In Haskell functions can be defined in terms of themselves, in order to achieve recursion

EX

Function defined using other functions

fac: Int -> Int

fac n = product [1…n]

function defined by itself

fac 0 = 1

fac n = n \* fac (n-1)

the word diverge is used when the function does not terminate, so for example in the above example if a negative integer is used, the base case is never reached and causes a stack overflow error

Can use mathematical induction to prove recursive functions

Can define functions in terms of one another

Advice

* Define the type, helps figure out exactly what the function wants to do
* Enumerate the cases
  + # of base cases
* Define the simple cases (base case and recursive case)
* Generalise

Define type

(takes an integer, and a list and returns a polymorphic (a) type)

* drop :: Int -> [a] -> [a]

enumerate the cases

* 4
* Drop 0 [] =
* Drop 0 (x:xs) =
* Drop n [] =
  + Empty list is the safe version
  + Can leave the case undefined so that the program has an error, leaves it for the user
* Drop n (x:xs) =

Define the cases

* Drop 0 [] = []
* Drop 0 (x:xs) = x:xs
* Drop n [] = []
* Drop n (x:xs) = drop (n-1) xs

Generalise

* Can use any integral type (int and integer, don’t need to restrict to Int)
* Drop :: Integral b -> b -> [a] -> [a]
* Can get rid of unnecessary variables, that aren’t really taken into account with that base case, for example with drop n [] it doesn’t matter what n is so can be replaced with an underscore (\_)