Haskell

David Cain

2012-12-05

About Haskell

History

- open source
- purely functional
- lazy
- Haskell Curry (1900 - 1982)

Functions

- Called with arguments space-delimited
  - `max 3 4` → 4
  - `mod 10 3` → 1
  - `div 10 3` → 3
- Infix notation
  - `3 ‘max‘ 4`
  - `10 ‘mod‘ 3`
  - `10 ‘div‘ 3`

Introduction

Fibonacci numbers

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Introduction

Lists

- Core data type
- Pythonic syntax
  - `[8,6,7,5,3,0,9]`
- Homogenous linked list
  - Valid: `['H','e','l','l','o']`
  - Illegal: `['W', 0, 'r','l','d']`

Chained lists

- Strings are lists
  - `"Dog" == ['D','o','g']`
- Head + rest of list (like Lisp)
  - `'H' : ['e','l','l','o']`

Lists

Ranges

- `[1,2..6]` → `[1,2,3,4,5,6]`
- `[4..8]` → `[4,5,6,7,8]`
- `[1,3..10]` → `[1,3,5,7,9]`
- `[10,20..]` → `[10,20,30,..]`

List comprehension

```
ten_squares = [x^2 | x <- [1..10]]
odd_squares = [y | y <- ten_squares, odd y]
```

Types

Typing system

- Types are static, inferred at compile time
- Ducks and donkeys
- Type classes
  - **not** classes in OO sense
  - kind of like Java interfaces ("only better")
  - divide types based of kinds of behavior they support
  - `[Eq]`, `[Ord]`, `[Num]`, `[Show]`

Polymorphism

- many kinds, deriving from type system
- Example: sorting algorithm
  - comparative: inferred for comparable types (those in `[Ord]`)
  - bin sort (e.g. radix sort): inferred to work on numeric types

Types

Type signatures

- Terse
- Give rules for parameters, state function behavior

```
sum :: Num x => [x] -> x
sum [] = 0
sum (x: xs) = x + sum xs
```

Custom types

- Standard built-in types (`Double`, `Integer`, `Int`, etc.)
- Aggregate types

Examples

ghci

- Glasgow Haskell Compiler
- Interactive shell (like `ipython` or `ruby`)

Custom data types

- Define a person
- Store information about a person

Example program

Look for:

- Type signature
- I/O
- Procedural code? . . .

Side effects?

- I/O *action* is bound to `name`, not the result
- I/O contained within a monad

Monads

Thus in Haskell, though it is a purely-functional language, side effects that **will be performed** *by a computation can be dealt with and combined purely at the monad's composition time... While programs may describe impure effects and actions outside Haskell, they can still be combined and processed ("assembled") purely, inside Haskell, creating a pure Haskell value - a computation action description that describes an impure calculation. That is how Monads in Haskell* **separate** *between the pure and the impure.*

Haskell Wiki - Monad

Lazy evaluation

To infinity...

```
squares = [x^2 | x <- [1..]]
odd_squares = [y | y <- squares, odd y]
```

- Lists are infinite
- How is this valid?

Lazy evaluation

Infinite lists are valid

- Key is lazy evaluation
- Computations only performed as needed

```
import Data.Array
arr = listArray (0, 1000) odd_squares
```

- Will just run infinitely if not bounded

Curried functions

Currying

- Haskell Curry
  - namesake
  - "re-discovered" (originally Schönfinkel)
- What happens if we call a function with too few arguments?
- Partial function application

Example

```
ghci> :t div
div :: Integral a => a -> a -> a
ghci> let div_fn = div 42
ghci> div_fn 21
2
ghci> div_fn 27
1
```

Real World Haskell

Criticism

- Language criticized for being overly "academic"
- High barriers to entry

Counter points

- fast programs
- rock-solid code
- gaining popularity

"Real World" programs

- pandoc
- xmonad
- darcs

Real World Haskell

Why learn Haskell?

- different paradigm
- fast code
- easy to debug
- new way to think about code
- lots of resources to help learn