PURESCRIPT

Carsten König

10. April 2018

AGENDA

- kleiner Überblick
- Syntax, Typsystem
- Beispiel / Elm-Architektur

EINLEITUNG

www.purescript.org

- funktionale Programmiersprache
- kompiliert in recht leserliches Javascript
- einfaches JavaScript FFI
- ausdruckstarkes Typensystem

SYNTAX UND FEATURES

AUSDRÜCKE/WERTE

- (fast) alles ist ein **Ausdruck**
- ein *Ausdruck* hat einen **Wert** und einen **Typen**
- Daten/Werte sind *nicht-veränderbar*

BEISPIELE

```
str :: String
str = "Hallo Magdeburg"

zahl :: Number
zahl = 0.99

ganzZahlen :: Array Int
ganzZahlen = [ 1, 2, 3, 4, 5 ]
```

FUNKTIONEN

Funktionen in **PureScript** sind *rein* und *total*

$f : \mathit{Domain} \rightarrow \text{Co-Domain}$

BEISPIEL

```haskell
fizzBuzzNumber :: Int -> String
fizzBuzzNumber = \ n ->
  case Tuple (n `mod` 3 == 0) (n `mod` 5 == 0) of
    Tuple true true  -> "FizzBuzz"
    Tuple true false -> "Fizz"
    Tuple false true -> "Buzz"
    _                -> show n

-- oder
fizzBuzzNumber n =
  case Tuple ...
```

```haskell
fizzBuzzNumbers :: Int -> Int -> Array String
fizzBuzzNumbers from to =
  map fizzBuzzNumber (range from to)

-- oder
fizzBuzzNumbers from to =
  fizzBuzzNumber <$> range from to
```

```
fizzBuzzNumbers :: Int -> Int -> Array String
fizzBuzzNumbers from to =
  map fizzBuzzNumber (range from to)

...

  log (joinWith "\n" (fizzBuzzNumbers 1 30))
```

ALGEBRAISCHE DATENTYPEN

Produkt- / **Summen-**Datentypen

PRODUKT

```haskell
data Tupel a b
  = Tupel a b

tupel = Tupel 42 "Handtuch"

first :: ∀ a b . Tupel a b -> a
first (Tupel a _) = a
```

WARUM *PRODUKT*?

Wieviele mögliche Werte hat der Typ

```haskell
data Kombination = MkKomb Bool Char
```

- `MkKomb false 'a'`
- `MkKomb false 'b'`
- …
- `MkKomb true 'a'`
- …

SUMMEN

```haskell
data Result err res
  = Err err
  | Ok res

okRes :: Result String Int
okRes = Ok 42

withDefault :: ∀ err res . res -> Result err res -> res
withDefault def (Err _) = def
withDefault _   (Ok  v) = v
```

WARUM *SUMME*?

Wieviele mögliche Werte hat der Typ

```haskell
data Alternative = Entweder Bool | Oder Char
```

- `Entweder false`
- `Entweder true`
- `Oder 'a'`
- `Oder 'b'`
- …

ALGEBRAISCH!

Produkt/Summen kann man mischen

```
data Algebraisch a = Entweder a Bool | Oder String
```

RECORDS

Produkt-Typen mit *Labels*

```
type Person =
    { name    :: String
    , caAlter :: Int
    }


carsten :: Person
carsten = { name: "Carsten", caAlter: 30 }


sagHallo :: Person -> String
sagHallo { name: n, caAlter: a } =
    if a <= 30 then "Hallo " <> n else "Guten Tag " <> n
```

gewohnter Syntax geht auch

```
sagHallo :: Person -> String
sagHallo p =
    if p.caAlter <= 30 then
      "Hallo " <> p.name
    else
      "Guten Tag " <> p.name
```

UI MIT PUX

ELM ARCHITECTUR

- **Zustand** wird als *HTML* dargestellt
- Ereignisse (Click,…) erzeugen **Nachrichten**
- aus dem aktuellen *Zustand* und einer *Nachricht* wird ein neuer *Zustand* generiert
- …

ELM ARCHITECTUR

```
type State = { .. }

data Event = ...

view :: State -> HTML Event

update :: Event -> State -> EffModel State Event AppEffects
```

SEITENEFFEKTE

- in **Pux** kann die `update` Funktion Seiteneffekte auslösen
- externe *Signale* können *Events* auslösen

```
main = do
  app <- start
    { initialState: initial
    , view
    , foldp: update
    , inputs: [] -- <- Signale hier bitte
    }
```

DEMO

ZUSTAND

```
type State =
  { scores :: Array Game.Score }
```

EREIGNISSE

```
data Event
  = Reset
  | ThrowDice
  | AddDie Game.Score
```

VIEW

```
view :: State -> HTML Event
view state = do
  h1 $ text "21.."
  div $ do
    viewScores
    viewTotal
    span $ do
      button #! onClick (const ThrowDice) $ text "throw"
      button #! onClick (const Reset) $ text "reset"
  where
    ...
```

UPDATE

noch ein Wurf

```
update :: Event -> State -> EffModel State Event AppEffects
update ThrowDice curState
  | not (Game.isGameOver curState) =
    { state: curState
    , effects:
      [ do
        score <- liftEff (randomInt 1 6)
        pure $ Just $ AddDie score
      ]
    }
  | otherwise =
    { state: curState, effects: [] }
```

UPDATE

Zufallsergebnis eingetroffen

```
update :: Event -> State -> EffModel State Event AppEffects
update (AddDie score) curState
  | not (Game.isGameOver curState) =
    let state' = Game.addDie score curState
    in
      { state: state'
      , effects:
        [ do
          when (Game.isGameOver state') $
            liftEff $ Notify.show "game ended"
          pure Nothing
        ]
      }
  | otherwise = { state: curState, effects: [] }
```

UPDATE

Reset gedrückt

```
update :: Event -> State -> EffModel State Event AppEffects
update Reset _ =
  { state: initialState, effects: [] }
```

NEXT LEVEL

ROW-POLYMORPHISM

Records sind eigentlich

```
data Record :: # Type -> Type
```

(siehe Prim)

Funktioniert mit jedem Record, der mindestens ein Feld `name` vom Typ `String` hat

```
hallo :: forall r . { name :: String | r } -> String
hallo rec = "Hallo " <> rec.name
```

(NATIVE) EFFEKTE

Seiteneffekte sind in *PureScript* explizit über das Typsystem (Monaden)

```
main :: forall e. Eff (console :: CONSOLE | e) Unit
main = log "Hallo Welt"
```

```
data Eff :: # Control.Monad.Eff.Effect -> Type -> Type
```

Effekte können “verzahnt” werden

```
ausgeben :: forall e . Int -> Eff (console :: CONSOLE | e) Unit
ausgeben n = log ("-> " <> show n)

wuerfel :: forall e . Eff (random :: RANDOM | e) Int
wuerfel = randomInt 1 6

wuerfeln :: forall e . Eff ( random :: RANDOM
                           , console :: CONSOLE | e) Unit
wuerfeln = do
  n <- wuerfel
  ausgeben n
```

TYPKLASSEN

```haskell
app :: forall a b . (a -> b) -> a -> b
app f a = f a

plusS :: Int -> Int -> String
plusS a b = show (a + b)
```

Typklassen *schränken* Datentypen ein um in der Klasse definierte Funktionen/Operatoren verfügbar zu machen.

```
plusS :: forall a. Show a => Semiring a => a -> a -> String
plusS a b = show (a + b)
```

TYPKLASSEN MIT MEHREREN PARAMETERN

```haskell
class (Monad m) <= MonadState s m | m -> s where
```

KINDS

```
-- value
a = 5

-- type
a :: Int

-- "type" of type?
Int :: Type
```

HIGHER-KINDED-TYPES

“Funktion” zwischen Typen

```
Int        :: Type

Maybe Int  :: Type

Maybe      :: Type -> Type

List       :: Type -> Type

Fix        :: (Type -> Type) -> Type
data Fix f = Fix (f (Fix f))
```

können Muster wie *Funktoren*, *Monaden*, … in der Sprache ausdrücken!

```haskell
class Functor f where
  map :: forall a b. (a -> b) -> f a -> f b

-- kind:
f :: Type -> Type
```

free monads oder rekursion schemes

```haskell
data Fix f = Fix (f (Fix f))

cata :: forall f a . Functor f => (f a -> a) -> Fix f -> a
cata alg (Fix ff) = alg (map (cata alg) ff)

data ListF el a = Nil | Cons el a
derive instance listFunctor :: Functor (ListF el)

type List el = Fix (ListF el)

mySum :: List Int -> Int
mySum = cata $ case _ of
    Nil          -> 0
    (Cons n acc) -> n + acc
```

HIGHER-RANKED-TYPES

hier gibt es ein `toStr` für ein festes `s`

```
notWorking :: forall s . Show s => (s -> String) -> String
notWorking toStr = toStr 42 <> " and " <> toStr true
                           ^^^ could not match type
```

hier gibt es für jedes s ein eigenes `toStr`

```
useIt :: (forall s . Show s => s -> String) -> String
useIt toStr = toStr 42 <> " and " <> toStr true
```

JS FFI

PureScript Module `Rechnen.purs`

```
foreign import addition :: Number -> Number -> Number
```

mit zugehöriger *JavaScript* Datei `Rechnen.js`

```
exports.addition = function(x) {
  return function (y) {
    return x + y;
  };
};
```

mit Effekten

```
module Notify where

foreign import data NOTIFY :: Effect
foreign import show :: forall eff .
    String -> Eff ( notify :: NOTIFY | eff ) Unit


// Notify.js
exports.show = function (text) {
    // soll ein Effekt werden
    return function () {
        ...
    }
}
```

RESOURCEN

- Homepage - www.purescript.org
- Dokumentation - github.com/purescript/documentation
- PureScript by Example (Buch) - leanpub.com/purescript/read
- Pursuit - pursuit.purescript.org

FRAGEN?

VIELEN DANK!