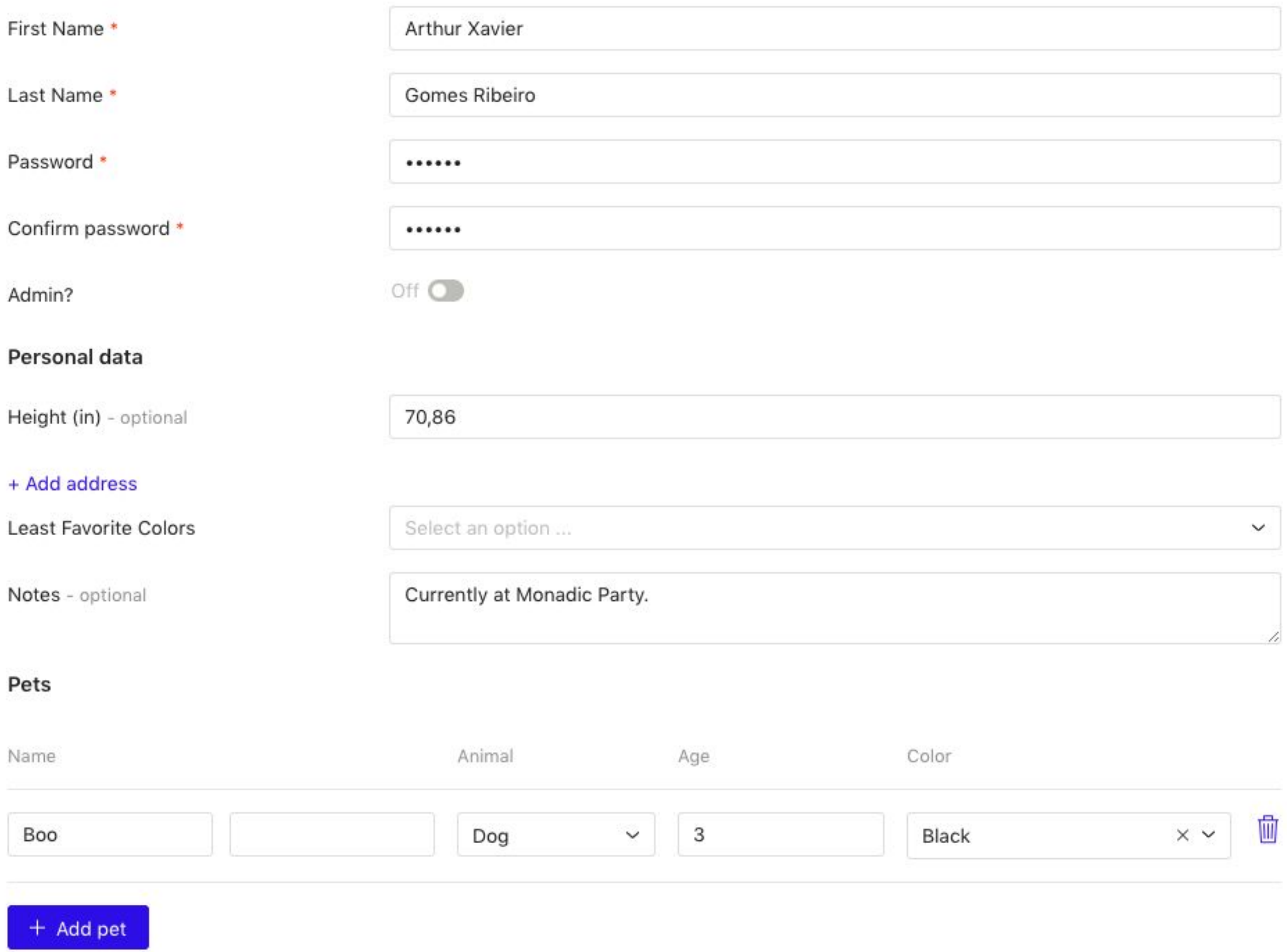
Questions?

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
```

```
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
```

```
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 ≅ 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?

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.