Programming in Haskell
Aug-Nov 2015

LECTURE 1

AUGUST 4, 2015

S P SURESH, http://www.cmi.ac.in/~spsuresh
CHENNAI MATHEMATICAL INSTITUTE

Administrative

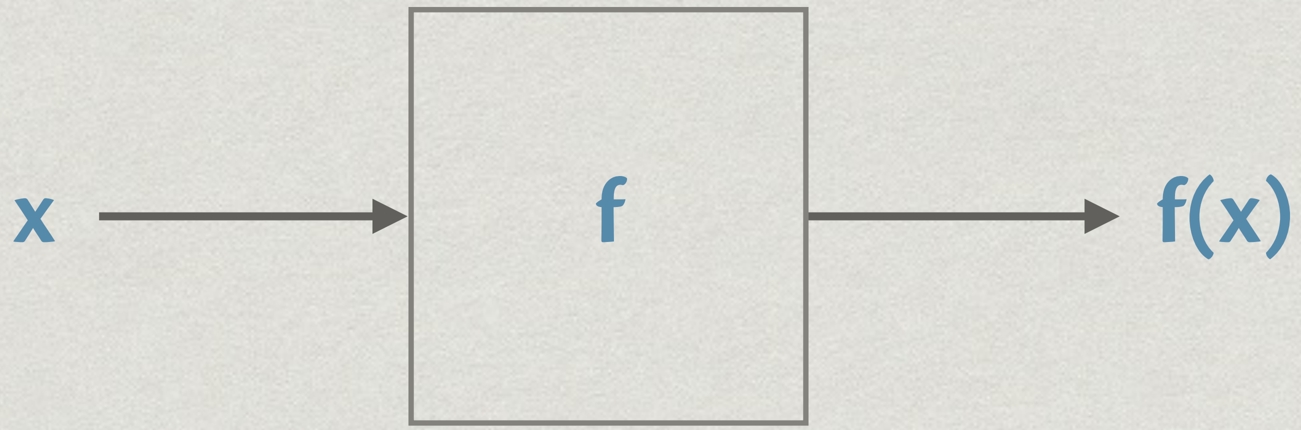
- Tuesdays and Thursdays 10.30 at NKN Hall
- Evaluation: Quizzes, 5 – 6 programming assignments, endsem, midsem
- TAs: Pranshu Bhatnagar, Anish Sevekari, Thejaswini
- Moodle page: http://moodle.cmi.ac.in/course/view.php?id=134
- Course page: http://www.cmi.ac.in/~spsuresh/teaching/prgh15

Resources

- http://www.haskell.org
- Introduction to Functional Programming using Haskell (Richard Bird)
- Thinking Functionally with Haskell (Richard Bird)
- Real World Haskell http://book.realworldhaskell.org/read/
- Learn You a Haskell for Great Good! http://learnyouahaskell.com/chapters
- Plenty of other resources

Programs as functions

- Functions transform inputs to outputs



- Program: rules to produce output from input
- Computation: process of applying the rules

Building up programs

How do we describe the rules?

- Start with built in functions
- Use these to build more complex functions

Building up programs …

Suppose

- … we have the whole numbers, `{0,1,2,…}`
- … and the successor function, `succ`

```
succ 0 = 1
succ 1 = 2
succ 2 = 3
…
```

- Note: we that write `succ 0`, not `succ(0)`

Building up programs ...

We can compose `succ` twice to build a new function

- `plusTwo n = succ (succ n)`

If we compose `plusTwo` and `succ` we get

- `plusThree n = succ (plusTwo n)`

Building up programs …

How do we define plus?

- `plus n m` means apply `succ` to `n`, `m` times
  - Again note: `plus n m`, not `plus(n,m)`
- `plus n 1` = `succ n`
  `plus n 2` = `succ (plus n 1)` = `succ (succ n)`
  …
  `plus n i` = `succ(succ(…(succ n)…)`
  i times
- How do we capture this rule for all `n`, `i`

Inductive/recursive definitions

- `plus n 0 = n`, for every `n`
- `plus n 1` = `succ n = succ (plus n 0)`
- Assume we know how to compute `plus n m`
- Then, `plus n (succ m)` is `succ (plus n m)`

Computation

* Unravel the definition

*
```
plus 7 3
= plus 7 (succ 2)
= succ (plus 7 2)
= succ (plus 7 (succ 1))
= succ (succ (plus 7 1))
= succ (succ (plus 7 (succ 0)))
= succ (succ (succ (plus 7 0)))
= succ (succ (succ 7))
```

Inductive/recursive definitions

- `plus n 0 = n`, for every `n`
- `plus n 1` = `succ n = succ (plus n 0)`
- Assume we know how to compute `plus n m`
- Then, `plus n (succ m)` is `succ (plus n m)`

Recursive definitions …

Multiplication is repeated addition

- `mult n m` means apply `plus n`, `m` times
- `mult n 0 = 0`, for every `n`
- `mult n (succ m) = plus n (mult n m)`

Summary

- Functional programs are rules describing how outputs are derived from inputs
- Basic operation is function composition
- Recursive definitions allow repeated function composition, depending on the input

Building up programs

* Start with built in functions
* Use function composition, recursive definitions to build more complex functions
* What kinds of values do functions manipulate?

Types

Functions work on values of a fixed type

- `succ` takes a whole number as input and produces a whole number as output
- `plus` and `mult` take two whole numbers as input and produce a whole number as output
    - Can also define analogous functions for real numbers

Types

How about `sqrt`, the square root function?

- Even if the input is a whole number, the output need not be—may have a fractional part
- Number with fractional values are a different type from whole numbers
  - In Mathematics, whole numbers are often treated as a subset of fractional or real numbers

Types

Other types

- `capitalize ‘a’ = ‘A’`, `capitalize ‘b’ = ‘B’`, …
- Inputs and outputs are letters or “characters”

Functions and types

- We will be careful to ensure that any function we define has a well defined type
    - The function `plus` that adds two whole numbers will be different from another function `plus` that adds two fractional numbers

Functions have types

- A function that takes inputs of type A and produces output of type B has a type A → B
  - In Mathematics, we write f: S → T for a function with domain S and codomain T
  - A type is a just a set of permissible values, so this is equivalent to providing the type of f

Collections

* It is often convenient to deal with collections of values of a given type
  * A list of integers
  * A sequence of characters — words or strings
  * Pairs of numbers
* Such collections are also types of values

Summary

- Functions manipulate values
- Each input and output value comes from a well defined set of possible values — a type
- We will only allow functions whose type can be defined
  - Functions themselves inherit a type
- Collections of values also types

Haskell

- A programming language for describing functions
- A function description has two parts
  - Type of inputs and outputs
  - Rule for computing outputs from inputs
- Example

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

Type definition
Computation rule

Basic types

* `Int`, Integers
  * Operations: `+`, `-`, `*`, `/` (Note: `/` produces `Float`)
  * Functions: `div`, `mod`
* `Float`, Floating point ("real numbers")
* `Char`, Characters, `'a'`, `'%'`, `'7'`, …
* `Bool`, Booleans, `True` and `False`

Basic types ...

- `Bool, Booleans, True and False`
- Boolean expressions
  - Operations: `&&`, `||`, `not`
  - Relational operators to compare `Int`, `Float`, ...
    - `==`, `/=`, `<`, `<=`, `>`, `>=`

Defining functions

- `xor` (Exclusive or)
  - Input two values of type `Bool`
  - Check that exactly one of them is `True`

```
xor :: Bool -> Bool -> Bool
xor b1 b2 = (b1 && (not b2)) ||
            ((not b1) && b2)
```

(why?)

Defining functions

- `inorder`
    - Input three values of type `Int`
    - Check that the numbers are in order

```
inorder :: Int -> Int -> Int -> Bool
inorder x y z = (x <= y) && (y <= z)
```

Pattern matching

- Multiple definitions, by cases

```
xor :: Bool -> Bool -> Bool
xor True  False = True
xor False True  = True
xor b1    b2    = False
```

- Use first definition that matches, top to bottom
  - `xor False True` matches second definition
  - `xor True True` matches third definition

Pattern matching ...

- When does a function call match a definition?
  - If the argument in the definition is a constant, the value supplied in the function call must be the same constant
  - If the argument in the definition is a variable, any value supplied in the function call matches, and is substituted for the variable (the “usual” case)

Pattern matching ...

- Can mix constants and variables in a definition

```
or :: Bool -> Bool -> Bool
or True  b    = True
or b     True = True
or b1    b2   = False
```

- `or True False` matches first definition
- `or False True` matches second definition
- `or False False` matches third definition

Pattern matching …

- Another example

```
and :: Bool -> Bool -> Bool
and True  b = b
and False b = False
```

- In the first definition, the argument supplied is used in the output

Recursive definitions

- Base case: f(0)
- Inductive step: f(n) defined in terms of smaller values, f(n-1), f(n-2), …, f(0)
- Example: factorial
  - $0! = 1$
  - $n! = n \times (n-1)!$

Recursive definitions ...

- In Haskell

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * (factorial (n-1))
```

- Note the bracketing in `factorial (n-1)`
  - `factorial n-1` would be read as `(factorial n) - 1`
- No guarantee of termination: what is `factorial (-1)`

Conditional definitions

- Use conditional expressions to selectively enable a definition
- For instance, "fix" `factorial` for negative inputs

```
factorial :: Int -> Int
factorial 0 = 1
factorial n
  | n < 0 = factorial (-n)
  | n > 0 = n * (factorial (n-1))
```

Conditional definitions ..

```
factorial :: Int -> Int
factorial 0 = 1
factorial n
  | n < 0 = factorial (-n)
  | n > 0 = n * (factorial (n-1))
```

- Second definition has two parts
  - Each part is guarded by conditional expression
  - Test guards top to bottom
  - Note the indentation

Conditional definitions ..

```
factorial :: Int -> Int
factorial 0 = 1
factorial n
  | n < 0 = factorial (-n)
  | n > 0 = n * (factorial (n-1))
```

- Multiple definitions can have different forms
  - Pattern matching for `factorial 0`
  - Conditional definition for `factorial n`

Conditional definitions ...

- Guards may overlap

```
factorial :: Int -> Int
factorial 0 = 1
factorial n
  | n < 0 = factorial (-n)
  | n > 1 = n * (factorial (n-1))
  | n > 0 = n * (factorial (n-1))
```

Conditional definitions …

- Guards may not cover all cases

```
factorial :: Int -> Int
factorial 0 = 1
factorial n
  | n < 0 = factorial (-n)
  | n > 1 = n * (factorial (n-1))
```

- No match for `factorial 1`

```
Program error: pattern match failure: factorial 1
```

Summary

- A Haskell function consists of a type definition and a computation rule
- Can have multiple rules for the same function
  - Rules are matched top to bottom
  - Use patterns, conditional expressions to split cases

Running Haskell programs

- Haskell interpreter ghci
  - Interactively call builtin functions
  - Load user-defined Haskell code from a text file
  - Similar to how Python works

Setting up ghci

- Download and install the Haskell Platform
    - `https://www.haskell.org/platform/`
    - Available for Windows, Linux, MacOS

Using ghci

- Create a text file (extension .hs) with your Haskell function definitions
- Run ghci at the command prompt
- Load your Haskell code
  - `:load myfile.hs`
- Call functions interactively within ghci

Caveats

- Cannot define new functions directly in ghci
  - Unlike Python
- Must create a separate .hs file and load it

Compiling

- ghc is a compiler that creates a standalone executable from a .hs file
  - ghc stands for Glasgow Haskell Compiler
  - ghci is the associated interpreter
- Using ghc requires some advanced concepts
  - We will come to this later in the course

Summary

* ghci is a user-friendly interpreter
    * Can load and interactively execute user defined functions
* ghc is a compiler
    * But we need to know more Haskell before we can use it