Functional Programming
Lecture 11: Haskell I/O

Viliam Lisý

Artificial Intelligence Center
Department of Computer Science
FEE, Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

Haskell is Purely Functional

- Functions have no side effects
  - outputs depend only on inputs
  - calling function with same arguments multiple times produces the same output
  - order of executing independent functions is arbitrary
  - Haskell functions cannot change files or print
- Pseudo-functions like rand() or getchar() in C
  - return different value each call
  - change files, network, content of the screen

Haskell is Purely Functional

- Optimizations are pure function transformations
  - rearrange calls, cache results
  - omits calling functions, unless their results are used (lazy)
  - might automatically parallelize (but granularity ☹)
  - easier to proof correctness of optimizations
- Optimization in C must be more conservative

- We want to keep purely functional nature
- But we want to be able to interact, change files, etc.

IO Actions

- Haskell separates the part of the program with side effects using values of special types
- (IO a) is an action, which when executed produces a value of type a

```
getChar :: IO Char
getLine :: IO String
putStrLn :: String -> IO ()
```

- IO actions are can be passed from function to function, but are not executed in standard evaluation

Main

Haskell program executes an action returned by function main in module Main

```
main :: IO ()
main = putStrLn "Hello, World!"
```

Running the program

ghc <filename.hs>; ./<filename>

runhugs <filename.hs>

Sequencing actions

In order to call multiple functions, they need to provide arguments for some other function

$$g(f_1, f_2, \dots, f_n)$$

In pure functional programming

- f_i can be called in arbitrary order
- are called only when we need the return value
  - When do we need the return value of `putStrLn`?

Combining actions

```
(>>) :: IO a -> IO b -> IO b
infixl 1 >>
```

(x >> y) is the action that performs x, dropping the result, then performs y and returns its result.

```
main = putStrLn "Hello" >> putStrLn "World"
```

Combining actions: bind

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

x >>= f is the action that first performs x, passed its result to f, which then computes a second action to be performed. That action is then executed.

```
main = putStrLn "Hello, what is your name?"
      >> getLine
      >>= \name -> putStrLn ("Hello, " ++ name ++ "!")
```

```
x >> y = x >>= \_ -> y
```

Combining actions: return

```
return :: a -> IO a
```

Transforms a value to IO action

Used to define the return value of a composed action

```
main :: IO ()
main = return "Viliam" >>= \name
       -> putStrLn ("Hello, " ++ name ++ "!")
```

Did we solve the problem?

There is no function

```
unsafe :: IO a -> a
```

hence all values related to side effects are "in" IO.

Everything outside IO is safe for all optimizations.

IO can be seen as

- a flag for values that came form functions with side effects
- a container for separating unsafe operations

Monad

IO is a special case of generally useful pattern

```
class Applicative m => Monad (m :: * -> *) where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
  fail :: String -> m a
```

Based on category theory

Way of meaningfully sequencing computations

1. Creating a (separated) boxed value
2. Creating functions for modifying them within the boxes

do Notation

Using monads leads to long sequences of operations chained by operators >>, >>=

```
main = putStrLn "Hello, what is your name?" >>
       getLine >>= \name ->
       putStrLn ("Hello, " ++ name ++ "!")
```

Do notation just makes these sequences more readable

(it is rewritten to monad operators before compilation)

```
main = do putStrLn "Hello, what is your name?"
          name <- getLine
          putStrLn ("Hello, " ++ name ++ "!")
```

do Notation

do is a syntax block, such as where and let

- action on a separate line gets executed
- `v <- x` runs action x and bounds the result to v
- `let a = b` defines a to be the same as b until the end of the block (no need for in)

Derived Primitives

Creating more complex IO actions from simpler

```
getLine :: IO String
getLine = do x ← getChar
             if x == '\n' then
                return []
             else
                do xs ← getLine
                   return (x:xs)
```

Derived Primitives

The same without the do notation

```
getLine2 :: IO String
getLine2 = getChar >>= \x
           -> if x == '\n' then
                 return []
              else getLine2 >>= \xs
                   -> return (x:xs)
```

Writing a string to the screen:

```
putStr :: String → IO ()
putStr []     = return ()
putStr (x:xs) = do putChar x
                   putStr xs
```

Writing a string and moving to a new line:

```
putStrLn :: String → IO ()
putStrLn xs = do putStr xs
                 putChar '\n'
```

Hangman

Consider the following version of hangman:

- One player secretly types in a word.
- The other player tries to deduce the word, by entering a sequence of guesses.
- For each guess, the computer indicates which letters in the secret word occur in the guess
- The game ends when the guess is correct.

We adopt a <u>top down</u> approach to implementing hangman in Haskell, starting as follows:

```
hangman :: IO ()
hangman = do putStrLn "Think of a word: "
             word ← sgetLine
             putStrLn "Try to guess it:"
             play word
```

The action <u>sgetLine</u> reads a line of text from the keyboard, echoing each character as a dash:

```
sgetLine :: IO String
sgetLine = do x ← getCh
              if x == '\n' then
                 do putChar x
                    return []
              else
                 do putChar '-'
                    xs ← sgetLine
                    return (x:xs)
```

The action getCh reads a single character from the keyboard, without echoing it to the screen:

```
import System.IO

getCh :: IO Char
getCh = do hSetEcho stdin False
           x ← getChar
           hSetEcho stdin True
           return x
```

The function play is the main loop, which requests and processes guesses until the game ends.

```
play :: String → IO ()
play word =
   do putStr "? "
      guess ← getLine
      if guess == word then
         putStrLn "You got it!"
      else
         do putStrLn (match word guess)
            play word
```

The function match indicates which characters in one string occur in a second string:

For example:

```
> match "haskell" "pascal"

"-as--ll"
```

```
match :: String → String → String
match xs ys =
   [if elem x ys then x else '-' | x ← xs]
```

Advanced Pattern Matching

Data constructors can be matched nested

(1, (x:xs), 'a', (2, Just y:ys))

but not x:x:xs

As pattern

f s@(x:xs) = x:s

Top-down, left-right

Matching can succeed, fail, diverge

Refutable patterns: [], Tree x l r

Irrefutable patterns: _, x, a, ~(x:xs).

Pattern Matching Divergence

Assume the infinite recursion

```
bot = bot
```

Pattern matching diverges if it tries to match `bot`

Order of definitions influences pattern matching failure

```
take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
```

```
take1 _ [] = []
take1 0 _ = []
take1 n (x:xs) = x : take1 (n-1) xs
```

Lazy Pattern

Lazy pattern `~pat` is irrefutable (always matches)

- The variable `pat` is bound only when used
- ~(x:xs) on LHS is equivalent to using head/tail on RHS
- ~(x,y) on LHS is equivalent to using fst/snd on RHS

```
> (\ ~(a,b) -> 1) bot
```

Dangerous with types with multiple constructors

Case Expressions

```
f p11 … p1k = e1
...
f pn1 ... pnk = en
```

where each `pij` is a pattern, is semantically equivalent to:

```
f x1 x2 ... xk = case (x1, ... , xk) of
    (p11, ..., p1k ) -> e1
     ...
    (pn1, ..., pnk ) -> en
```