Defining Haskell Values

- Functions definitions are written as equations
- double x = x + x

  quadruple x = double (double x)
- compute the length of a list

  – length, if empty, is zero

  length [] = 0

  – length if not empty is one plus the length of its tail

  length)x:xs) = 1 + length xs

Type Polymorphism

- What is the type of length?

  > length [1,2,3]

  3

  > length ['a','b','c','d']

  4

  > length [[],[1,2],[3,2,1],[],[6,7,8]]

  5
- length works for lists of elements of arbitrary type

  length :: [a] -¿ Int

  Here, 'a' denotes a type variable, so the above reads as "length takes a list of arbitray type a and returns an Int"

Laziness

- from n = n : (from (n+1))

  This recursive definition generates an infinite list of ascending numbers
- take n list - return first n elements of list
- take 10 (from 1)

  Haskell is a lazy language, so values are evaluated only when needed

Program Compactness

- Sorting the empty lsit gives the empty list

  qsort [] = []

  qsort (x:xs)

  = qsort [y — y ¡- xs, y ¡ x]

  ++ [x]

  ++ qsort [z — z ¡- xs, z ¿= x]

- Haskell list comprehensions

  [y — y ¡- xs, y ¡ x] "build list of ys, where y is drawn from xs such that y ¡ x"

Patterns in Mathematics

We characterise things in maths by the laws they obey, laws which often look like patterns:

$0! = 1$

$n! = n * (n-1)!$, n ¿ 0

len($\langle\rangle$) = 0

len(L1 L2) = len(L1) + len(L2)

$\langle\rangle$ denotes an empty list and L1L2 is a list concatenation

Factorial as Patterns

Maths:

$0! = 1$ $n! = n * (n-1)!$

Haskell (without patterns):

```
factorial_nop n  =  if n==0 then 1 else n * factorial_nop (n-1)
```

Haskell (with patterns):

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

Formal argument 0 is shorthand saying check the argument to see if it is zero. If so, do the righthand

Formal argument n says take the argument and refer it to the righthand side as n

Lists in Haskell There is a standard approach to constructing lists:

- Empty list using []

- Given a term x and a list xs we can construct a lsit consisting of x followed by xs as follows: x:xs

- So the lsit 1,2,3 can be built as 1:2:3:[]. Brackets show how it is built up: 1:(2:(3:[]))

- We can write [1,2,3] as shorthand for the above list

- Lists can contain charachters ['H', 'E', 'L', 'L', 'O'] but can be written shorthand as "HELLO".

  and : are list constructors. ":" is pronounced "cons"

Length with Patterns

Math:

len($\langle\rangle$) = 0

len(L1L2) = len(L1) + len(L2) len($\langle_{\rangle}$) = 1 len($\langle_{\rangle}$L) = 1 + len(L)

Haskell:

mylength [] = 0 mylength (x:xs) = 1 + mylength xs

The key idea in pattern-matching is that the syntax used to build values can also be used to look at a value, determine how it was built and extract out the individual sub-parts if required.

Compact "Truth Tables"

Patterns can be used to give an elegant expression to certain functions, for instances we can define a function over two boolean arguments like this:

myand True True = True myand _ _= False

Here, the underscore pattern "_" is a wildcard that matches anything

Types

Haskell is strongly typed - every expression/vale has a well-defined type:

myExpr :: MyType

"The value myExpr has type MyType"

Haskell supports type-inference. We dont have to declare types of functions in advance.

Atomic Types

Some Atomic types builtin to Haskell

- () - the unit type has only one value, also written as ().
- Bool - boolean values, of which there are only two: true and false
- Char - charachter values, representing Unicode charachters
- Int - fixed-precision integer type
- Intger - infinite-precision integer ttoe
- Float - floating point number of precision at least that of IEEE single-precision
- Double - floating point of precision at least that of IEEE double-precision

Composite Types

A type built on top of another type

- Functions have a type that indicates the type(s) of its input(s) and the type of its output
- Tuples gather values of other types together in a package
- Algebraic are data types that allow you to definte types whose valyes can have more than one form

Tuples

- We can create pairs, triples, n-tuples of values: (1,2) or (1, 'a', "Hi!")
- The type of the pair (42, 'z') where 42 has type Int, is (Int, Char)

  e.g. (42 'z') :: (Int,Char)

- We can use tuples as patterns in function definitions:

  sumPair (a,b) = a + b

Algebraic Data Types

Haskell lists are an example, with data that can have one of two forms:

Empty [] or non-empty (x:xs)

Function Types

A function type consists of the input tyoe, follow by a right-arrow and then the output type

myFun :: MyInputTyoe -¿ MyOutputType

like f: A -¿ B in maths

sumInt :: [Integer] -¿ Integer adds up a lsit of integers

Consider a rounding function that converts a floating point number to a fixed-width integer:

round :: Double -¿ Int

Inferring Function Types

FunDef - Given a function declaration like f x = e, if e has type b and (the use of) x (in e) has type a, then f must have type a -¿ b.

FunUse - Given a function applicayion f v, if f has type a -¿ b, then v must have type a, and f v will have type b.