Functional Programming

Introduction

Hannes Saffrich

University of Freiburg
Department of Computer Science
Programming Languages

18. Oktober 2024

Haskell Overview

- Algebraic datatypes
- Pattern matching
- Polymorphism
- Type classes
- Global and principal type inference
- Lazy evaluation
- Pure functional language
- Extensible effect system (monads)
- Many crazy typesystem extensions
  - Higher Kinded Types
  - Rank-N-Polymorphism
  - Linear Types
  - Type Families
  - DataKinds
  - Dependent Types
  - ...

Haskell Overview

- Algebraic Datatypes (Python Motivation)

```python
@dataclass
class MouseClick:
    x: int
    y: int

@dataclass
class KeyPress:
    key: Key

type Event = MouseClick | KeyPress

my_event: Event = MouseClick(23, 42)
```

Haskell Overview

- Pattern Matching (Python Motivation)

```
def event_to_str(e: Event) -> str:
    match e:
        case MouseClick(-42, y):
            return "Burn the witch!"
        case MouseClick(x, y, btn):
            return "Clicked at " + str((x, y))
        case KeyPress(key):
            return str(key)
```

Haskell Overview

- Algebraic Datatypes

```haskell
data Event = MouseClick Int Int | KeyPress Key

myEvent :: Event
myEvent = MouseClick 23 42
```

- Pattern Matching

```haskell
eventToStr :: Event -> String
eventToStr e = case e of
  MouseClick -42 y -> "Burn the witch!"
  MouseClick x   y -> "Clicked at " ++ show (x, y)
  KeyPress key     -> "Pressed key " ++ show key
```

Haskell Overview

- Polymorphism
  - Identity in Python:

    ```
    def id[A](x: A) -> A:
        return x
    ```

  - Identity in Haskell:

    ```
    id :: a -> a
    id x = x
    ```

Haskell Overview

- Polymorphism
  - Identity in Python:

    ```
    def id[A](x: A) -> A:
        return x
    ```

  - Identity in Haskell:

    ```
    id :: forall a. (a -> a)
    id x = x
    ```

Haskell Overview

- Type Classes

```
class Show a where
  show :: a -> String

instance Show Event where
  show e = eventToStr e

instance Show Int where ...    -- Defined in stdlib

showBoth :: (Show a, Show b) => a -> b -> String
showBoth x y = show x ++ ", " ++ show y

showBoth Left 42    -- evaluates to "Left, 42"
```

- Recall `42 :: Num a => a` and `42.0 :: Fractional a => a`

Haskell Overview

- Global and principal type inference
  - We can write

    ```
    showBoth x y = show x ++ ", " ++ show y
    ```

  - and Haskell figures out that

    ```
    showBoth :: (Show a, Show b) => a -> b -> String
    ```

  - However, by convention type annotations are written for top-level functions, which helps as documentation and improves error messages

Haskell Overview

- Lazy Evaluation
  - Consider the following function

```
f :: Bool -> Int -> Int
f b x = if b then x else 0
```

    which is called as `f False (5 + 2)`
  - Normally this happens:

```
  f False (5 + 2)
→ f False 7
→ if False then 7 else 0
→ 0
```

  - With lazy evaluation this happens:

```
  f False (5 + 2)
→ if False then (5 + 2) else 0
→ 0
```

Haskell Overview

- Pure Functional Language
  - Functions have no sideeffects
  - Same input implies same output
  - This allows for nice equational reasoning, e.g.

    ```
    f x + f x    ===    let y = f x in y + y
    ```

  - A rare property among production-grade languages:
    Haskell, Nix, Agda, Coq, Lean

wait... wut?

- If there are no side effects ...
- ... how is it possible to write any kind of reasonable program?

Haskell Overview

- Extensible Effect System (Monads)
  - Idea
    - functions don't actually perform side effects, but instead return a description of the side effects as data
    - this data is propagated to the `main` funtion which again returns it
    - the runtime system reads the descriptions and actually executes them
  - Example: Reading a file into a string

```
readFile :: FilePath -> IO String
```

  - Calling `readFile "foo.txt"` does not read the file, but instead returns an `IO` action, which only when executed reads the file and returns a string
  - Multiple actions can be combined and are finally returned from `main`, which causes them to be exectued by the runtime system.
  - The main expression has type

```
main :: IO ()
```

Haskell Overview

- Extensible Effect System (Monads)
  - What does this mean for purity? In particular, for the equation
    ```
    f x + f x    ===    let y = f x in y + y
    ```
  - Let's consider the following function to print to the terminal:
    ```
    putStrLn :: String -> IO ()
    ```
  - A "hello world" looks as follows:
    ```
    main :: IO ()
    main = putStrLn "Hello!"
    ```
  - What if we want to call `putStrLn` multiple times?
  - The >> operator allows to combine `IO` actions
    ```
    (>>) :: IO a -> IO b -> IO b
    ```
  - This allows us to write the following two programs:
    ```
    main = putStrLn "Hello!" >> putStrLn "Hello!"
    main = let x = putStrLn "Hello!" in
           x >> x
    ```

Why Learn Haskell?

- Some language constructs are just generally great for programming
  - Algebraic Datatypes, Pattern Matching, Type Classes, Higher-Order Functions
  - All available in Rust
- Some design concepts can also be enlightening in other languages
  - Learning the trade-offs of programming in a pure functional style
  - Learning the design patterns to make this work properly
  - It's a different way of thinking about programming
- If you want to, you can actually program in Haskell and find a job
  - Small amount of companies, but also small amount of Haskell programmers
- Many people, who went through the journey of learning Haskell, feel like it made them a better programmer in general
- Gateway drug to learning a theorem prover / dependently-typed language like Agda, Coq or Lean, which have to be pure to not allow for proving wrong theorems

Built-in Types

- Booleans (`Bool`)
- Integers (`Int` and `Integer`)
- Floats (`Float` and `Double`)
- Tuples (e.g. `(Int, Bool, Float)`)
- Lists (e.g. `[Int]`)
- Characters and Strings (`Char` and `String`)
- Functions (e.g. `Int -> Int`)

Built-in Types

Booleans

- Values

```
True  :: Bool
False :: Bool
```

- Logical connectives

```
not True       -- Negation
True && False  -- Conjunction
True || False  -- Disjunction
```

- If-Then-Else expression

```
if True then 0 else 1
```

- Short circuiting (like everything because of lazy evaluation)
- Defined by the algebraic datatype

```
data Bool = True | False
```

Built-in Types

Integers

- Values

```
5 :: Int
5 :: Integer
...
```

- `Int` is a fixed size 30bit integer with overflow
- `Integer` is a arbitrary sized bigint implementation
- Typical Operations

```
1 + 1    1 - 1    1 * 1
5 / 2       -- returns 2.5
5 `div` 2   -- returns 2
5 `mod` 2   -- returns 1
2 ^ 4       -- returns 16
1 > 1    1 >= 1    1 == 1    1 != 1
```

Built-in Types

Floats

- Values

```
5.0 :: Float
5.0 :: Double
...
```

- `Float` is a 32bit floating point number
- `Double` is a 64bit floating point number
- Basically same operations as on integers

```
(*) :: Num a => a -> a -> a
(^) :: (Num a, Integral b) => a -> b -> a
(/) :: Fractional a => a -> a -> a
```

Built-in Types

Tuples

- Values

```
()              :: ()
(1, 2)          :: (Int, Float)
(1, 2, True)    :: (Int, Int, Bool)
...
```

- `Float` is a 32bit floating point number
- `Double` is a 64bit floating point number
- Basically same operations as on integers

```
(*) :: Num a => a -> a -> a
(^) :: (Num a, Integral b) => a -> b -> a
(/) :: Fractional a => a -> a -> a
```

Built-in Types

Lists

- Values

```
[]               :: [Int]
[]               :: [Bool]
2 : []           :: [Int]
3 : (2 : [])     :: [Int]
3:2:[]           :: [Int]
[3, 2]           :: [Int]
```

- Careful: Linked lists, not arrays!
- Standard library functions

```
[1, 2] ++ [3, 4]  -- returns [1, 2, 3, 4] (list concat)
[1, 2] !! 0       -- returns 1 (list indexing)
length [1, 2]     -- returns 2 (list length)
...               -- many more in module Data.List
```

Built-in Types

Lists

- Values

```
[]              :: [Int]
[]              :: [Bool]
2 : []          :: [Int]
3 : (2 : [])    :: [Int]
3:2:[]          :: [Int]
[3, 2]          :: [Int]
```

- Behave as if they were defined by the Algebraic Datatype

```
data List a = Nil | Cons a (List a)
```

- Just instead of
  - `List a` we write `[a]`
  - `Nil` we write `[]`
  - `Cons 3 Nil` we write `3 : []`, etc.

Built-in Types

Characters and Strings

- Values

```
'a'            :: Char
"ab"           :: String
```

- Characters are Unicode
- `String` is a type alias for `[Char]`
- `"ab"` is literally the same as `['a', 'b']`
- Sounds insane, but sometimes reasonable, because of lazy evaluation
- the `text` package on Hackage provides the usual UTF-8 String datatype

Built-in Types

Functions

- Values

```
\x -> x + 1  :: Int -> Int
```

- The backslash \ looks sort of like a lambda λ
- Top-level function = global constant variable with function value, e.g.

```
increment :: Int -> Int
increment = \x -> x + 1
```

- Syntactic sugar to make it look nicer:
- Top-level function = global constant variable with function value, e.g.

```
increment :: Int -> Int
increment x = x + 1
```

Built-in Types

Functions

- Haskell supports only single-parameter functions
- Functions taking multiple arguments can be encoded in two ways:
  1. As *curried* functions, e.g.

```
add :: Int -> (Int -> Int)
add = \x -> (\y -> (x + y))

test :: Int
test = (add 2) 3
```

Built-in Types

Functions

- Haskell supports only single-parameter functions
- Functions taking multiple arguments can be encoded in two ways:
  1. As *curried* functions, e.g.

```
add :: Int -> Int -> Int
add = \x -> \y -> x + y

test :: Int
test = add 2 3
```

Built-in Types

Functions

- Haskell supports only single-parameter functions
- Functions taking multiple arguments can be encoded in two ways:
  1. As *curried* functions, e.g.

```
add :: Int -> Int -> Int
add = \x y -> x + y

test :: Int
test = add 2 3
```

Built-in Types

Functions

- Haskell supports only single-parameter functions
- Functions taking multiple arguments can be encoded in two ways:
  1. As *curried* functions, e.g.

```
add :: Int -> Int -> Int
add x y = x + y

test :: Int
test = add 2 3
```

Built-in Types

Functions

- Haskell supports only single-parameter functions
- Functions taking multiple arguments can be encoded in two ways:
  1. As *curried* functions, e.g.

     ```
     add :: Int -> Int -> Int
     add x y = x + y

     test :: Int
     test = add 2 3
     ```

  2. As *uncurried* functions, e.g.

     ```
     add :: (Int, Int) -> Int
     add = \xy -> fst xy + snd xy

     test :: Int
     test = add (2, 3)
     ```

Built-in Types

Functions

- Haskell supports only single-parameter functions
- Functions taking multiple arguments can be encoded in two ways:
  1. As *curried* functions, e.g.

     ```
     add :: Int -> Int -> Int
     add x y = x + y

     test :: Int
     test = add 2 3
     ```

  2. As *uncurried* functions, e.g.

     ```
     add :: (Int, Int) -> Int
     add = \(x, y) -> x + y

     test :: Int
     test = add (2, 3)
     ```

Built-in Types

Functions

- Haskell supports only single-parameter functions
- Functions taking multiple arguments can be encoded in two ways:
  1. As *curried* functions, e.g.

     ```
     add :: Int -> Int -> Int
     add x y = x + y

     test :: Int
     test = add 2 3
     ```

  2. As *uncurried* functions, e.g.

     ```
     add :: (Int, Int) -> Int
     add (x, y) = x + y

     test :: Int
     test = add (2, 3)
     ```

Built-in Types

Functions

- Haskell supports only single-parameter functions
- Functions taking multiple arguments can be encoded in two ways:
  1. As *curried* functions, e.g.

     ```
     add :: Int -> Int -> Int
     add x y = x + y

     test :: Int
     test = add 2 3
     ```

  2. As *uncurried* functions, e.g.

     ```
     add :: (Int, Int) -> Int
     add(x, y) = x + y

     test :: Int
     test = add(2, 3)
     ```

Built-in Types

Functions

- In Haskell people usually use curried functions
- Those are nicer for partial applications:

```haskell
add :: Int -> Int -> Int
add x y = x + y

inc :: Int -> Int
inc = add 1
```

- Compared to the uncurried version:

```haskell
add :: (Int, Int) -> Int
add (x, y) = x + y

inc :: Int -> Int
inc x = add (1, x)
```

Built-in Types

Functions

- Custom operators can be defined by using non-alphanumeric symbols, e.g.

```
(*+) :: Int -> Int -> Int
x *+ y = x * y + y

infixr 9 *+  -- make *+ right-associative and
             -- bind with strength 9

test :: Int
test = 5 *+ 3 *+ 4  -- same as 5 *+ (3 *+ 4)
```

Built-in Types

Functions

- Regular functions can also be used infix by enclosing them with backticks, e.g.

```
max :: Int -> Int -> Int
max x y = if x > y then x else y

test :: Int
test = max 5 2
test = 5 `max` 2
```

- This is how the `div` and `mod` functions are commonly used
- Operators can be partially applied by using *operator sections*

```
inc :: Int -> Int
inc = (1+)    -- (1+) is equivalent to \x -> 1 + x
```

Basic Language Constructs

Let Expression

- `let x = e1 in e2`
- Binds variable `x` to have the value of expression `e1` in expression `e2`
- Allows to bind multiple variables at once and supports optional type signatures

```
test :: Int
test =
  let
    x :: Int
    x = 3
    y :: Int
    y = 3
  in x + y
```

Basic Language Constructs

Pattern Matching

- The case expression can be used for pattern matching, e.g.

```
oneZero :: (Int, Int) -> Bool
oneZero p = case p of
  (0, y) -> True
  (x, 0) -> True
  _      -> False
```

- Patterns can also be used **everywhere** where variables are defined
  - In functions:

```
oneZero :: (Int, Int) -> Bool
oneZero (0, y) = True
oneZero (x, 0) = True
oneZero _      = False
```

  - In `let`-expressions:

```
oneZero :: (Int, Int) -> Bool
oneZero p = let (x, y) = p in
                x == 0 || y == 0
```

Basic Language Constructs

Pattern Matching

- Pattern guards allow integrating boolean expressions into pattern clauses
- Absolute value without pattern guards:

```
abs :: Int -> Int
abs x = if x > 0 then x else -x
```

- Absolute value wit pattern guards:

```
abs :: Int -> Int
abs x | x >= 0 = x
      | x <  0 = -x
```

Basic Language Constructs

Pattern Matching

- Pattern guards allow integrating boolean expressions into pattern clauses
- Absolute value without pattern guards:

```
abs :: Int -> Int
abs x = if x > 0 then x else -x
```

- Absolute value wit pattern guards:

```
abs :: Int -> Int
abs x | x >= 0 = x
abs x | x <  0 = -x
```

Basic Language Constructs

Pattern Matching

- Pattern guards allow integrating boolean expressions into pattern clauses
- Absolute value without pattern guards:

```
abs :: Int -> Int
abs x = if x > 0 then x else -x
```

- Absolute value wit pattern guards:

```
abs :: Int -> Int
abs x | x >= 0 = x
abs x          = -x
```

Basic Language Constructs
Pattern Matching

- Pattern guards allow integrating boolean expressions into pattern clauses
- Absolute value without pattern guards:

```
abs :: Int -> Int
abs x = if x > 0 then x else -x
```

- Absolute value wit pattern guards:

```
abs :: Int -> Int
abs x | x >= 0    = x
abs x | otherwise = -x
```