Haskell

A Wild Ride

Sven M. Hallberg

pesco@gmx.de

Structure

- Safety Precautions
  introduction + language fundamentals
- Basic Attractions
  some cool stuff
- Wipeout!
  the real fun stuff

15-min. Break

- Workout
  workshop part

Safety Precautions

Safety Precautions

- This is *not* a programming course.
- I’m only skimming details, to get to the interesting points.
- Feel free to ask!
- This is supposed to be a *fun ride*!

Language Classification

Haskell is:

- purely
- functional
- statically typed
- strongly typed
- lazy

Compiled or Interpreted?

Can be both, compiled and interpreted.

- GHC: compiler, machine code
- Hugs: interpreter
- NHC: compiler, bytecode
- HBI/HBC: interpreter/compiler
- Helium: interpreter, for a subset

Program Structure

- A program is a sequence of declarations.
- Declarations are absolute.
- “Variables” cannot be changed at run-time!

```
hello     = "Guten Tag!"
goodbye   = "Auf Wiedersehen!"
```

Function Definitions

```
foo x   = x + 2
bar x y = foo y + x
```

- Function arguments seperated only by whitespace.
- Operators are written in infix notation.
- No type signatures?
- Types are deduced!

Procedures

- All functions are “pure”.
- What about network or file I/O?
- Solution: Strict seperation by the *type system*!

```
main = do putStrLn hello
          input <- getLine
          putStrLn (show input ++ "?")
          putStrLn goodbye
```

Types

- The type system is one of the most important parts of Haskell.
- Every expression is statically typed.
- Type signatures are *optional*.
  - documentation for the programmer
  - better error reporting by the compiler

```
hello :: String
```

Lists

- Probably most important data structure.
- Singly-linked, similar to LISP.

```
intlist   :: [Int]
intlist   =  [1,2,3]

charlist  :: [Char]
charlist  =  ['H','i']

emptylist :: [a]
emptylist =  []
```

Function Types

- Functions are first-class values.
- One argument:

```
foo :: Int -> Int
```

- Two arguments:

```
bar :: Int -> Int -> Int
```

- Last type is the return type.

List Constructors

- Values are created by constructors.
- Constructors are functions.
- `[]` and `(:)` are the constructors for `[a]`.

```
fourints = 1 : (2 : (3 : (4 : [])))
```

- The syntax for list literals is only syntactic sugar for repeated application of `(:)`!

```
[1,2,3] == 1:2:3:[]
```

Regular Attractions

Local Definitions

- Introduced by `where` clause after a declaration.

```
readability :: String -> Float
readability text =
        if n==0 then 1
                else 1 / fromIntegral n
    where
    n = length text
```

- There is also `let` for use in expressions.

```
foo 5 == (let x=5 in x)
```

Constructor Pattern Matching

- Values are created by applying constructors to other values.
- constants = nullary constructors, e.g. `[]`, `1`, `2`,...
- The constructor and its arguments *are* the value.
- Inspection by pattern matching on the constructors.

```
null []     = True
null (x:xs) = False   -- ctor arguments bound
```

User-Defined Data Types

Example: Controlling a magnetic card reader

```
data Cmd =  Read   Track
         |  Write  Track

data Track = Track1 | Track2 | Track3
```

- Types and constructors are written in upper-case.
- Remember: Constructors can take any number of arguments, of any (given!) type.

User-Defined Data Types

Suppose the following control protocol for the reader:

- Commands are three bytes, to be sent over serial.
- First byte: `'a'` = "read", `'b'` = "write"
- Second byte: always `'a'`
- Third byte: `'a'`, `'b'`, `'c'` for track 1,2,3 resp.

E.g.: `"aaa"` for "read track 1".

User-Defined Data Types

⇒ Trivial Haskell function mapping `Cmd`s to control strings:

```
ctlstr :: Cmd -> String
ctlstr (Read   Track1) = "aaa"
ctlstr (Read   Track2) = "aab"
ctlstr (Read   Track3) = "aac"
ctlstr (Write  Track1) = "baa"
ctlstr (Write  Track2) = "bab"
ctlstr (Write  Track2) = "bac"
```

User-Defined Data Types

- Given I/O routine `sendstr` to transmit a string to the device:

```
sendcmd cmd = sendstr (ctlstr cmd)
```

⇒ Interactive device controll from an interpreter.

```
Main> sendcmd (Read Track1)
...stuff happens...
```

- Remember: `Cmd`s can be passed around and stored in data structures.

List Comprehensions

- Lists and list operations are very common.
- List comprehensions are syntactic sugar for combining
  - collection/selection of input elements and
  - generation of corresponding output elements.
- Similar to set comprehensions in Mathematics:

$$\{x^2 | x \in \mathbb{N}, x > 5\}$$

List Comprehensions

Example: Haskell implementation of Quicksort:

```
qs []       = []
qs (x:xs)   = qs [y | y<-xs, y<x]
              ++ [x] ++
              qs [y | y<-xs, y>=x]
```

Wipeout!

The Hackers Must Have Slack.

- Lazy evaluation enables construction of infinite data structures.
- Infinite lists especially
- Through recursive definitions

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Who Uses RC4 Anyway?

Example: Memoizing an infinite data structure.

“Arcfour” works roughly like this:

- From the key, calculate an S-Box.
- Iterate:
  - Extract a certain element from the S-Box, put in keystream.
  - Transform the S-Box in a certain way.
- XOR the resulting keystream with the plain-text.

Who Uses RC4 Anyway?

Imagine implementing this in C.

- data structure for the S-Box
- initialization routine to calculate it from key
- generation of the stream in little chunks

In Haskell:

- no chunking
- no initialization routine!

Who Uses RC4 Anyway?

Pseudocode:

```
type Key  = String
data SBox = ...

mksbox    :: Key -> SBox
keystream :: SBox -> [Word8]

rc4 :: Key -> [Word8] -> [Word8]
rc4 k xs = zipWith xor xs
             (keystream (mksbox k))
```

- What’s the point?

Partial Application

Suppose you want to encrypt a bunch of files with the same key.

Overly verbose:

```
key = "deadbeef"

file1_encrypted = rc4 key file1
file2_encrypted = rc4 key file2
```

Partial Application

Suppose you want to encrypt a bunch of files with the same key.

Sleek:

```
enc = rc4 "deadbeef"

file1_encrypted = enc file1
file2_encrypted = enc file2
```

Partial Application

Suppose you want to encrypt a bunch of files with the same key.

Sleek:

```
enc = rc4 "deadbeef"

file1_encrypted = enc file1
file2_encrypted = enc file2
```

- One problem: keystream will be calculated for each call to `enc`.

Memoization

- The keystream depends only on the key.
- The Haskell system is not smart enough to see that.
- Make it explicit by moving to outer closure:

```
rc4 k = \xs -> zipWith xor xs ks
    where
    ks = keystream (mksbox k)
```

- Then, all calls to `rc4 key` refer to the same `ks`.

Type Classes

- Haskell supports compile-time polymorphism.

```
null :: [a] -> Bool
```

- Sometimes, that's too general.
- `(+)` is polymorphic, but in a restricted way:

```
(+) :: (Num a) => a -> a -> a
```

- Read: "`a -> a -> a` under the constraint that `a` is a number".

Type Classes

The definition of a type class looks like this:

```
class Num a where
    (+)      :: a -> a -> a
    (*)      :: a -> a -> a
    negate   :: a -> a
    ...
```

- Type classes prescribe a kind of interface.
- Still *compile-time* polymorphic!

Typing Harder

- `Num` is essentially the algebraic class of rings.
- `Fractional` contains essentially the fields.

```
class (Num a) => Fractional a where
    (/)      :: a -> a -> a
    recip    :: a -> a
```

- Goal: Declare the class of *vector spaces*.
- Problem: vector space $\leftrightarrow$ associated field

```
    smul     :: a -> v -> v
```

“Multi-parameter type classes”

- Solution: Extend type classes (i.e. sets of types) to relations between types.

```
class (Fractional a) => VS v a
    where
    -- vector add and subtract
    (^+^)   :: v -> v -> v
    (^-^)   :: v -> v -> v
    -- scalar multiplication
    (*^)    :: a -> v -> v
```

“Multi-parameter type classes”

- Solution: Extend type classes (i.e. sets of types) to relations between types.
- Also needed: Functional dependencies on type relations.

```
class (Fractional a) => VS v a |v->a
    where
    -- vector add and subtract
    (^+^)   :: v -> v -> v
    (^-^)   :: v -> v -> v
    -- scalar multiplication
    (*^)    :: a -> v -> v
```

Vector Space Example

To declare the `Float`-pairs to form a vector space (over scalar type `Float`):

```
instance VS (Float,Float) Float where
      (x,y) ^+^ (a,b) = (x+a, y+b)
      (x,y) ^-^ (a,b) = (x-a, y-b)
      k       *^  (a,b) = (k*a, k*b)
```

Notes:

- Multi-parameter type classes and “fundeps” are not Haskell 98.
- Both are supported by all major implementations.

Conclusion

- Haskell is a vast topic.
- Extensions are under active research.
- Still, the language is quite clear.
- Can express many things very naturally.
  - Programs are very concise.
  - Rapid prototyping
- Safe and robust code

All Further Info

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

Workout

Materials

- GHC

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

  `http://www.haskell.org/hugs/`
- Emacs mode

  `http://www.haskell.org/haskell-mode/`
- Vim syntax highlighting

  `http://urchin.earth.li/~ian/vim/`

Ex. 1:

Implement “Hello, World!”.

a) Compile the program and run it as a stand-alone executable.

b) Run the main procedure from an interpreter prompt.

c) Try calling some basic I/O routines interactively at the prompt.

- Scream when done.

Ex. 2:

Implement a function that sums a list of numbers.

- Use pattern matching and recursion.

a) What is the type of this function?

b) Try your implementation on some example inputs.

Ex. 3:

Implement a function that increments all elements in a list of numbers.

- Use pattern matching, recursion, and list construction.
- The function should have the type:

```
(Num a) => [a] -> [a]
```

Ex. 4:

Generalize the function from ex. 3 to apply any given function of type `Int -> Int` to all elements of a list of `Int`s.

a) What should the type of this function be?

b) Can the function be generalized to other types than `Int`?

Ex. 5:

- Import the bit-manipulation modules.

```
import Data.Bits
import Data.Word
```

- Look up the `rotate` method of class `Bits` in the GHC documentation.

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

- Implement a function to rotate every byte in a given list by 4 bits.

- Use your solution to ex. 4b or the standard function `map`.

□