Type-directed Programming in Haskell

Fast Track to Haskell

Andres Löh, Edsko de Vries

8–9 April 2013 —

Skills Matter

Well-Typed
The Haskell Consultants

Goals

- Learn how to define functions
- How types help you while programming
- Syntax of Haskell
- How to define and use datatypes
- Overview of base types and datatypes

Structure of a Haskell program

- Haskell programs comprise one or more modules. One module per file. Main module is always called `Main`.
- Modules consist of declarations. Declarations introduce datatypes, functions and constants, type classes and instances.
- We will focus on functions and constants first, then datatypes. Later type classes and instances.

Declaring new functions and constants

$$\begin{aligned}
&\mathsf{length} :: [\mathsf{a}] \to \mathsf{Int} \\
&\mathsf{length}\ [\,] \qquad\quad = 0 \\
&\mathsf{length}\ (\mathsf{x} : \mathsf{xs}) = 1 + \mathsf{length}\ \mathsf{xs}
\end{aligned}$$

Declaring new functions and constants

$$
\begin{aligned}
&\boxed{\begin{array}{l}\mathsf{length}\\ \mathsf{length}\\ \mathsf{length}\end{array}}
\begin{array}{l} :: [a] \rightarrow \mathsf{Int} \\ [\,] \qquad\quad = 0 \\ (x : xs) = 1 + \mathsf{length}\ xs \end{array}
\end{aligned}
$$

- The name being introduced.

Declaring new functions and constants

$$\boxed{\text{length} :: [\text{a}] \rightarrow \text{Int}}$$
$$\text{length}\ [\,] \qquad\quad = 0$$
$$\text{length}\ (\text{x} : \text{xs}) = 1 + \text{length xs}$$

- The name being introduced.
- Type signature (optional, but recommended).

Declaring new functions and constants

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

- The name being introduced.
- Type signature (optional, but recommended).
- One or more equations defining the function.

Declaring new functions and constants

$$
\begin{array}{ll}
\text{length} :: [\text{a}] \rightarrow \text{Int} & \\
\text{length} \, [\,] & \boxed{=} \; 0 \\
\text{length} \, (\text{x} : \text{xs}) & \boxed{=} \; 1 + \text{length xs}
\end{array}
$$

- The name being introduced.
- Type signature (optional, but recommended).
- One or more equations defining the function.
- The $=$ symbol separates the left hand sides from the right hand sides.

Declaring new functions and constants

$$\begin{array}{l} \mathsf{length} :: [\mathsf{a}] \to \mathsf{Int} \\ \mathsf{length}\ \boxed{[\,]} = 0 \\ \mathsf{length}\ \boxed{(\mathsf{x} : \mathsf{xs})} = 1 + \mathsf{length}\ \mathsf{xs} \end{array}$$

- The name being introduced.
- Type signature (optional, but recommended).
- One or more equations defining the function.
- The $=$ symbol separates the left hand sides from the right hand sides.
- Cases are distinguished by patterns.

Declaring new functions and constants

$$
\begin{aligned}
&\text{length} :: [a] \to \text{Int} \\
&\text{length}\ [\,] \qquad\quad = \boxed{\begin{array}{l} 0 \\ 1 + \text{length xs} \end{array}} \\
&\text{length}\ (x : xs) =
\end{aligned}
$$

- The name being introduced.
- Type signature (optional, but recommended).
- One or more equations defining the function.
- The $=$ symbol separates the left hand sides from the right hand sides.
- Cases are distinguished by patterns.
- On the right hand side, we have expressions.

Declarations, patterns, expressions

Informally:

- A (function or constant) declaration binds a (new) identifier to an expression.
- A pattern occurs as an argument to an identifier on the left hand side of a declaration. It introduces names that are available on the right hand side. Patterns can be matched against actual function arguments. Matches can fail or succeed.
- Expressions occur on the right hand side of a function definition. Expressions can be evaluated.

Types

Every expression must have a type in Haskell – otherwise it will be rejected by the compiler:

- Haskell types can be inferred. There's usually no need for type annotations.
- Use `:t` in GHCi to obtain the inferred type of an expression.
- Type annotations (using `::`) are optional. But if they're given, their correctness is checked.

How to define a function?

There are two main design principles for defining functions:

- by (systematic) pattern matching and recursion,

How to define a function?

There are two main design principles for defining functions:

- by (systematic) pattern matching and recursion,
- by applying a higher-order function (such as composition, map, foldr, ...) and thereby reducing the problem to smaller subproblems.

In both cases, thinking about the types first helps you!

How to define a function?

There are two main design principles for defining functions:

- by (systematic) pattern matching and recursion,
- by applying a higher-order function (such as composition, map, foldr, ...) and thereby reducing the problem to smaller subproblems.

In both cases, thinking about the types first helps you!

We will focus on the pattern matching approach first.

Functions on lists

Most functions operate on structured data.

Lists are a simple data structure, so they're ideal for learning.

Functions on lists

Most functions operate on structured data.

Lists are a simple data structure, so they're ideal for learning.

Recall from the Quick Tour

Lists 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`).

Functions on lists

Most functions operate on structured data.

Lists are a simple data structure, so they're ideal for learning.

Recall from the Quick Tour

Lists 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`).

We call `[]` and `(:)` the constructors of the list datatype.

Another look at elem

Let's try to implement elem once more, systematically.

$$\mathsf{elem} :: \mathsf{Int} \rightarrow [\mathsf{Int}] \rightarrow \mathsf{Bool}$$

We start with the type.

Do we want to restrict ourselves to Int lists? No!

Another look at elem

Let's try to implement elem once more, systematically.

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

Let's make as few assumptions as possible.

In order to split up the programming problem, let's take a look at the input list . . .

Another look at elem

Let's try to implement elem once more, systematically.

$$\begin{array}{l} \mathsf{elem} :: a \to [a] \to \mathsf{Bool} \\ \mathsf{elem}\ x\ \boxed{\begin{array}{l}[\,] \\ (y : ys)\end{array}} \begin{array}{l} = \ldots \\ = \ldots \end{array} \end{array}$$

There are two cases, one per constructor of the list datatype.

Let's see if we can solve the simple case for [ ].

Another look at elem

Let's try to implement elem once more, systematically.

$$
\begin{aligned}
&\text{elem} :: a \to [a] \to \text{Bool}\\
&\text{elem}\ x\ [\,] \qquad\quad = \text{False}\\
&\text{elem}\ x\ (y : \boxed{ys}) = \ \ldots
\end{aligned}
$$

Now to the cons-case.

The ys is a shorter list – the most natural way to define functions on recursive datatypes is to use recursive functions.

Another look at elem

Let's try to implement elem once more, systematically.

$$\begin{aligned}
&\text{elem} :: a \to [a] \to \text{Bool} \\
&\text{elem}\ x\ [\,] \qquad\ = \text{False} \\
&\text{elem}\ x\ (y : ys) = \ \ldots\ \boxed{\text{elem}\ ys}\ \ldots
\end{aligned}$$

Let's try to complete the second case making use of the recursive call.

Another look at elem

Let's try to implement elem once more, systematically.

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

Done?

Another look at elem

Let's try to implement elem once more, systematically.

$$
\begin{aligned}
&\mathsf{elem} :: a \to [a] \to \mathsf{Bool}\\
&\mathsf{elem}\ x\ [\,] \qquad\quad = \mathsf{False}\\
&\mathsf{elem}\ x\ (y : ys) = x \boxed{==} y \mathbin{||} \mathsf{elem}\ x\ ys
\end{aligned}
$$

Oh, we actually need equality on the elements. That seems to be a sensible requirement for elem, so let's refine the type . . .

Another look at elem

Let's try to implement elem once more, systematically.

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

Now we're really done.

Another look at elem

Let's try to implement elem once more, systematically.

$$
\begin{aligned}
&\mathrm{elem} :: \mathrm{Eq}\ a \Rightarrow a \to [a] \to \mathrm{Bool} \\
&\mathrm{elem}\ x\ [\,] \qquad\quad = \mathrm{False} \\
&\mathrm{elem}\ x\ (y : ys) = x == y\ ||\ \mathrm{elem}\ x\ ys
\end{aligned}
$$

The systematic development we've just seen generalizes to most functions on lists and most functions on other structured datatypes.

Once again: length of a list

$\text{length} :: [a] \rightarrow \text{Int}$

Start with the type.

Once again: length of a list

$$\begin{aligned}
&\text{length} :: [a] \to \text{Int} \\
&\text{length}\ [\,] \qquad\quad = \ \ldots \\
&\text{length}\ (x : xs) = \ \ldots
\end{aligned}$$

Introduce cases based on the list structure.

Once again: length of a list

$$
\begin{aligned}
&\mathsf{length} :: [\mathsf{a}] \to \mathsf{Int}\\
&\mathsf{length}\ [\,] \qquad\quad = 0\\
&\mathsf{length}\ (\mathsf{x} : \mathsf{xs}) = \ \ldots
\end{aligned}
$$

Try to solve simple cases directly.

Once again: length of a list

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

Try to recurse where the datatype is recursive.

Once again: length of a list

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

Complete the cases.

Take a final look.

Everything looks fine.

Another example: take elements from a list

The call take n xs should return the first n elements from xs.

```
take :: Int → [a] → [a]
```

Skills Matter

Well-Typed

Another example: take elements from a list

The call take n xs should return the first n elements from xs.

```
take :: Int → [a] → [a]
take n []       = ...
take n (x : xs) = ...
```

What do we actually want to do if we want to take 3 elements of an empty list?

Another example: take elements from a list

The call take n xs should return the first n elements from xs.

```
take :: Int → [a] → [a]
take n []       = []
take n (x : xs) = ...
```

We take a simple approach.

Another example: take elements from a list

The call take n xs should return the first n elements from xs.

$$
\begin{aligned}
&\mathsf{take} :: \mathsf{Int} \to [\mathsf{a}] \to [\mathsf{a}] \\
&\mathsf{take}\ \mathsf{n}\ [\,] \qquad\quad = [\,] \\
&\mathsf{take}\ \mathsf{n}\ (\mathsf{x} : \mathsf{xs}) = \ \ldots\ \mathsf{take}\ \ldots\ \mathsf{xs}\ \ldots
\end{aligned}
$$

Wait, but what we want to do depends on n?

We can pattern match on an Int, too.

Another example: take elements from a list

The call take n xs should return the first n elements from xs.

```
take :: Int → [a] → [a]
take n []       = []
take 0 (x : xs) = ... take ... xs ...
take n (x : xs) = ... take ... xs ...
```

If cases overlap, the first matching case applies.

Another example: take elements from a list

The call take n xs should return the first n elements from xs.

$$
\begin{aligned}
&\mathsf{take} :: \mathsf{Int} \to [a] \to [a]\\
&\mathsf{take}\ n\ [\,] \qquad\quad = [\,]\\
&\mathsf{take}\ 0\ (x : xs) = [\,]\\
&\mathsf{take}\ n\ (x : xs) = \ \ldots\ \mathsf{take}\ \ldots\ xs\ \ldots
\end{aligned}
$$

Sometimes, we don’t need to recurse – even though we could.

Another example: take elements from a list

The call take n xs should return the first n elements from xs.

```
take :: Int → [a] → [a]
take n []       = []
take 0 (x : xs) = []
take n (x : xs) = x : take (n − 1) xs
```

Done.

Exercise – define the following functions

Append two lists:

$$(++) :: [a] \to [a] \to [a]$$

Hint: Only pattern match on the first list (i.e., don't distinguish more cases than needed).

Reverse a list:

$$\mathit{reverse} :: [a] \to [a]$$

Hint: Follow the standard pattern, and make use of $(++)$ that you have just defined.

Excursion: infix operators

Haskell allows you to create your own operators from a given set of symbols:

- names are either completely symbolic or completely alphanumeric;
- symbolic names are by default used infix, but can be used in prefix notation by surrounding them in parentheses (Example: `(+) 2 3` evaluates to 5);
- alphanumeric names are by default used prefix, but can be used in infix notation by surrounding them in backquotes (Example: ``8 `mod` 3`` evaluates to 2);
- you can define the associativity and priority of infix operators by using **infix**, **infixl**, and **infixr** declarations;
- by using `:i` or `:info` in GHCi, you can obtain information about the priority of infix operators.

Next: filtering a list

We want to traverse a list and keep all elements that have a certain property.

Question

How to best express a property of an element?

Next: filtering a list

We want to traverse a list and keep all elements that have a certain property.

Question

How to best express a property of an element?

As a function from the element to a Bool.

Next: filtering a list

We want to traverse a list and keep all elements that have a certain property.

Question

How to best express a property of an element?

As a function from the element to a Bool.

Recall from the Quick Tour: a Bool is a another datatype with two constructors, called True and False.

Defining filter

$\text{filter} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$

We can now write down the type.

Defining filter

$$
\begin{aligned}
&\text{filter} :: (a \to \text{Bool}) \to [a] \to [a] \\
&\text{filter } p\ [\,] &&= \ldots \\
&\text{filter } p\ (x : xs) &&= \ldots\ \text{filter}\ \ldots\ xs\ \ldots
\end{aligned}
$$

Defining filter

$$\begin{aligned}
&\mathrm{filter} :: (a \to \mathrm{Bool}) \to [a] \to [a]\\
&\mathrm{filter}\ p\ [\,] \qquad\quad = [\,]\\
&\mathrm{filter}\ p\ (x : xs) = \ldots\ \mathrm{filter}\ \ldots\ xs\ \ldots
\end{aligned}$$

It depends on the outcome of p x what we want to do!

We have several options here.

Defining filter

```
filter :: (a → Bool) → [a] → [a]
filter p []        = []
filter p (x : xs)  = if p x then x : filter p xs
                            else     filter p xs
```

We can use the built-in **if** - **then** - **else** construct.

Note that `Bool` is a type like any other. No need to write `p x == True`. Simply `p x` is simpler and equivalent.

Defining filter

```
filter :: (a → Bool) → [a] → [a]
filter p []          = []
filter p (x : xs)
    | p x            = x : filter p xs
    | otherwise      =     filter p xs
```

We can also use guards – each guard is tried in order.

Note that

```
otherwise :: Bool
otherwise = True
```

Using filter

There are some useful predicates:

$$\begin{aligned} &\mathsf{even}, \mathsf{odd} &&:: \mathsf{Integral}\ a \Rightarrow (a \to \mathsf{Bool}) \\ &\mathsf{isUpper}, \mathsf{isDigit} &&:: \mathsf{Char} \to \mathsf{Bool} \end{aligned}$$

We can also define our own:

```haskell
positiveInt :: Int -> Bool
positiveInt n = n > 0
```

```haskell
palindrome :: [Char] -> Bool
palindrome xs = reverse xs == xs
```

Note that `String` is a (type) synonym for `[Char]`.

Try using `filter` with these predicates.

Excursion: anonymous functions

In practice, functions such as filter are often used with lambda expressions or anonymous functions:

$\mathsf{filter}\ (\lambda \mathsf{n} \to \mathsf{n} > 10\ \&\&\ \mathsf{even}\ \mathsf{n})\ [1\,..\,100]$

Excursion: anonymous functions

In practice, functions such as filter are often used with lambda expressions or anonymous functions:

$$\text{filter}\ (\lambda \text{n} \rightarrow \text{n} > 10\ \&\&\ \text{even n})\ [1\mathinner{..}100]$$

A lambda expression is a way to define a function without giving it a name:

$$\text{myPredicate n} = \text{n} > 10\ \&\&\ \text{even n}$$

is just different syntax for

$$\text{myPredicate} = \lambda \text{n} \rightarrow \text{n} > 10\ \&\&\ \text{even n}$$

Excursion: operator sections

Partially applied infix operators have yet again special syntax:

λn $\rightarrow$ n $>$ 10

can be abbreviated to

($>$10)

Similarly, we can write (1 $+$) or ("Hello "$+\!\!+$) or (‘div‘ 5).

So it’s possible to say

filter ($>$10) [1 . . 100]

Exercise: defining map

The map function applies a given function to each element of a list:

```
map :: (a → b) → [a] → [b]
```

Tuples

Lists vs. tuples

It’s time to talk about a new (family of) datatypes: tuples.

- lists are a datatype that collects an arbitrary number of elements; all elements must be of the same type.
- tuples are a family of datatypes that collect a fixed number of elements; each element can have a different type.

Let’s look at examples.

Example tuples

Pairing a Bool and a String:

```haskell
example :: (Bool, String)
example = (True, "yes, it's true")
```

Example tuples

Pairing a Bool and a String:

```
example :: (Bool, String)
example = (True, "yes, it's true")
```

- The type states it is a pair, and also states the type of each component.

Example tuples

Pairing a Bool and a String:

```haskell
example :: (Bool, String)
example = (True, "yes, it's true")
```

- The type states it is a pair, and also states the type of each component.
- The corresponding expression has similar syntax, and constructs a pair out of two components.

Example tuples

Pairing a Bool and a String:

$$\text{example} :: (\text{Bool}, \text{String})$$
$$\text{example} = (\text{True}, \texttt{"yes, it's true"})$$

- The type states it is a pair, and also states the type of each component.
- The corresponding expression has similar syntax, and constructs a pair out of two components.

Here's a triple:

$$\text{triple} :: ([a] \to [a], [b] \to \text{Int}, \text{Char})$$
$$\text{triple} = (\text{reverse}, \text{length}, \texttt{'x'})$$

General structure of tuples

For each $n \geqslant 2$, there's a type of n-tuples.

- Each of these is a different type.
- Unlike lists (or `Bool`), each tuple type has a single constructor, conveniently written using parentheses and commas, with the arguments interspersed (it can also be used in prefix notation, actually).
- We can use pattern matching to extract the components of a tuple (but we do not need to distinguish several cases).

General structure of tuples

For each $n \geqslant 2$, there's a type of n-tuples.

- Each of these is a different type.
- Unlike lists (or Bool), each tuple type has a single constructor, conveniently written using parentheses and commas, with the arguments interspersed (it can also be used in prefix notation, actually).
- We can use pattern matching to extract the components of a tuple (but we do not need to distinguish several cases).

Remarks:

- There are no 1-tuples. Haskell treats (Int) and Int as the same type.
- The unit type () can be seen as a 0-tuple.

Example: selecting the first component of a pair

$\mathsf{fst} :: \ldots$

What is the type here?

$\mathsf{fst} :: (\mathsf{Int}, \mathsf{Char}) \to \mathsf{Int}$

is certainly too specific . . .

Example: selecting the first component of a pair

$\mathsf{fst} :: (a, b) \rightarrow a$

We can accept arbitrary component types. The two components can have different types. But the result type matches the type of the first component.

Now let's apply pattern matching on the input.

Example: selecting the first component of a pair

$$\mathsf{fst} :: (\mathsf{a}, \mathsf{b}) \to \mathsf{a}$$
$$\mathsf{fst}\ (\mathsf{x}, \mathsf{y}) = \ldots$$

Now x is the component of type a, and y the component of type b.

It's nearly trivial to finish the definition.

Example: selecting the first component of a pair

$$\mathsf{fst} :: (a, b) \rightarrow a$$
$$\mathsf{fst}\ (x, y) = x$$

And we are done.

Zipping two lists

Sometimes, we have two lists of equal length and want to combine them element by element:

$\mathsf{zip} :: [a] \to [b] \to [(a, b)]$

We start with the type.

It's a function over (two) lists, so let's apply the standard principle to the first list.

Zipping two lists

Sometimes, we have two lists of equal length and want to combine them element by element:

```
zip :: [a] → [b] → [(a, b)]
zip []       ys    = ...
zip (x : xs) ys    = ... zip xs ...
```

We actually need to look at the second list, too. So let's just match on that one as well.

Zipping two lists

Sometimes, we have two lists of equal length and want to combine them element by element:

$$
\begin{array}{llll}
\mathsf{zip} :: [a] \to [b] \to [(a, b)] & & & \\
\mathsf{zip}\ [\,] & [\,] & = & \ldots \\
\mathsf{zip}\ [\,] & (y : ys) & = & \ldots \\
\mathsf{zip}\ (x : xs) & [\,] & = & \ldots \\
\mathsf{zip}\ (x : xs) & (y : ys) & = & \ldots\ \mathsf{zip}\ xs\ ys\ \ldots
\end{array}
$$

The first case is easy: if both lists are empty, we return the empty list.

Zipping two lists

Sometimes, we have two lists of equal length and want to combine them element by element:

$$
\begin{array}{llll}
\mathsf{zip} :: [a] \to [b] \to [(a,b)] \\
\mathsf{zip}\ [\,] & [\,] & = & [\,] \\
\mathsf{zip}\ [\,] & (y : ys) & = & \ldots \\
\mathsf{zip}\ (x : xs) & [\,] & = & \ldots \\
\mathsf{zip}\ (x : xs) & (y : ys) & = & \ldots\ \mathsf{zip}\ xs\ ys\ \ldots
\end{array}
$$

In the final case, we can produce the first element of the resulting list and recurse.

Zipping two lists

Sometimes, we have two lists of equal length and want to combine them element by element:

$$
\begin{array}{llll}
\mathsf{zip} :: [a] \to [b] \to [(a,b)] \\
\mathsf{zip}\ [\,] & [\,] & = & [\,] \\
\mathsf{zip}\ [\,] & (y : ys) & = & \ldots \\
\mathsf{zip}\ (x : xs) & [\,] & = & \ldots \\
\mathsf{zip}\ (x : xs) & (y : ys) & = & (x, y) : \mathsf{zip}\ xs\ ys
\end{array}
$$

In the other two cases, there's a bit of flexibility:

- We could fail, yielding a partial function.
- But we can also just agree to return the shorter list.

Zipping two lists

Sometimes, we have two lists of equal length and want to combine them element by element:

$$
\begin{array}{llll}
\mathrm{zip} :: [a] \rightarrow [b] \rightarrow [(a, b)] \\
\mathrm{zip}\ [\,] & [\,] & = [\,] \\
\mathrm{zip}\ [\,] & (y : ys) & = [\,] \\
\mathrm{zip}\ (x : xs) & [\,] & = [\,] \\
\mathrm{zip}\ (x : xs) & (y : ys) & = (x, y) : \mathrm{zip}\ xs\ ys
\end{array}
$$

This definition has the advantage that we can use an infinite list as one argument:

$$\mathrm{zip}\ [1\,..]\ \mathrm{listOfNames}$$

Zipping two lists

Sometimes, we have two lists of equal length and want to combine them element by element:

$$
\begin{aligned}
&\mathsf{zip} :: [a] \to [b] \to [(a,b)] \\
&\mathsf{zip}\ (x : xs)\ (y : ys) = (x,y) : \mathsf{zip}\ xs\ ys \\
&\mathsf{zip}\ xs\ \qquad\quad ys \qquad\quad = [\,]
\end{aligned}
$$

We can actually collapse the first three cases into one, but now the order of patterns matters.

Simple variables match everything.

Zipping two lists

Sometimes, we have two lists of equal length and want to combine them element by element:

$$
\begin{aligned}
&\mathrm{zip} :: [a] \to [b] \to [(a, b)] \\
&\mathrm{zip}\ (x : xs)\ (y : ys) = (x, y) : \mathrm{zip}\ xs\ ys \\
&\mathrm{zip}\ \_\qquad\quad\ \_\qquad\quad\ \ = [\,]
\end{aligned}
$$

Pattern variables that are not used on the right hand side can be replaced by underscores.

Association lists

A list of pairs serves as a primitive way to associate keys with values.

```haskell
numbers :: [(Int, String)]
numbers = [(1, "one"), (5, "five"), (42, "forty-two")]
```

Association lists

A list of pairs serves as a primitive way to associate keys with values.

$$\text{numbers} :: [(\text{Int}, \text{String})]$$
$$\text{numbers} = [(1, \texttt{"one"}), (5, \texttt{"five"}), (42, \texttt{"forty-two"})]$$

Let's try to write a lookup function that obtains the value associated with a particular key …

Defining lookup – “bad” version

$\text{lookup} :: \text{key} \to [(\text{key}, \text{val})] \to \text{val}$

A first approximation of the type.

Let’s analyze the input list.

Defining lookup – “bad” version

$$
\begin{aligned}
&\mathsf{lookup} :: \mathsf{key} \to [(\mathsf{key}, \mathsf{val})] \to \mathsf{val} \\
&\mathsf{lookup}\ \mathsf{x}\ [\,] \qquad\qquad = \ldots \\
&\mathsf{lookup}\ \mathsf{x}\ (\mathsf{y} : \mathsf{ys}) \quad = \ldots\ \mathsf{lookup}\ \ldots\ \mathsf{ys}\ \ldots
\end{aligned}
$$

What val can we return if we reach the empty list and haven’t found our key?

Defining lookup – “bad” version

$$
\begin{aligned}
&\mathsf{lookup} :: \mathsf{key} \rightarrow [(\mathsf{key}, \mathsf{val})] \rightarrow \mathsf{val} \\
&\mathsf{lookup}\ \mathsf{x}\ [\,] \qquad\qquad = \mathsf{error}\ \texttt{"lookup: unknown key"} \\
&\mathsf{lookup}\ \mathsf{x}\ (\mathsf{y} : \mathsf{ys}) \qquad = \ldots\ \mathsf{lookup}\ \ldots\ \mathsf{ys}\ \ldots
\end{aligned}
$$

A bad solution is to trigger a run-time exception. We’ll improve on that shortly.

For the other case, we have to look at the first pair . . .

Defining lookup – “bad” version

$$
\begin{aligned}
&\mathsf{lookup} :: \mathsf{key} \to [(\mathsf{key},\mathsf{val})] \to \mathsf{val}\\
&\mathsf{lookup}\ x\ [\,] \qquad\qquad\quad = \mathsf{error}\ \texttt{"lookup: unknown key"}\\
&\mathsf{lookup}\ x\ ((k,v) : ys) = \ldots\ \mathsf{lookup}\ \ldots\ ys\ \ldots
\end{aligned}
$$

Now we have to compare x and k. Let’s use guards.

Defining lookup – “bad” version

```
lookup :: key → [(key, val)] → val
lookup x []            = error "lookup: unknown key"
lookup x ((k, v) : ys)
    | x == k           = ... lookup ... ys ...
    | otherwise        = ... lookup ... ys ...
```

If we found the key, we can immediately return the value. (So what will happen if the key occurs multiple times?)

Defining lookup – “bad” version

$$
\begin{aligned}
&\mathsf{lookup} :: \mathsf{key} \rightarrow [(\mathsf{key}, \mathsf{val})] \rightarrow \mathsf{val} \\
&\mathsf{lookup}\ \mathsf{x}\ [\,] && = \mathsf{error}\ \texttt{"lookup: unknown key"} \\
&\mathsf{lookup}\ \mathsf{x}\ ((\mathsf{k}, \mathsf{v}) : \mathsf{ys}) \\
&\quad \mid \mathsf{x} == \mathsf{k} && = \mathsf{v} \\
&\quad \mid \mathsf{otherwise} && = \ldots\ \mathsf{lookup}\ \ldots\ \mathsf{ys}\ \ldots
\end{aligned}
$$

In the remaining case, we simply recurse.

Defining lookup – “bad” version

```
lookup :: key → [(key, val)] → val
lookup x []            = error "lookup: unknown key"
lookup x ((k, v) : ys)
    | x == k           = v
    | otherwise        = lookup x ys
```

Let’s take a final look. Oh, we need equality on the key type . . .

Defining lookup – "bad" version

$$
\begin{array}{ll}
\mathsf{lookup} :: \mathsf{Eq\ key} \Rightarrow \mathsf{key} \to [(\mathsf{key}, \mathsf{val})] \to \mathsf{val} & \\
\mathsf{lookup\ x}\ [\,] & = \mathsf{error}\ \texttt{"lookup: unknown key"} \\
\mathsf{lookup\ x}\ ((\mathsf{k}, \mathsf{v}) : \mathsf{ys}) & \\
\quad \mid \mathsf{x} == \mathsf{k} & = \mathsf{v} \\
\quad \mid \mathsf{otherwise} & = \mathsf{lookup\ x\ ys}
\end{array}
$$

Now we're done, apart from the ugly call to error.

Excursion: about `error` and `undefined`

A call to `error` (as well as a function call for which no pattern match succeeds) causes a run-time exception.

Excursion: about error and undefined

A call to error (as well as a function call for which no pattern match succeeds) causes a run-time exception.

$$\begin{aligned}
&\text{error} &&:: \text{String} \to a \\
&\text{undefined} &&:: a
\end{aligned}$$

Note that these are polymorphic in the result type. This means they can be used in any context, because they abort normal control flow.

Excursion: total and partial functions

- A function that can trigger a run-time exception or that may loop is called a partial function.

Excursion: total and partial functions

- A function that can trigger a run-time exception or that may loop is called a partial function.
- Writing and using partial functions is discouraged – always try to cover all cases and make your functions total.

Excursion: total and partial functions

- A function that can trigger a run-time exception or that may loop is called a partial function.
- Writing and using partial functions is discouraged – always try to cover all cases and make your functions total.
- However, `undefined` and `error` can be useful tools while incrementally developing a program.

Maybe

How to fix lookup

We need a disciplined way to express failure without crashing.

How to fix `lookup`

We need a disciplined way to express failure without crashing.

Idea

Let's use a different result type for `lookup`, containing one additional value called `Nothing` to express failure.

The Maybe datatype

Given a type a, the type Maybe a contains all the values of type a plus one additional value:

- the term Nothing is a value of type Maybe a,
- if x is of type a, then Just x is of type Maybe a.

The Maybe datatype

Given a type a, the type Maybe a contains all the values of type a plus one additional value:

- the term Nothing is a value of type Maybe a,
- if x is of type a, then Just x is of type Maybe a.

There are two shapes / constructors of the Maybe datatype: Nothing and Just.

Defining lookup – "good" version

```
lookup :: Eq key => key -> [(key, val)] -> val
lookup x []             = error "lookup: unknown key"
lookup x ((k, v) : ys)
    | x == k            = v
    | otherwise         = lookup x ys
```

This is the version we had before.

Defining lookup – "good" version

$$
\begin{array}{lll}
\mathsf{lookup} :: \mathsf{Eq\ key} \Rightarrow \mathsf{key} \to [(\mathsf{key}, \mathsf{val})] \to \boxed{\mathsf{Maybe\ val}} \\
\mathsf{lookup\ x}\ [\,] & = \mathsf{error}\ \texttt{"lookup: unknown key"} \\
\mathsf{lookup\ x}\ ((\mathsf{k}, \mathsf{v}) : \mathsf{ys}) \\
\quad \mid \mathsf{x} == \mathsf{k} & = \mathsf{v} \\
\quad \mid \mathsf{otherwise} & = \mathsf{lookup\ x\ ys}
\end{array}
$$

We are now adapting the result type.

This requires changes in the rest of the function.

Defining lookup – “good” version

$$
\begin{array}{lll}
\mathsf{lookup} :: \mathsf{Eq}\ \mathsf{key} \Rightarrow \mathsf{key} \rightarrow [(\mathsf{key}, \mathsf{val})] \rightarrow \mathsf{Maybe}\ \mathsf{val} & & \\
\mathsf{lookup}\ x\ [\,] & = & \boxed{\mathsf{error}\ \texttt{"lookup: unknown key"}} \\
\mathsf{lookup}\ x\ ((k, v) : ys) & & \\
\quad \mid x == k & = & \boxed{v} \\
\quad \mid \mathsf{otherwise} & = & \mathsf{lookup}\ x\ ys
\end{array}
$$

The first of the right hand sides can be improved now. The other is no longer type correct.

Defining lookup – "good" version

```
lookup :: Eq key ⇒ key → [(key, val)] → Maybe val
lookup x []          = Nothing
lookup x ((k, v) : ys)
    | x == k         = v    -- still wrong
    | otherwise      = lookup x ys
```

Instead of using error, we can now return Nothing.

Defining lookup – "good" version

$$
\begin{aligned}
&\mathsf{lookup} :: \mathsf{Eq\ key} \Rightarrow \mathsf{key} \rightarrow [(\mathsf{key}, \mathsf{val})] \rightarrow \mathsf{Maybe\ val} \\
&\mathsf{lookup\ x}\ [\,] \qquad\qquad = \mathsf{Nothing} \\
&\mathsf{lookup\ x}\ ((\mathsf{k}, \mathsf{v}) : \mathsf{ys}) \\
&\quad \mid \mathsf{x} == \mathsf{k} \qquad\quad = \boxed{\mathsf{Just\ v}} \\
&\quad \mid \mathsf{otherwise} \qquad = \mathsf{lookup\ x\ ys}
\end{aligned}
$$

To inject v into the Maybe type, we use Just.

Defining lookup – “good” version

$$
\begin{aligned}
&\mathsf{lookup} :: \mathsf{Eq\ key} \Rightarrow \mathsf{key} \to [(\mathsf{key},\mathsf{val})] \to \mathsf{Maybe\ val}\\
&\mathsf{lookup\ x}\ [\,] \qquad\qquad\quad = \mathsf{Nothing}\\
&\mathsf{lookup\ x}\ ((\mathsf{k},\mathsf{v}) : \mathsf{ys})\\
&\quad \mid \mathsf{x} == \mathsf{k} \qquad\qquad\ = \mathsf{Just\ v}\\
&\quad \mid \mathsf{otherwise} \qquad\quad\ = \mathsf{lookup\ x\ ys}
\end{aligned}
$$

Done. This version of lookup is total.

It does not crash, but the type tells the user that Nothing may be returned, and forces the caller to deal with it.

Handling exceptions with Maybe

Given a default value, we can always recover a value from an optional value:

```
fromMaybe :: a → Maybe a → a
fromMaybe def x          = ...
```

We pattern match on the Maybe.

Two constructors, Nothing and Just.

Handling exceptions with Maybe

Given a default value, we can always recover a value from an optional value:

```
fromMaybe :: a → Maybe a → a
fromMaybe def Nothing  = ...
fromMaybe def (Just x) = ...
```

We use the default value in the Nothing case, and the wrapped value in the other.

Handling exceptions with Maybe

Given a default value, we can always recover a value from an optional value:

```
fromMaybe :: a → Maybe a → a
fromMaybe def Nothing = def
fromMaybe def (Just x) = x
```

Done.

Combining Maybe computations

We can provide a “backup” computation for a possibly failing computation.

$$
\begin{array}{l}
(<\!|\!>) :: \mathsf{Maybe}\ a \to \mathsf{Maybe}\ a \to \mathsf{Maybe}\ a \\
x \qquad\quad <\!|\!>\ y = \ \ldots
\end{array}
$$

We pattern match on the first input.

Combining Maybe computations

We can provide a “backup” computation for a possibly failing computation.

$$
\begin{aligned}
&(\mathord{<}|\mathord{>}) :: \mathsf{Maybe}\ a \rightarrow \mathsf{Maybe}\ a \rightarrow \mathsf{Maybe}\ a \\
&\mathsf{Nothing}\ \ \mathord{<}|\mathord{>}\ y = \ldots \\
&\mathsf{Just}\ x\ \ \mathord{<}|\mathord{>}\ y = \ldots
\end{aligned}
$$

We take the second computation if the first fails, otherwise ignore it.

Combining Maybe computations

We can provide a "backup" computation for a possibly failing computation.

```
(<|>) :: Maybe a → Maybe a → Maybe a
Nothing <|> y = y
Just x  <|> y = Just x
```

Done.

Note the similarity with (||) on Booleans (from the Quick Tour).

Why Maybe?

- By using Maybe in a result, we can express explicitly that the function can fail.
- The caller has to address the potential failure.
- By using Maybe in an argument, we can express that an argument is optional.
- The function writer has to say what to do if Nothing is passed.
- Only Maybe types have Nothing. This is different from null in other languages. There are no “null pointer exceptions” in Haskell.

Datatypes

Parameterized types, type constructors

- In Haskell, many types are parameterized by others.
- From existing types, we can make new types.
- Parameterized types are often called type constructors.

Parameterized types, type constructors

- In Haskell, many types are parameterized by others.
- From existing types, we can make new types.
- Parameterized types are often called type constructors.

Examples:

"List" is a type constructor.

Given any type a, there is also a type $[a]$.

"Pair" is a type constructor.

Given any types a and b, there is also a type (a, b).

"Function" is a type constructor.

Given any types a and b, there is also a type $a \to b$.

Building types with type constructors

There are no limits to composing type constructors:

$$([\mathsf{Int} \to \mathsf{Bool} \to \mathsf{Int}], [[\mathsf{Double}]] \to (\mathsf{Bool}, \mathsf{Int}))$$

Building types with type constructors

There are no limits to composing type constructors:

$$([\mathsf{Int} \to \mathsf{Bool} \to \mathsf{Int}], [[\mathsf{Double}]] \to (\mathsf{Bool}, \mathsf{Int}))$$

There are arbitrarily many type constructors, because we can define our own!

Defining data types

New datatypes with the data construct

```
data Weekday = Mo | Tu | We | Th | Fr | Sa | Su
```

This is an enumeration type. There are 7 constructors:

```
Mo, Tu, We, Th, Fr, Sa, Su :: Weekday
```

New datatypes with the **data** construct

```
data Weekday = Mo | Tu | We | Th | Fr | Sa | Su
```

This is an enumeration type. There are 7 constructors:

```
Mo, Tu, We, Th, Fr, Sa, Su :: Weekday
```

```
data Date = D Int Int Int    -- year, month, day
```

This is a record type. It has a single constructor:

```
D :: Int → Int → Int → Date
```

(Data) Constructors

$$
\begin{array}{ll}
\mathsf{Mo} & :: \mathsf{Weekday} \\
\mathsf{D} & :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \to \mathsf{Date} \\
\mathsf{False} & :: \mathsf{Bool} \\
(:) & :: a \to [a] \to [a] \\
(,) & :: a \to b \to (a, b) \\
\mathsf{Just} & :: a \to \mathsf{Maybe}\ a
\end{array}
$$

- Constructors are constants or functions that can be used to construct terms on the right hand side of a declaration.
- They have types targetting the datatype they belong to.
- Constructors determine the shape of values – they are not reduced, but evaluate to themselves.
- We can pattern-match on constructors (and not on ordinary constants or functions).

Datatypes yield programming patterns

From the datatype definition, we can read off the standard design principle for functions over the datatype:

- For each constructor, make a case.
- Use the arguments of the constructor on the right hand side.
- Whenever the datatype is recursive, consider making the function recursive.

Booleans

$$\textbf{data}\ \mathsf{Bool} = \mathsf{False} \mid \mathsf{True}$$

An enumeration type, like Weekday.

Two constructors, no recursion.

Booleans

```
data Bool = False | True
```

An enumeration type, like `Weekday`.

Two constructors, no recursion.

Example functions:

```
not :: Bool → Bool
not False = True
not True  = False

(&&) :: Bool → Bool → Bool
(&&) True True = True
(&&) _    _    = False
```

Tuples

$$\textbf{data}\ (a,b) \quad = (a,b)$$
$$\textbf{data}\ (a,b,c) = (a,b,c)$$

Parameterized. One constructor each. No recursion. Built-in syntax.

Tuples

$$\textbf{data}\ (a, b) \quad = (a, b)$$
$$\textbf{data}\ (a, b, c) = (a, b, c)$$

Parameterized. One constructor each. No recursion. Built-in syntax.

We could define our own, with less convenient syntax:

```
data Pair   a b   = MakePair a b
data Triple a b c = MakeTriple a b c
```

Well-Typed

Tuples

$$\mathbf{data}\ (a, b) \quad = (a, b)$$
$$\mathbf{data}\ (a, b, c) = (a, b, c)$$

Parameterized. One constructor each. No recursion. Built-in syntax.

We could define our own, with less convenient syntax:

$$\mathbf{data}\ \mathrm{Pair}\quad a\ b \quad = \mathrm{MakePair}\ a\ b$$
$$\mathbf{data}\ \mathrm{Triple}\ a\ b\ c = \mathrm{MakeTriple}\ a\ b\ c$$

Example functions:

$$\mathrm{secondOfThree} :: (a, b, c) \to b$$
$$\mathrm{secondOfThree}\ (x, y, z) = y$$
$$\mathrm{secondOfThree}' :: \mathrm{Triple}\ a\ b\ c \to b$$
$$\mathrm{secondOfThree}'\ (\mathrm{MakeTriple}\ x\ y\ z) = y$$

Maybe

```
data Maybe a = Nothing | Just a
```

Parameterized. Two constructors. No recursion.

Maybe

```
data Maybe a = Nothing | Just a
```

Parameterized. Two constructors. No recursion.

Example function:

```
fromMaybe :: a → Maybe a → a
fromMaybe def Nothing = def
fromMaybe def (Just x) = x
```

Lists

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

Parameterized. Two constructors. Recursive. Built-in syntax.

We could define our own, with less convenient syntax.

```
data List a = Nil | Cons a (List a)
```

We have seen lots of example functions following the standard design principle.

The **data** construct

The syntax of the **data** construct:

$$\textbf{data}\ \mathsf{Type}\ \mathsf{arg}_1\ \mathsf{arg}_2 \ldots \mathsf{arg}_m = \mathsf{Con}_1\ \mathsf{ty}_1\ \mathsf{ty}_2 \ldots \mathsf{ty}_n$$
$$\mid\ \mathsf{Con}_2 \ldots$$
$$\mid\ \ldots$$

Introduces the new datatype Type and the data constructors Con_1, Con_2, $\ldots$.

The **data** construct

The syntax of the **data** construct:

$$\textbf{data}\ \mathsf{Type}\ \mathsf{arg}_1\ \mathsf{arg}_2 \ldots \mathsf{arg}_m = \mathsf{Con}_1\ \mathsf{ty}_1\ \mathsf{ty}_2 \ldots \mathsf{ty}_n \\ \quad \mid \mathsf{Con}_2 \ldots \\ \quad \mid \ \ldots$$

Introduces the new datatype Type and the data constructors Con_1, Con_2, $\ldots$.

Types of constructors are determined by the data declaration:

$$\mathsf{Con}_1 :: \mathsf{ty}_1 \rightarrow \mathsf{ty}_2 \rightarrow \ldots \rightarrow \mathsf{ty}_n \rightarrow \mathsf{Type}\ \mathsf{arg}_1\ \mathsf{arg}_2 \ldots \mathsf{arg}_m$$

The data construct

The syntax of the **data** construct:

$$\textbf{data}\ \mathsf{Type}\ \mathsf{arg}_1\ \mathsf{arg}_2 \ldots \mathsf{arg}_m = \mathsf{Con}_1\ \mathsf{ty}_1\ \mathsf{ty}_2 \ldots \mathsf{ty}_n$$
$$\mid\ \mathsf{Con}_2 \ldots$$
$$\mid\ \ldots$$

Introduces the new datatype Type and the data constructors Con_1, Con_2, $\ldots$.

Types of constructors are determined by the data declaration:

$$\mathsf{Con}_1 :: \mathsf{ty}_1 \to \mathsf{ty}_2 \to \ldots \to \mathsf{ty}_n \to \mathsf{Type}\ \mathsf{arg}_1\ \mathsf{arg}_2 \ldots \mathsf{arg}_m$$

Type and constructor names must start with an uppercase letter; symbolic infix constructors must start with a colon (:).
Lists and tuples support additional built-in syntax that cannot be used for other datatypes.

Applying the design principle

Also for new datatypes, always keep in mind that by looking at the datatype, you obtain a design principle for functions over that type:

```
data Weekday = Mo | Tu | We | Th | Fr | Sa | Su
```

Applying the design principle

Also for new datatypes, always keep in mind that by looking at the datatype, you obtain a design principle for functions over that type:

```
data Weekday = Mo | Tu | We | Th | Fr | Sa | Su
```

```
isWeekend :: Weekday → Bool
isWeekend Mo = False
isWeekend Tu = False
isWeekend We = False
isWeekend Th = False
isWeekend Fr = False
isWeekend Sa = True
isWeekend Su = True
```

Applying the design principle

Also for new datatypes, always keep in mind that by looking at the datatype, you obtain a design principle for functions over that type:

```
data Weekday = Mo | Tu | We | Th | Fr | Sa | Su
```

```
isWeekend :: Weekday → Bool
isWeekend Sa  = True
isWeekend Su  = True
isWeekend _   = False
```

Collapsing cases – the order of cases then matters!

Another example

```
data Date = D Int Int Int    -- year, month, day
```

One constructor. No recursion.

Example function:

```
valid :: Date → Bool
valid (D y m d) = m ⩾ 1 && m ⩽ 12 && d ⩾ 1 && d ⩽ 31
```

Of course, this is not an optimal definition.

Type synonyms with **type**

Often, for datatypes with a single constructor, the constructor is named the same as the datatype itself:

```
data Date = Date Int Int Int
```

Type synonyms with **type**

Often, for datatypes with a single constructor, the constructor is named the same as the datatype itself:

```
data Date = Date Int Int Int
```

It's often better to give more meaningful names to types without creating a completely new type:

```
type Year  = Int
type Month = Int
type Day   = Int
data Date = Date Year Month Day
```

Type synonyms with **type**

Often, for datatypes with a single constructor, the constructor is named the same as the datatype itself:

```
data Date = Date Int Int Int
```

It's often better to give more meaningful names to types without creating a completely new type:

```
type Year  = Int
type Month = Int
type Day   = Int
data Date = Date Year Month Day
```

Note that **type** introduces type synonyms. For example, `2 :: Year` and `2 :: Int`. No conversion function is required.

Renamed types with newtype

We could also define:

```haskell
data Year = Year Int
```

Now Year and Int are different:

```haskell
Year :: Int → Year    -- the constructor
```

To extract the Int from a year, we can use pattern matching:

```haskell
fromYear :: Year → Int
fromYear (Year n) = n
```

Renamed types with newtype

We could also define:

```haskell
data Year = Year Int
```

Now Year and Int are different:

```haskell
Year :: Int → Year    -- the constructor
```

To extract the Int from a year, we can use pattern matching:

```haskell
fromYear :: Year → Int
fromYear (Year n) = n
```

For the case of a single-constructor, single-argument datatype (i.e., a renamed type), there's a more efficient construct:

```haskell
newtype Year = Year Int
```

Lessons

- Let the types guide you.
- Use pattern matching to get at components of values, and to distinguish cases.
- Try to follow the recursive structure of types (lists, trees).
- Cover all cases.
- Use precise types, such as `Maybe`, rather than causing uncontrolled errors.

Syntax

Haskell expression syntax

The most important syntactic aspects of syntax, you probably know from the examples. Let us nevertheless discuss a few constructs you will see, need or use sooner or later:

- pattern matching using **case**;
- local declarations using **let** and **where**;
- layout.

The **case** construct

Using **case**, you can pattern match on the result of an expression.

Normal function definitions, comprising several lines for several cases, can be rewritten to (possibly nested) **case** statements:

$$
\begin{array}{ll}
\text{length} :: [a] \to \text{Int} & \\
\text{length}\ [\,] & = 0 \\
\text{length}\ (x : xs) & = 1 + \text{length}\ xs
\end{array}
$$

This is the length function as we know it.

The case construct

Using **case**, you can pattern match on the result of an expression.

Normal function definitions, comprising several lines for several cases, can be rewritten to (possibly nested) **case** statements:

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

This is length using a **case** construct. Note the use of → rather than =.

Using **case** can be useful if you want to analyze intermediate results without declaring a helper function.

Efficiency of `reverse`

Let's look at the standard solutions for $(+\!\!+)$ and `reverse` earlier:

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

$$
\begin{array}{l}
\mathsf{reverse} :: [a] \to [a] \\
\mathsf{reverse}\ [\,] \qquad\quad = [\,] \\
\mathsf{reverse}\ (x : xs) = \mathsf{reverse}\ xs +\!\!+ [x]
\end{array}
$$

What is the efficiency of `reverse`?

Efficiency of reverse

Let's look at the standard solutions for $(+\!\!+)$ and reverse earlier:

$$
\begin{array}{l}
(+\!\!+) :: [a] \to [a] \to [a] \\
[\,] \qquad\quad +\!\!+\ ys = ys \\
(x : xs) +\!\!+\ ys = x : (xs +\!\!+\ ys)
\end{array}
$$

$$
\begin{array}{l}
\mathit{reverse} :: [a] \to [a] \\
\mathit{reverse}\ [\,] \qquad\quad = [\,] \\
\mathit{reverse}\ (x : xs) = \mathit{reverse}\ xs +\!\!+\ [x]
\end{array}
$$

What is the efficiency of reverse?

Answer: it has quadratic complexity, as $(+\!\!+)$ is linear in its left argument and reverse reduces to a linear chain of $(+\!\!+)$ invocations.

A better reverse

Let's build the reversed list as we go in an additional, accumulating argument:

$$\begin{aligned}
&\text{reverseAcc} :: [\text{a}] \to [\text{a}] \to [\text{a}] \\
&\text{reverseAcc acc } [\,] \qquad\quad = \text{acc} \\
&\text{reverseAcc acc } (\text{x} : \text{xs}) = \text{reverseAcc } (\text{x} : \text{acc}) \text{ xs}
\end{aligned}$$

$$\begin{aligned}
&\text{reverse} :: [\text{a}] \to [\text{a}] \\
&\text{reverse} = \text{reverseAcc } [\,]
\end{aligned}$$

A better reverse

Let's build the reversed list as we go in an additional, accumulating argument:

$$
\begin{aligned}
&\mathsf{reverseAcc} :: [a] \to [a] \to [a] \\
&\mathsf{reverseAcc}\ acc\ [\,] \qquad\quad = acc \\
&\mathsf{reverseAcc}\ acc\ (x : xs) = \mathsf{reverseAcc}\ (x : acc)\ xs
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{reverse} :: [a] \to [a] \\
&\mathsf{reverse} = \mathsf{reverseAcc}\ [\,]
\end{aligned}
$$

Note:

- the function reverseAcc uses the standard design principle for lists;
- we now traverse the list only once;
- the accumulating parameter techique is generally useful, and quite close to a stateful loop in an imperative language.

Local declarations using **where**

The function reverseAcc is only a helper for reverse, so we could define it locally:

```
reverse :: [a] → [a]
reverse = go []
  where
    go :: [a] → [a] → [a]
    go acc []       = acc
    go acc (x : xs) = go (x : acc) xs
```

Identifiers bound in a **where** are only visible in the case directly preceding the **where** clause.

On the other hand, functions in a **where** clause can use identifiers bound on the left hand side of the preceding clause.

Another example using **where**

$$
\begin{array}{l}
\mathsf{map} :: (a \to b) \to [a] \to [b] \\
\mathsf{map}\ f = go \\
\quad \textbf{where} \\
\qquad go\ [\,] \qquad\quad = [\,] \\
\qquad go\ (x : xs) = f\ x : go\ xs
\end{array}
$$

Another example using **where**

```
map :: (a → b) → [a] → [b]
map f = go
  where
    go []       = []
    go (x : xs) = f x : go xs
```

The f can be used in go. It never changes during the recursion.

Local declarations using let

Using **let** - **in**, we can also define local values.

Often, **let** is used to abbreviate repeatedly used subexpressions.

Example: testing if a list has at least 10 and at most 30 elements.

```
validList :: [a] → Bool
validList xs = let l = length xs
               in l ⩾ 10 && l ⩽ 30
```

Local declarations using **let**

Using **let** - **in**, we can also define local values.

Often, **let** is used to abbreviate repeatedly used subexpressions.

Example: testing if a list has at least 10 and at most 30 elements.

```
validList :: [a] → Bool
validList xs = let l = length xs
               in l ⩾ 10 && l ⩽ 30
```

Whether to use **let** or **where** is mainly a matter of taste, but **let** - **in** is an expression, whereas **where** is attached to a declaration.

Layout in Haskell

Layout of code plays an important semantic role in Haskell:

- never use TABs in Haskell code; that's just confusing;
- all cases of a function, all declarations in a module or **let** or **where**, all branches of a case must start on the same column;
- if you want to continue the expression from the previous line, you usually have to indent more than the column that would be used for the next declarations or branch.