

Introduction to PureScript for Haskell Developers

Adrian Sieber (ad-si.com)

2019-08-15

Why PureScript Instead of JavaScript

- Safety
 - No more undefined is not a function
 - No more 0 == false
 - No more mutation

```
const people = [{name: 'John'}, {name: 'Anna'}]  
const newPeople = people.map(x => {  
  x.color = 'green'  
  return x  
})
```

- More robust
- Better maintainability and refactoring

Example

Write a function which creates a CSS RGB string:

```
function getRgbColor (red, green, blue) {  
  return `rgb(${red},${green},${blue})`  
}
```

Well ... this shouldn't be possible:

```
console.log(getRgbColor(undefined, 234, null))
```

Let's fix it:

```
function getRgbColor (red, green, blue) {  
  if (red && green && blue) {  
    return `rgb(${red},${green},${blue})`  
  }  
  else throw new Error('Please provide a valid color')  
}
```

But wait a minute:

```
console.log(getRgbColor(221, 175, 15))  
// rgb(221,175,15)  
  
console.log(getRgbColor(221, 0, 89))  
// Error: Please provide a valid color
```

Solution:

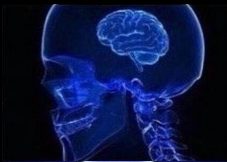
```
function getColor (red, green, blue) {  
  const isValidColor =  
    isFinite(red) &&  
    isFinite(green) &&  
    isFinite(blue)  
  
  // TODO: Check for upper and lower bound  
  
  if (isValidColor) return `rgb(${red},${green},${blue})`  
  else throw new Error('Please provide a valid color')  
}
```

NOT THE GLOBAL isFinite

```
isFinite(null) === true  
Number.isFinite(null) === false
```

JavaScript is a Minefield!

ES3



ES2015



TypeScript



PureScript



Compilation Example 1: Curried Functions

PureScript

```
add3 :: Int -> Int -> Int -> Int
add3 valA valB valC =
    valA + valB + valC
```

JavaScript

```
var add3 = function (valA) {
    return function (valB) {
        return function (valC) {
            return (valA + valB | 0) + valC | 0;
        };
    };
};
```

Compilation Example 2: Foreign Function Interface

JavaScript

```
exports.calculateInterest = amount => {  
  return amount * 0.1  
}
```

PureScript

```
module Interest where  
  
foreign import calculateInterest :: Number -> Number
```

Main Differences

- Prelude
 - Must be included explicitly
 - Smaller
 - No libraries are distributed with the compiler
 - Strict, and not Lazy
 - `head`, `tail`, ... safe per default
 - More explicit:
 - “X has unspecified imports, consider using the explicit form”
 - Type-classes must explicitly be imported with the `class` keyword
- ```
import A (class B)
```
- Applicative Do with `ado` keyword
  - `|` (pipe character) must appear on every line in documentation comments

# Class and Type Differences

- Type variables must be declared

```
length :: forall a. Array a -> Int
```

- Instance chains (type-class programming without overlapping instances)

```
instance zeroSucc :: Succ "zero" "one"
else instance oneSucc :: Succ "one" "two"
...
```

- Class constraint arrow is flipped

```
class (Eq a) <= Ord a where ...
```

where <= means “logical implication”

- No default member implementations for type classes (yet)

# Missing Features

- Lists only via external library (`[]` and `(:)` for Arrays, `List` and `Cons` for Lists)
- No built in tuples (but external `Tuple` library with `Tuple a b`)
- No `qualified` keyword as `import` is qualified per default
- No Template PureScript (yet)
- Orphan instances are completely disallowed

# Deriving

Basically as if `StandaloneDeriving` was enabled in GHC.

Haskell:

```
data Point = Point Int Int deriving (Eq, Ord)
```

PureScript:

```
data Point = Point Int Int

derive instance eqPoint :: Eq Foo
derive instance ordPoint :: Ord Foo
```

# Instance Names

Instances must be given names:

```
instance arbitraryUnit :: Arbitrary Unit where ...`
```

- Increase readability of compiled JavaScript
- Deterministic names are good, but no good function which still produces nice names
- Renaming a class or type can break FFI code
- Name instances differently: `instance refl :: TypeEquals a a`

# Defining Operators

Only available as operator alias for named functions:

Haskell:

```
f $ x = f x
```

PureScript:

```
apply f x = f x
infixr 0 apply as $
```



# Operator Sections

Sections of an infix operator are only available with wildcards:

Haskell:

```
(2 /)
(/ 2)
```

PureScript:

```
(2 / _)
(_ / 2)
```

# Multiline Strings

Additional support for `"""` multiline strings:

Haskell and PureScript:

```
sentence = "\
 \This is\n\
 \just some text\n\
 \split over several lines\n"
```

PureScript:

```
sentence = """This is
just some text
split over several lines
```

# Names

| Haskell      | PureScript       |
|--------------|------------------|
| IO           | Effect           |
| data () = () | data Unit = unit |
| &            | #                |
| Bool         | Boolean          |
| ..           | range            |
| ++           | <>               |
| fmap         | map              |
| Text         | String           |

| Haskell           | PureScript         |
|-------------------|--------------------|
| .                 | <<<                |
| >>>               | >>>                |
| [a]               | Array a / List a   |
| (a, b)            | Tuple a b          |
| return            | pure               |
| [x^2   x<-[1..5]] | list monad + guard |
| undefined         | unsafeCoerce       |
| >>                | *>                 |

But as nothing is included per default, you can change everything!

# Records

```
module Main where
import Effect.Console (log)

book =
 { title: "Eine Woche voller Samstage"
 , author: "Paul Maar"
 , year: 1973
 }

main = log book.title
```

compiles to ...

```
// Generated by purs version 0.12.0
"use strict";
var Effect_Console = require("../Effect.Console/index.js");
var book = {
 title: "Eine Woche voller Samstage",
 author: "Paul Maar",
 year: 1973
};
var main = Effect_Console.log(book.title);
module.exports = {
 book: book,
 main: main
};
```

*Side note:* The syntax for fields is the reason why `.` isn't used for function composition

# Field Access Function

```
main = do
 log $ _.title {title: "Just a title"}
 log $ _.title book
```

## Updating a Record

```
nextBook = book {title = "Am Samstag kam das Sams zurück"}
```

*Side note:* This would be ambiguous if Records used = instead of : for assignments.

Does it mean “apply function book to object { title: "...” }” or “update the title of book”?

## Pattern Matching on Records

```
paulsTitle {author: "Paul Maar", title: t} = Just t
paulsTitle _ = Nothing
```



# Record Types

Haskell:

```
data Book = Book
 { title :: Text
 , author :: Text
 , year :: Int
 }
```

PureScript:

```
type Book =
 { title :: String
 , author :: String
 , year :: Int
 }
```

# Row Polymorphism

```
showPrint :: forall a.
 { title :: String, author :: String | a }
 -> String
showPrint b =
 b.title <> " by " <> b.author

main :: Effect Unit
main = do
 log $ showPrint book
```

The kind of `a` is `# Type` (A row of types)

```
> :kind { title :: String }
Type
> :kind (title :: String)
Type
> :kind Record
Type -> Type
> :kind Record (title :: String)
Type
```

- A row of types is a type level description of pairs of labels and types
- { title :: String } is just syntax sugar for Record ( title :: String )

```
type EitherTitleOrYear = Variant (title :: String, year :: Int)
```

# Type Class Hierarchy

Finer subdivision into more classes:

- `Category` has a superclass `Semigroupoid` (provides `<<<`, does not require an identity)
- `Monoid` has a superclass `Semigroup` (provides `<>`, does not require an identity)
- `Applicative` has a superclass `Apply` (provides `<*>`, does not require an implementation for `pure`)

# Active Extensions

Equivalent to enabling following extension in GHC

- DataKinds
- EmptyDataDecls
- ExplicitForAll
- FlexibleContexts
- FlexibleInstances
- FunctionalDependencies
- KindSignatures
- MultiParamTypeClasses
- PartialTypeSignatures
- RankNTypes
- RebindableSyntax
- ScopedTypeVariables
- UndecidableInstances (cuts off after too much looping)

# Low Level Adaption for JavaScript

Several design decisions were made to improve the generated JavaScript:

- Direct mapping to JavaScript:

```
Boolean = true | false
```

- Arrays instead of lists
- Named instances
- Records  $\rightarrow$  JavaScript Objects
- String is a JavaScript String (and not [Char])

# Tooling

| Haskell             | PureScript                              |
|---------------------|-----------------------------------------|
| ghc                 | purs                                    |
| ghci                | purs repl                               |
| ghcid               | pscid                                   |
| stack               | pulp                                    |
| stack init          | pulp init                               |
| stack exec          | pulp run                                |
| haskell.org         | purescript.org                          |
| hackage.haskell.org | bower.io (or psc-package via git)       |
| hoogle.haskell.org  | pursuit.purescript.org                  |
| try.haskell.org     | try.purescript.org (not up to date yet) |

# Use Cases and Notable Projects 1

- Frontend
  - Halogen - UI library
  - Pux - Web apps like Elm
- Reactive Tools / Websites
  - Flare - Reactive UI
  - PureScript Pop - FRP demo
- JavaScript Plugins
- Cloud functions



## Use Cases and Notable Projects 2

- CLI tools (can also be executed in the browser!)
  - Insect - CLI calculator
  - Transity - Plaintext accounting tool
  - Neodoc - CLI args parser
- Game Development
  - PureScript is Magic
- Interfaces, bindings, and wrappers for JavaScript libraries
  - D3
  - React
- Language experiments
  - Neon - Alternative Prelude without type class hierarchy

# The Not so Good

- Trying to remember the differences to Haskell
- Smaller Ecosystem than Haskell
- New compiler versions break a lot
- Performance
- Memory footprint of Node.js

# Future Development

## PureScript

- More backends:
  - pure-c - C backend for PureScript
  - pureswift - Swift backend for PureScript
- Stable compiler

## Haskell

- Backport features (most PureScript developers are also Haskell developers)
- Maybe it catches up
  - Compile to Webassembly
  - Fix Records

PureScript is currently the best way to  
write JavaScript!