A Quick Tour of Haskell

Fast Track to Haskell

Andres Löh, Edsko de Vries

Well-Typed
The Haskell Consultants

Overview

- Short overview of Haskell concepts.
- Many aspects of the language (all at once).
- Not a lot of detail (yet).

Setup

Make sure you have:

- an editor window open with the Haskell source file accompanying this Quick Tour;
- GHCi open with that file loaded.

Starting point

Running example problem

Given a particular number together with a list of numbers, check whether the number is contained in the list.

For example:

- Given `7` together with `6`, `9` and `42`, the answer is “no”.
- Given `9` with the same list, the answer is “yes”.

Lists

List are one of Haskell's most important data structures.

They are defined inductively:

- The empty list `[]` is a list.
- Given a single element `y` and a list `ys`, we can construct a new list `y : ys` (pronounced `y` cons `ys`).

Example:

```
6 : (9 : (42 : []))
```

```
6 : 9 : 42 : []
[6, 9, 42]
```

All three expressions are equivalent (syntactic sugar).

Deconstructing lists

In the same way we can construct lists, we can deconstruct them: given an arbitrary list, it must

- either be the empty list `[]`,
- or of the shape `y : ys` for some head element `y` and some tail list `ys`.

This process of deconstruction is called pattern matching and is at the heart of defining functions in Haskell.

Defining a function on lists

```
elem x []       = False
elem x (y : ys) = x == y || elem x ys
```

Note the following:

- Two equations, two cases.
- First line says: No element `x` is contained in the empty list.
- In Haskell, we say “no” by using `False`.
- Second line: operator `==` compares two things, `||` is logical or.
- The symbol `=` is used for definitions, whereas `==` is a comparison operator.
- The function calls itself recursively on the tail.
- Semantics of `||`: stop as soon as the first operand evaluates to `True`.

Evaluation

Haskell evaluates expressions by rewriting them according to definitions until they can no longer be simplified.

The resulting final expression is called a value.

Example:

```
   elem 9 [6, 9, 42]
⇝ elem 9 (6 : (9 : (42 : [])))
⇝ 6 == 9 || elem 9 (9 : 42 : [])
⇝ False  || elem 9 (9 : 42 : [])
⇝ elem 9 (9 : 42 : [])
⇝ 9 == 9 || elem 9 (42 : [])
⇝ True   || elem 9 (42 : [])
⇝ True
```

Remember:

```
elem x []       = False
elem x (y : ys) = x == y || elem x ys
```

Equational reasoning

Calculations with programs are generally possible in Haskell.

They are performed often to reason about programs, or to transform programs into more efficient equivalent programs.

This process is also called equational reasoning.

The definition of "or"

Truth values are not really special.

They are a datatype, like lists.

A truth value is either

- the value True,
- or the value False.

Once again, we can use pattern matching to define functions.

Example: the definition of "or":

```
True  || y = True
False || y = y
```

Note:

- infix operators can be defined,
- equations correspond to the rules we already used.

Lazy evaluation

Look at the definition of "or" again:

$$\text{True} \;\; || \; y = \text{True}$$
$$\text{False} \; || \; y = y$$

In many languages, shortcut behaviour for "or" is a special hack.

In Haskell, lazy evaluation is the general default:

- only the first argument is needed to decide which equation of $||$ applies,
- arguments are only evaluated when required (usually, by pattern matching).

Static types

Haskell is a statically typed language:

- every expression is first type-checked,
- only if the expression can be assigned a valid type, the program can be run.

Question

How is this compatible with the fact that we have not seen any type declarations so far?

Type inference

A mechanical form of applying common sense:

- If you know the type of some expressions, you can check whether they are used consistently.
- You can conclude information about the type of an expression from the types of the subexpressions.

Example

We know:

- `2` is a number,
- `[]` is a list,
- `:` is an operator that takes a number and a list to a list.

We can conclude that `2 : []` is a type-correct list.

We can also conclude that `2 : 3` cannot be correct, because the right argument of “cons” is a number and not a list.

Types are important

Type annotations in Haskell are optional, but

- it is allowed to specify types of functions explicitly,
- this is good practice,
- in fact, types are invaluable for specification, documentation, and help you to write the program systematically.

Example

Logical negation:

```
not True  = False
not False = True
```

Compiler infers:

- it's a function,
- it takes a truth value,
- and it yields a truth value.

Explicit type signature:

$$\text{not} :: \text{Bool} \to \text{Bool}$$

Type signatures

$\text{not} :: \text{Bool} \to \text{Bool}$

Read $::$ as "has type".

If a type signature is given, the compiler verifies that the function has the given type – much better than a comment.

Currying

Question

What is the type of "or"?

The operator takes two expressions of type Bool and produces a Bool again.

One option:

Two Booleans can form a pair.

A pair of Booleans is written $(\text{Bool}, \text{Bool})$ in Haskell.

Thus our candidate signature for "or":

$(\text{Bool}, \text{Bool}) \to \text{Bool}$

Currying

The option Haskell encourages and actually uses:

Bool → Bool → Bool

A function that returns a function.

Consider a vending machine with multiple products that can be selected by typing a number:

machine :: Money → Number → Product

If one person walks away after throwing in money, the next person can just enter a number to obtain a product.

Treating several-argument functions like this is called currying.

Partial application

The type signature for elem:

elem :: Int → [Int] → Bool

As with the vending machine, Haskell allows us to “walk away” after applying some arguments:

```
containsZero :: [Int] → Bool
containsZero = elem 0
```

Example

Consider elem once again:

$$\begin{aligned}
&\text{elem}\ x\ [\,] && = \text{False} \\
&\text{elem}\ x\ (y : ys) && = x == y \mid\mid \text{elem}\ x\ ys
\end{aligned}$$

Haskell infers a more general type than $\text{Int} \rightarrow [\text{Int}] \rightarrow \text{Bool}$.

Question

How could it be more general?

We don't actually assume anything in the code about numbers.
We only assume that we can compare elements for equality.

Type classes

A type class is a collection of types that support a common functionality.

Types supporting equality are in the type class Eq.

$$(==) :: \text{Eq}\ a \Rightarrow a \rightarrow a \rightarrow \text{Bool}$$

Read: If a supports equality, then $==$ takes two arguments of type a (the same type), and returns a Bool.

Similarly:

$$\text{elem} :: \text{Eq}\ a \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}$$

Functions with class constraints in their types are called overloaded.

Overloaded literals

Haskell uses a lot of overloading.

Even numeric literals are overloaded:

$23 :: \mathrm{Num}\ a \Rightarrow a$

This allows us to treat 23 as both an integer or a floating point number, depending on context.

(Parametric) Polymorphism

Question

What is the most general type of the empty list [] ?

Both [Int] and [Bool] would be too specific. Nothing is assumed about the elements yet . . .

We can use a type variable again – this time, without a class constraint:

$[\,] :: [a]$

Types with type variables are called polymorphic.

Polymorphism unrestricted by classes is also called parametric polymorphism.

Example

$$
\begin{aligned}
&(+\!\!+) :: [a] \to [a] \to [a] \\
&[\,] \quad\quad\;\; +\!\!+\; ys = ys \\
&(x : xs) +\!\!+\; ys = x : (xs +\!\!+\; ys)
\end{aligned}
$$

Operators are not built-in syntax, but can be defined as any other function.

Questions

- What does this function do?
- What does the type say?
- Are both arguments evaluated?

Data types

Some types (such as Int) are built-in.

Most types can be defined in a library. For example,

data Bool = False | True

Two labelled alternatives. The labels False and True are called (data) constructors.

False :: Bool
True :: Bool

Lists

We can define our own list type:

$$\textbf{data}\ \text{List a} = \text{Nil} \mid \text{Cons a (List a)}$$

Note:

- two constructors again, Nil and Cons,
- the a and List a are arguments of the Cons constructor,
- the datatype is parameterized and recursive.

Just as [] and (:) :

$$\begin{aligned}&\text{Nil} &&:: \text{List a}\\ &\text{Cons} &&:: \text{a} \to \text{List a} \to \text{List a}\end{aligned}$$

Recursion

Recursion is ubiquitous in Haskell:

- it is used in both datatypes and functions,
- often, the recursive structure of functions follows the recursive structure of datatypes,
- it is Haskell's way of writing "loops",
- it is not inefficient.

A possibility for abstraction

We often capture recurring patterns in their own functions.

Consider:

```haskell
elem :: Eq a => a -> [a] -> Bool
elem x []       = False
elem x (y : ys) = x == y || elem x ys
```

```haskell
(++) :: [a] -> [a] -> [a]
[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

Question

Can you see the similarities in the structure?

Generic list traversals

```haskell
elem :: Eq a => a -> [a] -> Bool
elem x []       = False
elem x (y : ys) = x == y || elem x ys
```

```haskell
(++) :: [a] -> [a] -> [a]
[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

Can be written as:

```haskell
elem x ys = foldr (\y r -> x == y || r) False ys
xs ++ ys  = foldr (\x r -> x : r)       ys xs
```

Anonymous functions

$$\begin{aligned} &\text{elem } x \; ys = \text{foldr } (\lambda y \; r \to x == y \; || \; r) \; \text{False } ys \\ &xs +\!\!+ \; ys \quad = \text{foldr } (\lambda x \; r \to x : r) \qquad ys \; xs \end{aligned}$$

The λ introduces an anonymous function.

A function that doubles its argument: $\lambda x \to x * 2$ or $\lambda x \to x + x$.

No side effects

Haskell functions do not have side effects.

When applied to the same arguments, Haskell functions always produce the same results.

Example

A typical impure function is a random number generator that takes a number n and produces a random number between 0 and n. Such a function cannot have type $\text{Int} \to \text{Int}$ in Haskell.

Example

A “function” that reads a line from the terminal and returns it as a String cannot have type String in Haskell.

Explicit effects

Fortunately,

- using side effects in Haskell is possible,
- but we have to be explicit about them in the types.

Most interactions with the world are marked with Haskell's built-in type former IO :

```
generateRandomNumber :: Int → IO Int
readString           :: IO String
```

Think of an expression of type IO a as a plan for interaction with the outside world – one that, when executed, yields an a .

Purity and effects

A function of type Int → Int always yields the same Int when passed the same number.

A function of type Int → IO Int does not. But it always yields the same plan!

The indirection of using IO allows us to talk about side-effecting programs without giving up our principles.

The main program

Every Haskell program has an entry point:

$$main :: IO\ ()$$

The whole program may have interactions with the outside world. The plan that is built for main is executed by the run-time system.

The type () is pronounced “unit”.

It has a single constructor, also ().

Used here to indicate that the final result of the main program is uninteresting.

Hello world!

To end this tour, we can now write “Hello world!”:

```
main = putStrLn "Hello world!"
```

where

```
putStrLn :: String → IO ()
```

prints a given string on the terminal.