# Haskell and Domain-Specific Languages

## Haskell nejen pro informatiky

Otakar Smrž

Institute of Formal and Applied Linguistics
Faculty of Mathematics and Physics
Charles University in Prague
`otakar.smrz@mff.cuni.cz`

`https://wiki.ufal.ms.mff.cuni.cz/courses:pfl080`

Part I

# Introduction

# Declarative Programming

Declarative operation is one that is independent of any execution state outside of itself, is itself stateless, and is deterministic.

# Declarative Programming

Declarative operation is one that is independent of any execution state outside of itself, is itself stateless, and is deterministic.

Important properties of declarative programming (4):

- declarative programs are compositional
- reasoning about declarative programs is simple

# Declarative Programming

Declarative operation is one that is independent of any execution state outside of itself, is itself stateless, and is deterministic.

Important properties of declarative programming (4):

- declarative programs are compositional
- reasoning about declarative programs is simple

Further classification of declarative languages as in (4):

- descriptive vs. programmable
- observational vs. definitional

# Functional Programming

Functional programming is declarative. Functional operations are void of side-effects and the order of evaluation is irrelevant. Programs are referentially transparent.

# Functional Programming

Functional programming is declarative. Functional operations are void of side-effects and the order of evaluation is irrelevant. Programs are referentially transparent.

Programs and their components are modeled as functions from input arguments to output results.

Functions are first-class values, i.e. can be returned as well as passed as arguments to higher-order functions.

# Functional Programming

Functional programming is declarative. Functional operations are void of side-effects and the order of evaluation is irrelevant. Programs are referentially transparent.

Programs and their components are modeled as functions from input arguments to output results.
Functions are first-class values, i.e. can be returned as well as passed as arguments to higher-order functions.

> *Functional languages contribute greatly to modularity . . . Modularity is the key to successful programming.* (2)

`http://www.md.chalmers.se/~rjmh/Papers/whyfp.html`

# Domain-Specific Languages

*A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.* (3)

DSLs can be embedded in some general-purpose language, such as Haskell . . .

# Haskell

Haskell is a purely functional programming language based on typed $\lambda$-calculus, with lazy evaluation of expressions and many impressive higher-order features.

# Haskell

Haskell is a purely functional programming language based on typed $\lambda$-calculus, with lazy evaluation of expressions and many impressive higher-order features.

> *Haskell computes using definitions rather than the assignments found in traditional languages.* (1)

# Haskell

Haskell is a purely functional programming language based on typed $\lambda$-calculus, with lazy evaluation of expressions and many impressive higher-order features.

> *Haskell computes using definitions rather than the assignments found in traditional languages.* (1)

Haskell is named after the logician H. B. Curry (1900–1982) . . .

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x, y)
```

# Haskell

Haskell is a purely functional programming language based on typed $\lambda$-calculus, with lazy evaluation of expressions and many impressive higher-order features.

> *Haskell computes using definitions rather than the assignments found in traditional languages.* (1)

Haskell is named after the logician H. B. Curry (1900–1982) . . .

```
curry :: ((a, b) -> c) -> (a -> b -> c)
curry f = \ x y -> f (x, y)
```

# Haskell

Haskell is a purely functional programming language based on typed $\lambda$-calculus, with lazy evaluation of expressions and many impressive higher-order features.

> *Haskell computes using definitions rather than the assignments found in traditional languages.* (1)

Haskell is named after the logician H. B. Curry (1900–1982) . . .

```
curry :: ((a, b) -> c) -> (a -> (b -> c))
curry f = \ x -> \ y -> f (x, y)
```

# Haskell

Haskell is a purely functional programming language based on typed $\lambda$-calculus, with lazy evaluation of expressions and many impressive higher-order features.

> *Haskell computes using definitions rather than the assignments found in traditional languages.* (1)

Haskell is named after the logician H. B. Curry (1900–1982) . . .

```
curry :: (((a, b) -> c) -> (a -> (b -> c)))
curry = \ f -> \ x -> \ y -> f (x, y)
```

# Online Resources

Haskell Website `http://www.haskell.org/`

# Online Resources

Haskell Website `http://www.haskell.org/`

Hugs Haskell Interpreter `http://www.haskell.org/hugs/`

Glasgow Haskell Compiler/Interpreter

`http://www.haskell.org/ghc/`

# Online Resources

Haskell Website `http://www.haskell.org/`

Hugs Haskell Interpreter `http://www.haskell.org/hugs/`

Glasgow Haskell Compiler/Interpreter

`http://www.haskell.org/ghc/`

Bibliography on Haskell Research

`http://haskell.readscheme.org/`

# Online Resources

Haskell Website `http://www.haskell.org/`

Hugs Haskell Interpreter `http://www.haskell.org/hugs/`

Glasgow Haskell Compiler/Interpreter

`http://www.haskell.org/ghc/`

Bibliography on Haskell Research

`http://haskell.readscheme.org/`

A Gentle Introduction to Haskell

`http://www.haskell.org/tutorial/`

Yet Another Haskell Tutorial

`http://darcs.haskell.org/yaht/yaht.pdf`

# Other Courses

University of Pennsylvania http://www.cis.upenn.edu/~bcpierce/courses/advprog/

# Other Courses

University of Pennsylvania `http://www.cis.upenn.edu/~bcpierce/courses/advprog/`

Saarland University `http://www.st.cs.uni-sb.de/edu/seminare/2005/advanced-fp/`

Chalmers University `http://www.cs.chalmers.se/Cs/Grundutb/Kurser/afp/`

# Other Courses

University of Pennsylvania `http://www.cis.upenn.edu/~bcpierce/courses/advprog/`

Saarland University `http://www.st.cs.uni-sb.de/edu/seminare/2005/advanced-fp/`

Chalmers University `http://www.cs.chalmers.se/Cs/Grundutb/Kurser/afp/`

Charles University `http://kam.mff.cuni.cz/~rakdver/teaching.html`

# Part II

# Types and Polymorphism

# Types

Types are disjoint sets of uniquely identified values.

Data types describe data structures, the function type -> can be viewed as an encapsulated operation that would map input values to output values.

# Types

Types are disjoint sets of uniquely identified values.

Data types describe data structures, the function type -> can be viewed as an encapsulated operation that would map input values to output values.

The structure of a program must conform to the type system, and conversely, types of expressions can be inferred from the structure of the program. The verification of this important formal property is referred to as type checking.

Values can be defined on the symbolic level, and can be atomic or structured. Numbers, characters, lists of values, sets, finite maps, trees, etc. are all different data types.

Values can be defined on the symbolic level, and can be atomic or structured. Numbers, characters, lists of values, sets, finite maps, trees, etc. are all different data types.

```
data Language = Arabic | Korean | Farsi | Czech | English
data Family = Semitic | IndoEuropean | Altaic
data Answer = Yes | No | Web

isFamily :: Language -> Family -> Answer

isFamily Arabic Semitic = Yes
isFamily Czech  Altaic  = No
isFamily _      _       = Web
```

# Polymorphism

Polymorphism means that types can be parametrized with other types. This implementation of lists is an example thereof:

```haskell
data List a = Item a (List a) | End
```

# Polymorphism

Polymorphism means that types can be parametrized with other types. This implementation of lists is an example thereof:

```
data List a = Item a (List a) | End
```

In other words, lists of some type `a` consist of an `Item` joining the value of type `a` with the rest of `List a`, which repeats until the `End`. Lists like these are homogeneous—all elements of a given list must have the same type `a`.

# Polymorphism

Polymorphism means that types can be parametrized with other types. This implementation of lists is an example thereof:

```
data List a = Item a (List a) | End
```

In other words, lists of some type `a` consist of an `Item` joining the value of type `a` with the rest of **`List`** `a`, which repeats until the `End`. Lists like these are homogeneous—all elements of a given list must have the same type `a`.

In Haskell, lists are predefined and recognize the `:` and `[]` values instead of `Item` and `End`.

## Part III

# Laziness

# Sieve of Eratosthenes

```
primes         :: [Int]
primes         =  sieve [ 2 .. ]

sieve          :: [Int] -> [Int]
sieve (x:xs) = x : sieve [ y | y <- xs, y `rem` x /= 0 ]
```

# Sieve of Eratosthenes

```
primes          :: [Int]
primes          =  sieve [ 2 .. ]

sieve           :: [Int] -> [Int]
sieve (x:xs)    = x : sieve [ y | y <- xs, y `rem` x /= 0 ]


isPrime         :: Int -> Bool
isPrime x       =  x `elemInc` primes

    where   elemInc x (y:ys) | x > y     = elemInc x ys
                             | x == y    = True
                             | otherwise = False
            elemInc _ []                 = False
```

# Fibonacci Numbers

Infinite lists are called streams. Their lazy evaluation is essential not only for these implementations, but is in general a very powerful feature promoting modularity and abstraction.

```
fib = 1 : 1 : [ a + b | (a, b) <- zip fib (tail fib) ]
```

```
fib ⟹ [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ... ]
```

# Fibonacci Numbers

Infinite lists are called streams. Their lazy evaluation is essential not only for these implementations, but is in general a very powerful feature promoting modularity and abstraction.

```
fib = 1 : 1 : [ a + b | (a, b) <- zip fib (tail fib) ]
```

```
fib ⟹ [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ... ]
```

Obviously, not all implementations in Haskell are efficient . . .

```
fibonacci 1 = 1
fibonacci 2 = 1
fibonacci x | x > 2 = fibonacci (x - 1) +
                      fibonacci (x - 2)
            | otherwise = 0
```

# Pascal's Triangle

```
pascalRows, pascalDiag :: [[Integer]]

pascalRows = [1] : [ zipWith (+) ([0] ++ r) (r ++ [0]) |
                     r <- pascalRows ]
```

# Pascal's Triangle

```
pascalRows, pascalDiag :: [[Integer]]

pascalRows = [1] : [ [1] ++ zipWith (+) r (tail r) ++ [1]
                   | r <- pascalRows ]
```

# Pascal's Triangle

```haskell
pascalRows, pascalDiag :: [[Integer]]

pascalRows = [1] : [ [1] ++ zipWith (+) (init r) (tail r)
                     ++ [1] | r <- pascalRows ]
```

# Pascal's Triangle

```
pascalRows, pascalDiag :: [[Integer]]

pascalRows = [1] : [ zipWith (+) ([0] ++ r) (r ++ [0]) |
                     r <- pascalRows ]

pascalDiag = [1, 1 ..] : [ q | d <- pascalDiag, let
                                  q = zipWith (+) d (0 : q) ]
```

# Pascal's Triangle

```
pascalRows, pascalDiag :: [[Integer]]

pascalRows = [1] : [ zipWith (+) ([0] ++ r) (r ++ [0]) |
                     r <- pascalRows ]

pascalDiag = [1, 1 ..] : [ q | d <- pascalDiag, let
                               q = zipWith (+) d (0 : q) ]

pascalRows !! x !! y == pascalDiag !! y !! (x - y)
                     == binomial x y

binomial x y | y < 0 || x < y = 0
             | otherwise = product [y + 1 .. x] `div`
                           product [1 .. x - y]
```

# Pascal's Triangle

```
pascalRows, pascalDiag :: [[Integer]]

pascalRows = [1] : [ zipWith (+) ([0] ++ r) (r ++ [0]) |
                     r <- pascalRows ]

pascalDiag = [1, 1 ..] : [ q | d <- pascalDiag, let
                                  q = zipWith (+) d (0 : q) ]

pascalRows !! x !! y == pascalDiag !! y !! (x - y)
                     == binomial x y


binomial x y | y < 0 || x < y = 0
             | otherwise = product [y + 1 .. x] `div`
                           product [1 .. x - y]

product = foldl' (*) 1    -- strict foldl using $!
```

# References

- Paul Hudak, John Peterson, and Joseph Fasel.
  A Gentle Introduction to Haskell 98.
  `http://www.haskell.org/tutorial/`, 1999.
- John Hughes.
  Why Functional Programming Matters.
  *Computer Journal*, 32(2):98–107, 1989.
- Arie van Deursen, Paul Klint, and Joost Visser.
  Domain-Specific Languages: An Annotated Bibliography.
  *SIGPLAN Notices*, 35(6):26–36, June 2000.
- Peter Van Roy and Seif Haridi.
  *Concepts, Techniques, and Models of Computer Programming*.
  MIT Press, Cambridge, March 2004.