Functional programming

Introduction

- High level of abstraction
- Based on lambda calculus
- Language of choice: Haskell
- Functional programming -> expressions > statements.

Example:

```
--Add the first ten numbers together
sum[1..10]
```

- Install `Hugs`, Haskell interpreter (`ghci` is another interpreter)

First steps

- Start `hugs` in terminal

Examples:

```
> head[1,2,3,4] --take first element
1
> tail[1,2,3,4] --remove first element
[2,3,4]
> [1,2,3,4] !! 2 --element #2
3
> take 3 [1,2,3,4] -- generalization of head
[1,2,3]
> drop 3 [1,2,3,4,5] --generalization of tail
[4,5]
> [1,2,3]++[4,5] --append
[1,2,3,4,5]
```

- First element in list has index 0
- List different to array -> indexing bad idea, not in constant time but in linear

Function application: function application is denoted by space. Higher priority

```haskell
f a b + c*d --f(a,b) + c d from math
f a + b --f(a) + b from math
```

- Haskell file (script) –> `.hs`
- Define function in script, then open Hugs with script as argument so that functions are available. If script is changed use `:reload`. Also possible to load using `:load` *script*
- *Infix operator*: `xfy --> f x y`

- *Naming*:
  - function and parameter name must begin with lowercase
  - can use quotes (*prime*)
  - type has to start with uppercase
  - convention –> `s` at the end means list, `ss` list of lists

- Indentation like Python, implicit grouping
- Useful commands –> `:load` *script*, `:reload`, `:edit` *script*, `:type` *expression*,`:?`
- Comments: one line `--comment`, nested:

```haskell
{-
very long
comment goes
here
-}
```

Types and classes

- **Type**: name for a collection of related values. Example `Bool`
- Applying a function to a wrong type makes a *type error*
- `e :: t` –> `e` has type `t`
- *Type inference*: compiler calculates type of expression prior to execution. Haskell programs are *type safe*, type error never happens in run time
- `:type <exp>` to calculate type of expression

Type	Explanation
`Bool`	Logical value: `True` or `False`
`Char`	Single character, enclosed in sigle quotes: `'a'`
`String`	String of characters, double quotes: `"abc"`
`Int`	Fixed precision integer

Type	Explanation
`Integer`	Arbitrary precision integer, doesn't overflow
`Float`	Single precision floating point number

Table 1: Different types in Haskell

- **List**: sequence of values with same type. Can be infinite. Examples:

```
[False, True, False]::[Bool] -- list of elements type Bool
['a', 'b']:: [Char] -- list of elements type char
```

- **Tuple**: sequence of values of different type. Number of elements is called *arity*. Finite number of elements because type of all have to be calculated. Example:

```
(False, 'a') :: (Bool, Char) -- length appears in type
```

- **Function**: mapping from values of a type to values of another type. Examples:

```
not :: Bool -> Bool
isDigit :: Char -> Bool

function :: t1 -> t2 -- from domain to range in general
```

- *Curried function*: functions that return arguments one at a time (functions can return functions): `a -> (a -> a)` equivalent to `a -> a -> a`, arrow associates to the right. Any function that returns more than one values can be curried. Useful for *partially applying functions*. **Most** functions applied in curried form, if tuples are not explicitly declared.
- *Polymorphic function*: functions not defined for a particular type. Example:

```
length :: [a] -> Int
```

- Price for polymorphism: type variables start with lowercase and types with uppercase
- *Overloaded function*: functions with same name but different types. In Haskell overloading means that there is a restriction in the type class. Example:

```
sum :: Num [a] => [a] -> Int -- only numeric values allowed
```

Defining functions

Conditional expressions

Example:

```
abs :: Int -> Int
abs n = if n >=0 then n else -n
```

- Can be nested
- Conditional expressions **must** have an `else` branch

Guarded equation:

- Sequence of logical expressions
- Alternative to conditional (Haskell people prefer this)

```
abs n | n >= 0     = n -- |= such that
      | otherwise = -n
```

- Can be used to make definitions involving multiple conditions

Pattern matching

```
(&&) :: Bool -> Bool -> Bool
True && True = True
_    && _    = False --anything but True&&True
```

More efficient way using wildcard + lazy evaluation:

```
(&&) :: Bool -> Bool -> Bool
True  && b = b -- True && something --> something
False && _ = False -- always False
```

- Order is important
- Patterns may not repeat variables: all the variables inside the pattern have to be different
- *Lists in pattern matching*: use *cons* definition `(:)`[1]. Only matches not empty list. These pattern must be parethesized because function application has higher priority.

```
head :: [a ] → a
head (x : ) = x
```

[1] Lists are constructed one element at a time from the empty list using *cons operator* `[1,2,3] = 1:(2:(3:[]))= 1:2:3:[]`

Lambda expressions

Functions can be constructed without naming using *lambda expressions*:

```
\x -> x + x --\x = \lambda x (from lambda calculus)
```

- Useful for currying:

```
add x y = x + y
add = \x -> (\y -> x+y)
```

- For returning functions as results
- For avoiding naming functions only used once:

```
odds n = map f [0..n-1]
         where
          f x = x*2 +1

odds n = map (\x -> x*2 + 1)[0..n-1] --pass lambda as parameter to map
```

Sections

Operator written between two arguments can be used in curried way using parenthesis:

```
--Examples:
(1+) --sucessor
(1/) --reciprocate
(*2) --double
(/2) --half
```

For avoiding naming.

List comprehensions

- Code that manipulates collections
- Favorite collection for mathematicians: **sets**. Problems with sets:
  - No duplication
  - Deal with equality

- Haskell has tricks to deal with sets as lists
- Set comprehensions in Math: $\{x^2|x \in \{1, ..., 5\}\}$
- List comprehension in Haskell: `[x^2 | x <- [1..5]]`
- `x <- [1..5]` is called *generator*
- Comprehensions can have multiple generators (similar to nested loop, x is the outer loop and y is the inner):

```
> [(x,y) | x <- [1,2,3], y <- [4,5]]
> [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
> [(x,y) | y <- [4,5],  x <- [1,2,3]]
> [(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

- Generators can depend on each other, as in loops:

```
> [(x,y) | x <- [1,2,3], y <- [x..3]]
> [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

- Very concise code
- Filters (*guards*): `[x | x <- [1..10], even x]`

The zip function

- Combines two list to a list of pairs
- Useful when programming with list comprehensions.

```
zip :: [a] -> [b]-> [(a,b)]
```

String comprehension

- Strings = character list `[Char]`
- Everything that can be done in lists will work with strings

Recursive functions

- *Tail call elimination*

```
factorial 1 = 0
factorial n = n * factorial (n-1)
```

- Some functions are simpler to define using recursions
- *Induction* can be used to prove porperties of recursive functions
- Recursion can be used also in lists:

```
product :: [Int] -> Int
product [] = 1
product (n:ns) = n * product ns
```

Note: `:` appends element to list, `++` concatenates lists

- *Quicksort*: algorithm for sorting integers. Two rules:
  - The empty list is already sorted
  - Bolzano in the rest

```
qsort :: [Int] -> Int
qsort [] = []
qsort (x:xs)=
  qsort smaller ++ [x] ++ qsort larger
  where
    smaller = [a | a <- xs, a <= x]
    larger  = [b | b <- xs, b > x]
```

Higher-order functions

Higher-order functions are functions that take functions as arguments or return functions as results.

Useful for:

- *Programming idioms*, avoid repetition
- *Domain specific languages*
- *Algebraic properties* to reason about programs

Map

Applies function to every element in list

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

Map can be defined using list comprehension:

```
map f xs = [f x | x <- xs]
```

Or recursively (for abstraction):

```
map f [] = []
map f (x:xs) = f x : map f xs
```

Filter

Removes elements that don't satisfy a predicate

```
filter :: (a -> Bool) -> [a] -> [a]
```

Definition using list comprehension:

```
filter p xs = [x | x <- xs, p x]
```

Or recursively:

```
filter p [] = []
filter p (x:xs)
  | p x       = x:filter p xs --cons
  | otherwise = filter p xs --forget about xs
```

Foldr & foldl

Homomorfism over list -> generalization of sum, product…

- r: from the right
- l: from the left

```
f []    = v
f (x:xs)= x (+operator) f xs
```

Examples:

```
sum = foldr (+) 0
product = foldr (*) 1
and = folder (&&) True
```

Replaces the empty list by `v` and cons by `f`

Useful for:

- For defining recursive functions
- Properties of functions defined usig foldr can be proved using algebracic properties
- Program optimization

Other library functions

Composition: combines two functions into one

```
(.) :: (b -> c)-> (a -> b) -> (a -> c)
f . g = \x -> f(g x) --first apply g, apply f to result
```

Example:

```
odd :: Int -> Bool
odd = not . even
```

Use it sparingly because it's difficult to read

all: decides if every element in a list satisfies condition **any**: decides if every any in a list satisfies condition **takeWhile**: takes elements while condition is true **dropWhile**: drop elements while condition is true

Functional parsers and monads

Parser: program hat analyses piece of test and determines its syntantic structure

```
type Parser = String -> Tree

--If there is unused output
type Parser = String -> (Tree, String)

--If there are more than one option of parsing
type Parser = String -> [(Tree, String)]

--Parsers can produce any type
type Parser = String -> [(a, String)]
```

Basic parsers

Examples

```
--Parsers single character

item :: Parser Char
item = \inp -> case inp of
                    [] -> []
                    (x:xs) -> [(x,xs)]
```

```haskell
--case for pattern matching in the body of definition

--Always succeeds

return :: a -> Parser a
return v = \inp -> [(v,inp)]

--Always fails

failure :: Parser a
failure = \inp -> []
```

The function `parse` applies parser to input:

```haskell
parse :: Parser a -> String -> [(a,String)]
parse p inp = p inp
```

Sequencing

A sequence of parsers can be combined using `do`:

```haskell
p :: Parser (Char, Char)
p = do x <- item
       item
       y <- item
       return (x,y)
```

- Parser can be combined using `+++`(*else*) , if the first one fails, apply the second and so on
- Layout rule!
- If one parser fails, all fail

Derived primitives

Using the three basic parsers we can define parser that returns single characters that return characters that satisfy a given predicate:

```haskell
sat :: (Char -> Bool) -> Parser Char
sat p = do x <- item
           if p x then return x else failure
```

Using `sat` and different predicates we can define parsers for digits, lower-case letters…

Interactive programs

Interaction with keyboard and screen

Problem: Haskell programs have no side effects. Same arguments give same results. `Readline`, for example, does not give the same result all the time!

Solution: new type `IO a`, actions that have side effects. Function that return `void`: `IO ()` (empty tuple)

Basic actions

Actions in the standard library. Examples:

```
getChar :: IO Char --reads character from standard input
putChar :: IO Char --writes character to standard output
```

Sequencing

Works like parsing

Derived primitives

We can read a string from standard input basing on `getChar`, for instance.

Declaring types and classes

- We can define new types in Haskell
- Things in common with objects in Object Oriented Programming

Type declarations

Using `type` synonym between types can be defined. Example:

```
type String = [Char]
```

- Type definition can be used for better readability
- Type declarations can be nested, but they can't be recursive[2].

[2] Recursive declaration is only valid in nominal types (see Data declaration)

Data declarations

Using `data` completely new types can be defined giving their values. Example:

```
data Bool = False | True --Boolean types can be True or False
```

- These can be declared recursively, because are data and types at the same time.
- The new values of the type are called **constructors** in this case.
- Types and constructors names must begin with capital letter.
- The same constructor name cannot be used in more than one type.
- Values of new types can be used as built-in types
- The construtors can also have arguments
- Data declarations can be parameterised

Recursive types

New types can be declared in terms of themselves, if declared using `data`:

```
data Nat = Zero | Succ Nat --Creates an infinite sequence of values
```

Class and instance declaration

Classes are declared using `class`. Example:

```
class  Eq a  where
   (==), (/=) :: a -> a -> Bool

       -- Minimal complete definition:
       --      (==) or (/=)
   x /= y     =  not (x == y)
   x == y     =  not (x /= y)
```

For a type `a` to be an instance of a given class it has to comply – with the conditions in the class definition. For instancing:

```
instance Eq Bool where
  False == False = True
  True == True   = True
  _ ==_          = False
```

- Only types declared using `data` can become classes.

- Classes can also be extended to make new classes (inheritance) using `=>`
- For making a type instance of a built-in type use `deriving`. Example:

```
data Bool = False | True
          deriving(Eq, Ord, Show, Read)
```

Lazy evaluation

Everything uses lazy evaluation except you make things strict. Features of Haskell evaluation:

1. Avoid *unnecessary evaluation*
2. Allow programs to be *more modular*
3. Allows *infinite lists*

Expressions are evaluated by applying definitions until no further simplification is possible:

```
square n = n*n
square (3 + 4) --> square 7 --> 7*7 --> 49
```

If there are two different ways to evaluate an expression, the two of them will give the same result (*pure language*, no side effects). Makes it easier to refactor Haskell code.

Termination

- **Innermost reduction**: evaluates from the expression that has no expression inside
- **Outermost reduction**: evaluates from the expression that is not contained in any expression
  - May give a result when innermost fails to terminate
  - If there exists any reduction that terminates, outermost terminates
  - May require more steps because of duplication, this problem can be solved using **pointers** for **sharing**

Lazy evaluation : outermost reduction with sharing

Infinite lists

Lazy evaluation allows infinite lists, while innermost evaluation does not terminate in this case.

Lazy evaluation

> Using lazy evaluation, expressions are only evaluated as much as required to produce the final result.

So, an infinite list is only a *potentially infinite list* because it is only evaluated as needed.

Modular programming

We can generate finite list for taking elements from an infinite list, this allows *data control*