Computational Semantics with Haskell

Yulia Zinova

Winter 2016/2017

We follow Van Eijck and Unger 2010, electronic access from the library

The programming language Haskell

- Member of Lisp family together with Scheme, ML, Occam, Clean, Erlang
- Based on lambda calculus (the whole family)
- Functions are everything in Haskell: they can be arguments and results of other functions
- Functions can be recursive
- Arguments are evaluated only when needed (if at all) – *lazy evaluation*

What we need

- An *interpreter* or *compiler*
- An interpreter is a system that allows you to execute function definitions interactively
- On computers here: use Windows 10
- On your laptop: go to `www.haskell.org/downloads` and get either the minimum package or the whole platform
- Follow the link to the GHCi (Glasgow Haskell Compiler) manual
- Task: find the command that one calls the compiler with.

First Experiments

- The prompt *Prelude* means that only the predefined functions from the Haskell Prelude are available
- Here is the Haskell wiki: `https://wiki.haskell.org/Haskell`
- First commands:
  - `:l`⟨*file name*⟩ – load a file or module
  - `:r` – to reload the currently loaded file
  - `:t`⟨*expression*⟩ – display the type of an expression
  - `:q` quit the compiler

First experiments

- Interpreter as a calculator: let us calculate the number of seconds in a year
- Try several calculations, find out the precedence order of the operations $+, -, *, \hat{}, /$
- How does the interpreter read 2^3^4?

Define your own function

- Define and use functions:

```
let square x = x * x in square 3
```

- Or use lambda abstraction:

```
(\ x -> x * x) 4
```

- Or define the function in a text file:

```
square :: Int -> Int
square x = x * x
```

Load the code

- Download `http://www.computational-semantics.eu/FPH.hs`
- Load it: `:l FPH`
- Play with the function `square`

Basic types

- Characters – `Char`, single quotes
- String – `String` (equivalent to `[Char]`), double quotes
- List of integers – `[Int]`
- Empty string = empty list
- Boolean – `Bool`

Putting items in the list

```
"Hello World!"
['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!']
'H':'e':'l':'l':'o':' ':'W':'o':'r':'l':'d':'!': []
```

What happens? What is the type of the colon operator ':'?

Putting items in the list

```
"Hello World!"
['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!']
'H':'e':'l':'l':'o':' ':'W':'o':'r':'l':'d':'!': []
```

What happens? What is the type of the colon operator ':'?

```
Char -> [Char] -> [Char]
```

Recursion

- Look at the `hword` function

Boolean operations

- Conjunction is `& &`
- Disjunction is `| |`
- Negation is `not`
- Which types do they have?
- For a prefix version of a two-place function, use brackets

Infix operators

- `bright && beautiful = (&&) bright beautiful`
- `x op y = (op) x y`
- `(op x)` – the operation resulting from applying op to its right hand side argument
- `(x op)` – the operation resulting from applying op to its left hand side argument
- http://directpoll.com/r?XDbzPBd3ixYqg8NGqyk61sB4bD4jMvNsRdQsGg7pFh

Type polymorphism

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

- Check the type of the concatenation function (++)

Recursion

- What is recursion?

Recursion

- What is recursion?
- A recursive function calls itself, but without infinite regress
- How do we make sure it tops?

Recursion

- What is recursion?
- A recursive function calls itself, but without infinite regress
- How do we make sure it tops?
- Base case that does not call the function
- Examine the function `story`. Try `putStrLn (story 5)`. What about `putStrLn (story (-1))`

List types and list comprehension

- Have look at the type of the colon operation. What does it mean?

List types and list comprehension

- Have look at the type of the colon operation. What does it mean?
- We combine an element of some type with a list of elements of the same type

```
head : : [a] -> a
head (x:_) = x

tail : : [a] -> [a]
tail (_:xs) = xs
```

List patterns

- The underscore matches any object
- The list pattern `[]` matches empty list
- The list pattern `[x]` matches any singleton list
- The list pattern `(x:xs)` matches any non-empty list
- `http://directpoll.com/r?XDbzPBd3ixYqg81uUQf0SHnX2XEtW5X2bna2QqHzPr`

Lists

- Lists can be given as ranges: `[1 . . 243]`, `['m' . . 'x']`
- This works only for ordered types!
- What do you think `[0 . . ]` will produce?

Lists

- Lists can be given as ranges: `[1 . . 243]`, `['m' . . 'x']`
- This works only for ordered types!
- What do you think `[0 . . ]` will produce?
- Use *Ctrl-C* to stop
- Try `take 5 [0 . .]`

List comprehension

- General form: `[x | x <- A, P x]`

```
[n | n <- [0..10], odd n]
[odd n | n <- [0..10] ]
[x ++ y |  x <- ["use", "faith"], y <- ["ful", "less"] ]
```

List processing

- Function `map` takes a function and a list and returns a list containing the results of applying the function to the individual list members
- What will `map (+1) [0..9]` do? And `map hword ["fish", "and", "chips"]`?
- The `filter` function takes a property and a list, and returns the sublist of all list elements satisfying the property.
- Guarded equations:

```
foo t | condition_1 = body_1
    | condition_2 = body_2
    | condition_3 = body_3
```

Composition

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(f . g) x = f (g x)
```

- If we have a Dutch-to-English and an English-to-French dictionaries and we want a Dutch-to-French dictionary, what do we do?

Quantification

```
and :: [Bool] -> Bool
and [] = True
and (x :xs) = x \&\& (and xs)

or :: [Bool] -> Bool
or [] = False
or (x :xs) = x || (or xs)

any, all :: (a -> Bool) -> [a] -> Bool
any p = or . map p
all p = and . map p
```

Type classes

- Check the type of (1)

  (1) `(\ x y -> x /= y)`
- Is there a difference between (1) and `(/=)`?
- Check the type of the function composition `all . (/=)`. How could you name it?
- Check the type of the function composition `any . (==)`. How could you name it?

Recursion: exercise

- **Exercise 3.1** Write a function that will test two infinite strings for being equal.
- **Exercise 3.2** The predefined function `min` computes the minimum of two objects if they can be ordered. Use it to define a function `minList:: Ord a => [a] -> a` for computing the minimum of a non-empty list.
- **Exercise 3.3** Define a function `delete` that removes an occurrence of an object x from a list of objects in class `Eq`. Delete only the first occurrence, if x is not in the list, do nothing.
- **Exercise 3.4** Define a function `srt :: Ord a => [a] -> [a]` that puts the minimum of the list in front of the result of sorting the list that results from removing its minimum. Empty list is already sorted.

References:

Van Eijck, J. and Unger, C. (2010). *Computational semantics with functional programming*. Cambridge University Press.