Haskell Refresher

Informatics 2D

Kobby. K.A. Nuamah
30 January 2015

Haskell

- Purely functional! : "Everything is a function"
- Main topics:
  - Recursion
  - Currying
  - Higher-order functions
  - List processing functions such as map, filter, foldl, sortBy, etc
  - The Maybe monad
- More on Haskell: *http://www.haskell.org/haskellwiki/Haskell*

Types

- Unlike other programming languages like Java, Haskell has type inference.
- However, type declarations ensures that you are specific about the input arguments of your function and the output values.
- Example:

```
next :: Trace -> [Trace]
```

- The **`next`** function takes an argument of type **Trace** and returns a list of ***Trace***s

Type Synonyms

```
type Trace = [(Int,Int)]

type Game = [Int]
```

- The type Trace is a synonym for a list of `(Int,Int)` tuples.
- For code clarity.

Recursion

- Important role in Haskell.
- Function is recursive when one part of its definition includes the function itself again.
- Always have a termination condition to avoid infinite loop.

```
length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```


Currying

- The process of creating intermediate functions when feeding arguments into a complex function.
- Note: all functions in Haskell really only take one argument
- Example:

  2 * 3 in Haskell:
  - (*) function takes first argument 2, and return an intermediate function (2*)
  - The new function (2*) takes one argument,3, and completes the multiplication
- Applying only one parameter to a function that takes two parameters returns a function that takes one parameter

Higher-Order Functions

- Functions are just like any other value in Haskell.
- Functions can take functions as parameters and also return functions.

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f ( x: xs ) = f x : map f xs
```

- Map takes a function and list and applies that function to every element in the list.

List Processing Functions
(map, filter, foldl, etc.)

- **`map`** : takes a function and list and applies that function to every element in the list.

  `map :: (a -> b) -> [a] -> [b]`
- **`filter`**: takes a predicate (function that returns true or false) and list and then returns the list of all elements that satisfy the predicate.

  `filter:: (a -> Bool) -> [a] -> [a]`
- **`foldl`**: takes a binary function, an accumulator and a list. It ‘folds’ up the items in the list and return a single value.

  `foldl:: (a -> b -> a) -> a -> [b] -> a`

List Comprehension

- Build more specific sets out of general sets.
- Example: to create a list of integers that are multiples of 2 and greater than than 20:

```
[x*2 | x <- [1..25] , x*2 >= 20]
```

Output function — Elements bound to x — Condition Or Predicate

Maybe Monad

- The Maybe monad represents computations which might "go wrong" by not returning a value.
- If a value is returned, it uses **Just a**, where a is the type of the value.
- If no value is available, it returns **Nothing.**
- **Example:**

```
safeDiv::Double->Double->Maybe Double
safeDiv x y
     | y == 0    = Nothing
     | otherwise = Just (x/y)
```


Coursework Overview

- **Trace** type for search problems

```
type Trace = [(Int,Int)]
```

- Example :
- A path from (1,1) to (4,2)

```
[(1,1),(1,2),(2,2),(3,2),(4,2)]
```

	1	2	3	4	5	6	7	8	9	10
1	x	x								
2		x								
3		x								
4		x								
5										
6										
7										
8										
9										
10										

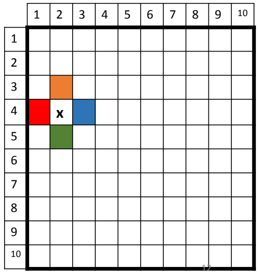
Successor Function

- The next function returns the possible continuations of the path

```
next::Trace->[Trace]
```

- Example :
- Suppose we start from are at (4,2)
- Possible continuations generated by **next**

```
[[(1,1),(1,2),(2,2),(3,2),(4,2),(4,1)],
 [(1,1),(1,2),(2,2),(3,2),(4,2),(3,2)],
 [(1,1),(1,2),(2,2),(3,2),(4,2),(4,3)],
 [(1,1),(1,2),(2,2),(3,2),(4,2),(5,2)]]
```



Consistency with representation

- Be consistent with your representation of Traces in Haskell

```
[(1,1),(1,2),(2,2),(3,2),(4,2)]

[(4,2),(3,2),(2,2),(1,2),(1,1)]
```

- Both are ok, provided you are consistent with the head and tail of your list.
- Same applies to [Trace]

Higher-Order Functions in Coursework

```
Example:
bestFirstSearch::(Trace -> Bool) -> (Trace ->[Trace]) ->
((Int,Int) -> Int)->[Trace] -> Maybe Trace
```

- **(Trace → Bool)** is the type of the goal function (same as uninformed search).
- **(Trace → [Trace])** is the type of the next function (same as uninformed search).
- **((Int,Int) → Int)** is the type of the heuristic function, which defines at least an ordering on the nodes in the search agenda.
- **[Trace]** is the search agenda (same as uninformed search).
- **Maybe Trace** is the value the function returns (same as uninformed search).

Game (Tic-Tac-Toe) Representation

- Game represented as a list of Integers
  `type Game = [Int]`
- A new game will be represented as
- [-1,-1,-1,-1,-1,-1,-1,-1,-1]

- Max player is represented by a **1** in the list.
- Min player is represented as **0** in the list.
- An unplayed cell is represented as **-1**

- Types for Cell and Player
  `type Player = Int`
  `type Cell = (Int,Int)`

X	O	X
O	X	O
O	O	X

Game Representation Examples

- New Game: `[-1,-1,-1,-1,-1,-1,-1,-1,-1]`

- Min Move: `[-1,-1,-1,-1, 0,-1,-1,-1,-1]`

	O	

- Max Move: `[ 1,-1,-1,-1, 0,-1,-1,-1,-1]`

X		
	O	

Lines in Game

- The Line type represents any of the lines on the game board: rows, columns and diagonals.

```
type Line = [Int]
```

- Examples of Lines for the game state given:
- Row 1: `[1,0,1]` Row 3: `[0,0,1]`
- Column 1: `[1,0,0]` Diagonal 1: `[1,1,1]`

X	O	X
O	X	O
O	O	X

- To get all lines for a game state, use function:

```
getLines::Game->[Line]
```


Other useful functions

- maxPlayer function checks if the given player is max, and returns a Boolean.
  `maxPlayer::Player->Bool`
- switch function alternates between players.
  `switch::Player->Player`
- terminal function checks if the game argument is in a terminal state.
  `terminal::Game->Bool`
- isMoveValid checks if a move made in a given game state is a valid one for a given player.
  `isMoveValid::Game->Player->Cell->Bool`
- playMove makes a move to a cell and returns the new game state. This functions is called for human player moves.
  `playMove::Game->Player->Cell->Game`
- moves function returns a list of possible moves/successor states that a player can make given a game state.
  `moves::Game->Player->[Game]`
- checkWin function checks if the game state is a win for the player argument.
  `checkWin::Game->Player->Bool`