A Quick Tour of Haskell

Haskell and Functional Programming

Andres Löh

2 Jul, 2015 —

Well-Typed
The Haskell Consultants

Introduction

Goals

What this is ...

- Overview of all important Haskell concepts.
- A lot of things in a relatively short amount of time.

Not:

- A detailed introduction.

Try things out!

You can:

- open `QuickTour.hs` in an editor,
- open GHCi and load that file,
- type in lots of stuff an see what happens.

High-level overview

- Defining functions
- Types
- Higher-order functions and IO

Our first goal

A programming problem

Given a sequence of numbers and a particular number, let's find out whether the number is contained in the sequence.

Our first goal

A programming problem

Given a sequence of numbers and a particular number, let's find out whether the number is contained in the sequence.

For example:

- 7 is not contained in the sequence 6, 9, 42.
- 9 is contained in the sequence 6, 9, 42.

Our first goal

A programming problem

Given a sequence of numbers and a particular number, let's find out whether the number is contained in the sequence.

For example:

- 7 is not contained in the sequence 6, 9, 42.
- 9 is contained in the sequence 6, 9, 42.

Requires us to talk about:

- numbers,
- sequences,
- "being contained in",
- ...

Expressions, constants, functions, bindings, values

Expressions

Expressions

Essentially: compound terms built up from constants and function calls.

An expression can be evaluated, yielding a value.

Expressions

Expressions

Essentially: compound terms built up from constants and function calls.

An expression can be evaluated, yielding a value.

Examples of constants:

```
2       -- a number
'x'     -- a character
[]      -- an empty "list"
True    -- a "Boolean" value
```

Well-Typed

Function calls

Examples of function calls:

```
not True    -- logical negation
min 7 2     -- minimum
2 + 3       -- addition
```

Function calls

Examples of function calls:

```
not True    -- logical negation
min 7 2     -- minimum
2 + 3       -- addition
```

```
7 `min` 2   -- same as above
(+) 2 3     -- same as above
```

Function calls

Examples of function calls:

```
not True    -- logical negation
min 7 2     -- minimum
2 + 3       -- addition
```

```
7 `min` 2   -- same as above
(+) 2 3     -- same as above
```

```
1 : [ ]     -- “cons” (prepend to list)
```

Function application syntax

“Space” is function application:

```
min 7 2    -- function applied to two arguments
```

Function application syntax

"Space" is function application:

```
min 7 2    -- function applied to two arguments
```

Parentheses are used for grouping:

```
〉 min 7 (2 + 6)
7
〉 min 7 2 + 6
8
```

Function application binds stronger than operators.

Operators

Operators are merely functions in “infix” syntax:

```
(+) 2 3      -- symbolic names can still be written prefix
7 `min` 2    -- alphanumeric names can still be written infix
```

There is no limited operator table – you can define your own (symbolic and alphanumeric) functions.

Bindings

Binding

Giving a name to an expression so that it can be reused:

```
five = 2 + 3
```

Pitfall: In GHCi, bindings have to be prefixed by **let** , so

```
〉 let five = 2 + 3
```

Then:

```
〉 five + five
10
```

Functions

Function

The essential unit of abstraction. A parameterized expression that can subsequently be applied to concrete arguments many times.

Functions

Function

The essential unit of abstraction. A parameterized expression that can subsequently be applied to concrete arguments many times.

$$
\begin{aligned}
&\mathsf{plusTwo}\ \mathsf{x} &&= \mathsf{x} + 2 \\
&\mathsf{plusTwo}' = \lambda \mathsf{x} \to \mathsf{x} + 2 && \text{-- “same” as above}
\end{aligned}
$$

Functions

Function

The essential unit of abstraction. A parameterized expression that can subsequently be applied to concrete arguments many times.

$$\text{plusTwo x} \quad = \text{x} + 2$$
$$\text{plusTwo}' = \lambda\text{x} \rightarrow \text{x} + 2 \quad \text{-- “same” as above}$$

Then:

$$\rangle\ \text{plusTwo } 3$$
$$5$$
$$\rangle\ (\lambda\text{x} \rightarrow \text{x} + 4)\ 1 \quad \text{-- anonymous function}$$
$$5$$

The Prelude

Very little is built into Haskell. E.g., all of

```
(+)
min
not
```

are library functions.

The Prelude

Very little is built into Haskell. E.g., all of

```
(+)
min
not
```

are library functions.

- Code is organized into modules.
- One special module Prelude is implicitly available in any other Haskell module.

First recap

- Expressions
- Values
- Constants
- Functions
- Bindings

Back to our goal

We want to check whether a number is contained in a sequence of numbers.

Back to our goal

We want to check whether a number is contained in a sequence of numbers.

- We want to define a function, let's call it elem.
- Two arguments: the number and the sequence.
- The result: “yes” or “no”.

Back to our goal

We want to check whether a number is contained in a sequence of numbers.

- We want to define a function, let's call it elem.
- Two arguments: the number and the sequence.
- The result: "yes" or "no".

How to talk about a "sequence"?

Lists

Lists

- One of many Haskell datatypes.
- Represents an ordered collection of elements such as numbers.
- A lot of built-in syntax, but otherwise not special.
- Good for learning: not trivial, but not too complicated either.

The structure of lists

```
[]             -- a list with no elements
[2]            -- a list containing one number
[6, 9, 42]     -- ... three numbers
[1, 3, 5, 7, 9] -- ... five numbers
```

The “cons” operator

Constructs a new list out of a single element and a list:

```
〉 6 : [9, 42]
[6, 9, 42]
```

```
〉 1 : [3, 5, 7, 9]
[1, 3, 5, 7, 9]
```

“Cons”-ing repeatedly

$\rangle$ 1 : (3 : (5 : (7 : (9 : [ ]))))
[1, 3, 5, 7, 9]

Even:

$\rangle$ 1 : 3 : 5 : 7 : 9 : [ ]
[1, 3, 5, 7, 9]

Observation

Every list can be built from [ ] by repeatedly applying (:) .

The structure of lists

In Haskell, a list is either:

- the empty list `[]`,
- or constructed as `x : xs`, by prepending (cons-ing) a single element `x` to a list `xs`.

Well-Typed

The structure of lists

In Haskell, a list is either:

- the empty list `[]`,
- or constructed as `x : xs`, by prepending (cons-ing) a single element `x` to a list `xs`.

```
1 : 2 : 3 : []   -- actual internal representation
[1, 2, 3]        -- “syntactic sugar”
```

Pattern matching

Back to defining elem

```
elem 5 [ ]
```

Back to defining elem

```
elem 5 [ ] = ...
```

Well-Typed

Back to defining elem

```
elem 5 [ ] = False
```

Back to defining `elem`

```
elem 5 [ ] = False
```

It does not matter that we're looking for `5` – nothing is ever contained in the empty list.

Back to defining elem

```
elem 5 [ ] = False
elem y [ ] = False
```

It does not matter that we're looking for 5 – nothing is ever contained in the empty list.

Back to defining elem – contd.

```
elem 5 [5, 7, 12] = ...
```

Well-Typed

Back to defining elem – contd.

```
elem 5 (5 : 7 : 12 : [ ]) =   . . .
```

Well-Typed

Back to defining elem – contd.

```
elem 5 (5 : 7 : 12 : [ ]) = True
```

Back to defining elem – contd.

```
elem 5 (5 : 7 : 12 : [ ]) = True
elem 5 (5 : xs          ) = True
```

It doesn't matter what comes after the 5 if the element we're looking for happens to be first.

Back to defining elem – contd.

```
elem 5 (5 : 7 : 12 : [ ]) = True
elem 5 (5 : xs          ) = True
elem 5 (6 : xs          ) =   ...
```

Back to defining elem – contd.

```
elem 5 (5 : 7 : 12 : [ ]) = True
elem 5 (5 : xs          ) = True
elem 5 (6 : xs          ) = elem 5 xs
```

Back to defining elem – contd.

```
elem 5 (5 : 7 : 12 : [ ]) = True
elem 5 (5 : xs          ) = True
elem 5 (6 : xs          ) = elem 5 xs
elem y (x : xs          ) = y == x || elem y xs
```

Back to defining elem – contd.

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

These two lines together are the definition of the elem function we're looking for.

On the left hand side we have patterns. If we call elem on actual arguments, we look for a matching equation and bind the parameters accordingly.

Evaluation

In GHCi

$\rangle$ elem 7 [6, 9, 42]
False
$\rangle$ elem 9 [6, 9, 42]
True

Let's look at what's happening in more detail ...

Example

```
elem 9 [6, 9, 42]
```

Let's remove syntactic sugar ...

Example

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

Example

```
elem 9 (6 : 9 : 42 : [ ])
elem y [ ]                  = False
```

Well-Typed

Example

```
elem 9 (6 : 9 : 42 : [ ])
elem y [ ]                = False
```

Does not match!

Well-Typed

Example

```
elem 9 (6 : 9 : 42 : [ ])
elem y (x : xs          ) = y == x  || elem y xs
```

Well-Typed

Example

```
elem 9 (6 : 9 : 42 : [])
elem y (x : xs          ) = y == x || elem y xs
```

Matches, with

```
y  = 9
x  = 6
xs = 9 : 42 : []
```

Example

```
elem 9 (6 : 9 : 42 : []) = 9 == 6 || elem 9 (9 : 42 : [])
elem y (x : xs         ) = y == x || elem y xs
```

Matches, with

```
y  = 9
x  = 6
xs = 9 : 42 : []
```

Well-Typed

Example

$$\text{elem } 9\ (6 : 9 : 42 : [\,]) = 9 == 6\ \ ||\ \text{elem } 9\ (9 : 42 : [\,])$$

What is $9 == 6$?

Example

$$\text{elem } 9\ (6 : 9 : 42 : [\,]) = \text{False} \mid\mid \text{elem } 9\ (9 : 42 : [\,])$$

Example

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

```
False || True  = ...
False || False = ...
```

Example

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

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

Well-Typed

Example

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

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

Simplify:

```
False || something = something
```

Here:

```
something = elem 9 (9 : 42 : [ ])
```

Well-Typed

Example

$$\mathrm{elem}\ 9\ (6 : 9 : 42 : [\,]) = \qquad \mathrm{elem}\ 9\ (9 : 42 : [\,])$$

Example

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

Again, a call to `elem` – but on a shorter list!

Example

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

Let's continue – does this match any of

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

Example

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

The second line matches

```
elem y (x : xs)          = y == x  || elem y xs
```

with

```
y  = 9
x  = 9
xs = 42 : [ ]
```

Example

```
elem 9 (6 : 9 : 42 : [ ]) =
elem 9 (9 : 42 : [ ])      = 9 == 9  || elem 9 (42 : [ ])
```

The second line matches

```

elem y (x : xs)            = y == x  || elem y xs
```

with

```
y  = 9
x  = 9
xs = 42 : [ ]
```

Example

```
elem 9 (6 : 9 : 42 : []) =
elem 9 (9 : 42 : [])      = 9 == 9 || elem 9 (42 : [])
```

What is `9 == 9`?

Example

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

Example

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

```
True || True  = ...
True || False = ...
```

Example

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

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

Example

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

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

Simplify:

```
True || something = True
```

Here:

```
something = elem 9 (42 : [ ])
```

Example

```
elem 9 (6 : 9 : 42 : [ ]) =
elem 9 (9 : 42 : [ ])     = True
```

Done!

Equational reasoning

Haskell's evaluation model

- Expressions are "reduced" to values.
- For function calls, find matching equations.
- Replace left hand sides by right hand sides.
- Stop once no more reduction is possible (a value is reached).

Once again, on a single page

Example:

$$\text{elem } 9\ [6, 9, 42]$$

Remember:

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

```
False || something = something
True || something  = True
```

Once again, on a single page

Example:

```
   elem 9 [6, 9, 42]
⇝ elem 9 (6 : (9 : (42 : [ ])))
```

Remember:

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

```
False || something = something
True || something  = True
```

Once again, on a single page

Example:

$$
\begin{aligned}
& \mathsf{elem}\ 9\ [6, 9, 42] \\
\rightsquigarrow\ & \mathsf{elem}\ 9\ (6 : (9 : (42 : [\,]))) \\
\rightsquigarrow\ & 6 == 9\ \ ||\ \mathsf{elem}\ 9\ (9 : 42 : [\,])
\end{aligned}
$$

Remember:

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

```
False || something = something
True || something  = True
```

Once again, on a single page

Example:

$$
\begin{array}{ll}
 & \mathsf{elem}\ 9\ [6,9,42] \\
\leadsto & \mathsf{elem}\ 9\ (6 : (9 : (42 : [\,]))) \\
\leadsto & 6 == 9\ \ ||\ \mathsf{elem}\ 9\ (9 : 42 : [\,]) \\
\leadsto & \mathsf{False}\ ||\ \mathsf{elem}\ 9\ (9 : 42 : [\,])
\end{array}
$$

Remember:

$$
\begin{array}{ll}
\mathsf{elem}\ x\ [\,] & = \mathsf{False} \\
\mathsf{elem}\ x\ (y : ys) & = x == y\ ||\ \mathsf{elem}\ x\ ys
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{False}\ ||\ something & = something \\
\mathsf{True}\ ||\ something & = \mathsf{True}
\end{array}
$$

Once again, on a single page

Example:

$$
\begin{aligned}
& \mathsf{elem}\ 9\ [6, 9, 42] \\
\rightsquigarrow\ & \mathsf{elem}\ 9\ (6 : (9 : (42 : [\,]))) \\
\rightsquigarrow\ & 6 == 9\ \ ||\ \mathsf{elem}\ 9\ (9 : 42 : [\,]) \\
\rightsquigarrow\ & \mathsf{False}\ ||\ \mathsf{elem}\ 9\ (9 : 42 : [\,]) \\
\rightsquigarrow\ & \mathsf{elem}\ 9\ (9 : 42 : [\,])
\end{aligned}
$$

Remember:

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

$$
\begin{aligned}
\mathsf{False}\ ||\ something &= something \\
\mathsf{True}\ ||\ something &= \mathsf{True}
\end{aligned}
$$

Once again, on a single page

Example:

$$
\begin{aligned}
& \mathsf{elem}\ 9\ [6, 9, 42] \\
\rightsquigarrow\ & \mathsf{elem}\ 9\ (6 : (9 : (42 : [\,]))) \\
\rightsquigarrow\ & 6 == 9\ \ ||\ \mathsf{elem}\ 9\ (9 : 42 : [\,]) \\
\rightsquigarrow\ & \mathsf{False}\ ||\ \mathsf{elem}\ 9\ (9 : 42 : [\,]) \\
\rightsquigarrow\ & \mathsf{elem}\ 9\ (9 : 42 : [\,]) \\
\rightsquigarrow\ & 9 == 9\ \ ||\ \mathsf{elem}\ 9\ (42 : [\,])
\end{aligned}
$$

Remember:

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

$$
\begin{aligned}
\mathsf{False}\ ||\ something &= something \\
\mathsf{True}\ ||\ something &= \mathsf{True}
\end{aligned}
$$

Once again, on a single page

Example:

$$
\begin{aligned}
& \text{elem } 9\ [6, 9, 42] \\
\rightsquigarrow\ & \text{elem } 9\ (6 : (9 : (42 : [\,]))) \\
\rightsquigarrow\ & 6 == 9\ \ ||\ \text{elem } 9\ (9 : 42 : [\,]) \\
\rightsquigarrow\ & \text{False}\ ||\ \text{elem } 9\ (9 : 42 : [\,]) \\
\rightsquigarrow\ & \text{elem } 9\ (9 : 42 : [\,]) \\
\rightsquigarrow\ & 9 == 9\ \ ||\ \text{elem } 9\ (42 : [\,]) \\
\rightsquigarrow\ & \text{True}\ \ \ ||\ \text{elem } 9\ (42 : [\,])
\end{aligned}
$$

Remember:

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

$$
\begin{aligned}
\text{False}\ ||\ \text{something} &= \text{something} \\
\text{True}\ ||\ \text{something} &= \text{True}
\end{aligned}
$$

Once again, on a single page

Example:

$$
\begin{array}{ll}
& \mathrm{elem}\ 9\ [6, 9, 42] \\
\leadsto & \mathrm{elem}\ 9\ (6 : (9 : (42 : [\,]))) \\
\leadsto & 6 == 9\ \ ||\ \mathrm{elem}\ 9\ (9 : 42 : [\,]) \\
\leadsto & \mathrm{False}\ ||\ \mathrm{elem}\ 9\ (9 : 42 : [\,]) \\
\leadsto & \mathrm{elem}\ 9\ (9 : 42 : [\,]) \\
\leadsto & 9 == 9\ \ ||\ \mathrm{elem}\ 9\ (42 : [\,]) \\
\leadsto & \mathrm{True}\ \ ||\ \mathrm{elem}\ 9\ (42 : [\,]) \\
\leadsto & \mathrm{True}
\end{array}
$$

Remember:

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

```
False || something = something
True  || something = True
```

The definition of “or”

While talking about `elem`, we have “discovered” the definition of `(||)`:

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

The definition of “or”

While talking about `elem`, we have “discovered” the definition of `(||)`:

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

Once again, definition by pattern matching:

- A list is either the empty list `[]`, or constructed by consing an element `x` to a list `xs` by writing `x : xs`. Two shapes or (data) constructors.
- A Boolean is either `False` or `True`. Again, two shapes or data constructors.

The definition of “or”

While talking about `elem`, we have “discovered” the definition of `(||)`:

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

Once again, definition by pattern matching:

- A list is either the empty list `[]`, or constructed by consing an element `x` to a list `xs` by writing `x : xs`. Two shapes or (data) constructors.
- A Boolean is either `False` or `True`. Again, two shapes or data constructors.

Constructors play an important role both in constructing data and in destructing data (via pattern matching).

Equational reasoning

The process of replacing equals by equals is called equational reasoning:

- A good mental model to reason about Haskell evaluation.
- Can be used to argue that certain expressions (say, different algorithms) are equivalent.
- Works locally, because the expression is the program. There is no implicit state.

Lazy evaluation

Lazy evaluation

Let's look at the definition of "or" again:

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

- We can make a decision without looking at the second argument (and indeed we did, while reducing `elem`).
- This definition of `(||)` has "shortcut behaviour".
- Unlike in many languages, this does not require a special hack, but follows from the definition and Haskell's evaluation strategy that essentially says "only evaluate things once they are needed".

Second recap

- Data is shaped by constructors.
- Functions are often defined by pattern matching on constructors.
- Evaluation is driven by pattern matching,
- and replacing (matching) left hand sides by right hand sides.

Types

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 – otherwise, we get a type error.

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.

A simple example

```
2 : [ ]
2 : 3
```

A simple example

```
2 : []
2 : 3
```

We know:

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

A simple example

```
2 : []
2 : 3
```

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.

A simple example

```
2 : []
2 : 3
```

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.

A more interesting example

Logical negation:

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

A more interesting 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.

A more interesting 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:

```
not :: Bool → Bool
```

Type signatures are checked!

Types are important

Type annotations in Haskell are optional, but

- it is good practice to provide type signatures;
- types are a design tool in Haskell.

Type inference in practice

Ask GHCi to infer types for you:

```
⟩ :t True
True :: Bool
⟩ :t not
not :: Bool → Bool
⟩ :t not True
not True :: Bool
```

Currying

Question

What is the type of “or”?

Currying

Question

What is the type of “or”?

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

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 $(\mathsf{Bool}, \mathsf{Bool})$ in Haskell.

Thus our candidate signature for “or”:

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

Currying

The option Haskell encourages and actually uses:

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

A function that returns a function.

Currying

The option Haskell encourages and actually uses:

$\text{Bool} \to \text{Bool} \to \text{Bool}$

A function that returns a function.

Currying

The option Haskell encourages and actually uses:

Bool $\rightarrow$ Bool $\rightarrow$ Bool

A function that returns a function.

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

machine :: Money $\rightarrow$ Number $\rightarrow$ Product

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

Currying

The option Haskell encourages and actually uses:

$\mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}$

A function that returns a function.

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

$\mathsf{machine} :: \mathsf{Money} \to \mathsf{Number} \to \mathsf{Product}$

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
```

Partial application

The type signature for elem:

$$\text{elem} :: \text{Int} \to [\text{Int}] \to \text{Bool}$$

(Partial) application:

$$\text{elem} :: \text{Int} \to [\text{Int}] \ \ \to \text{Bool}$$

Partial application

The type signature for `elem`:

```
elem :: Int → [Int] → Bool
```

(Partial) application:

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

Partial application

The type signature for `elem`:

```
elem :: Int → [Int] → Bool
```

(Partial) application:

```
elem :: Int → [Int]   → Bool
elem    0   :: [Int]  → Bool
elem    0      [1, 2] :: Bool
```

Partial application – contd.

As with the vending machine, we can “walk away” after applying some arguments:

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

Partial application – contd.

As with the vending machine, we can “walk away” after applying some arguments:

$$
\begin{aligned}
&\mathsf{containsZero} :: [\mathsf{Int}] \rightarrow \mathsf{Bool}\\
&\mathsf{containsZero} = \mathsf{elem}\ 0
\end{aligned}
$$

Then in GHCi:

$$
\begin{aligned}
&\rangle\ \mathsf{containsZero}\ [1, 2, 3]\\
&\mathsf{False}\\
&\rangle\ \mathsf{containsZero}\ [1, 0, 1, 0]\\
&\mathsf{True}
\end{aligned}
$$

Overloading and Polymorphism

Example

Consider `elem` once again:

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

Example

Consider `elem` once again:

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

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

Example

Consider elem once again:

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

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

Question

How could it be more general?

Example

Consider elem once again:

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

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`.

Type classes

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

Types supporting equality are in the type class `Eq`.

```
(==) :: Eq a ⇒ a → a → Bool
```

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

Type classes

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

Types supporting equality are in the type class Eq.

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

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

Similarly:

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

Functions with class constraints in their types are called overloaded.

Overloaded literals

Many Haskell functions are overloaded.

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 `[]` ?

(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 . . .

(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]
```

(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

What does this function do? And what is its type?

```
mystery []       = 0
mystery (x : xs) = 1 + mystery xs
```

Example

What does this function do? And what is its type?

```
length []       = 0
length (x : xs) = 1 + length xs
```

Well-Typed

Example

What does this function do? And what is its type?

```
length :: [a] → Int     -- or even:  Num b ⇒ [a] → b
length    []        = 0
length    (x : xs) = 1 + length xs
```

Data types

Data types

In Haskell, it is easy to define your own datatypes.

For example:

```
data Bool = False | True
data Dir = GoLeft | GoRight | GoUp | GoDown
```

Data types

In Haskell, it is easy to define your own datatypes.

For example:

```
data Bool = False | True
data Dir = GoLeft | GoRight | GoUp | GoDown
```

But also:

```
data [a] = [ ] | a : [a]
```

and many others . . .

Recursion and higher-order functions

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

```haskell
length :: [a] → Int
length []       = 0
length (x : xs) = 1 + length xs
```

A possibility for abstraction

We often capture recurring patterns in their own functions.

Consider:

```
elem :: Eq a ⇒ a → [a] → Bool
elem y []       = False
elem y (x : xs) = y == x || elem y xs
```

```
length :: [a] → Int
length []       = 0
length (x : xs) = 1 + length xs
```

Question

Can you see the similarities in the structure?

Generic list traversals

```
elem :: Eq a ⇒ a → [a] → Bool
elem y []       = False
elem y (x : xs) = y == x || elem y xs
```

```
length :: [a] → Int
length []       = 0
length (x : xs) = 1 + length xs
```

Generic list traversals

```
elem :: Eq a ⇒ a → [a] → Bool
elem y []       = False
elem y (x : xs) = y == x || elem y xs
```

```
length :: [a] → Int
length []       = 0
length (x : xs) = 1 + length xs
```

Can be written as:

```
elem y xs = foldr (λx r → y == x || r) False xs
length xs = foldr (λx r → 1 + r)       0     xs
```

Purity and effects

No side effects

Haskell functions do not have side effects.

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

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 $\mathsf{Int} \rightarrow \mathsf{Int}$ in Haskell.

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.

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
```

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 $\text{Int} \rightarrow \text{Int}$ always yields the same Int when passed the same number.

Purity and effects

A function of type $\mathrm{Int} \to \mathrm{Int}$ always yields the same Int when passed the same number.

A function of type $\mathrm{Int} \to \mathrm{IO\ Int}$ does not. But it always yields the same plan!

Purity and effects

A function of type $\mathsf{Int} \rightarrow \mathsf{Int}$ always yields the same Int when passed the same number.

A function of type $\mathsf{Int} \rightarrow \mathsf{IO\ Int}$ 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 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 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!”:

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

where

```haskell
putStrLn :: String → IO ()
```

prints a given string on the terminal.

Well-Typed