Introduction to Haskell

Tom Schrijvers

Scientica

Copyright © 2022 Tom Schrijvers

Copying prohibited

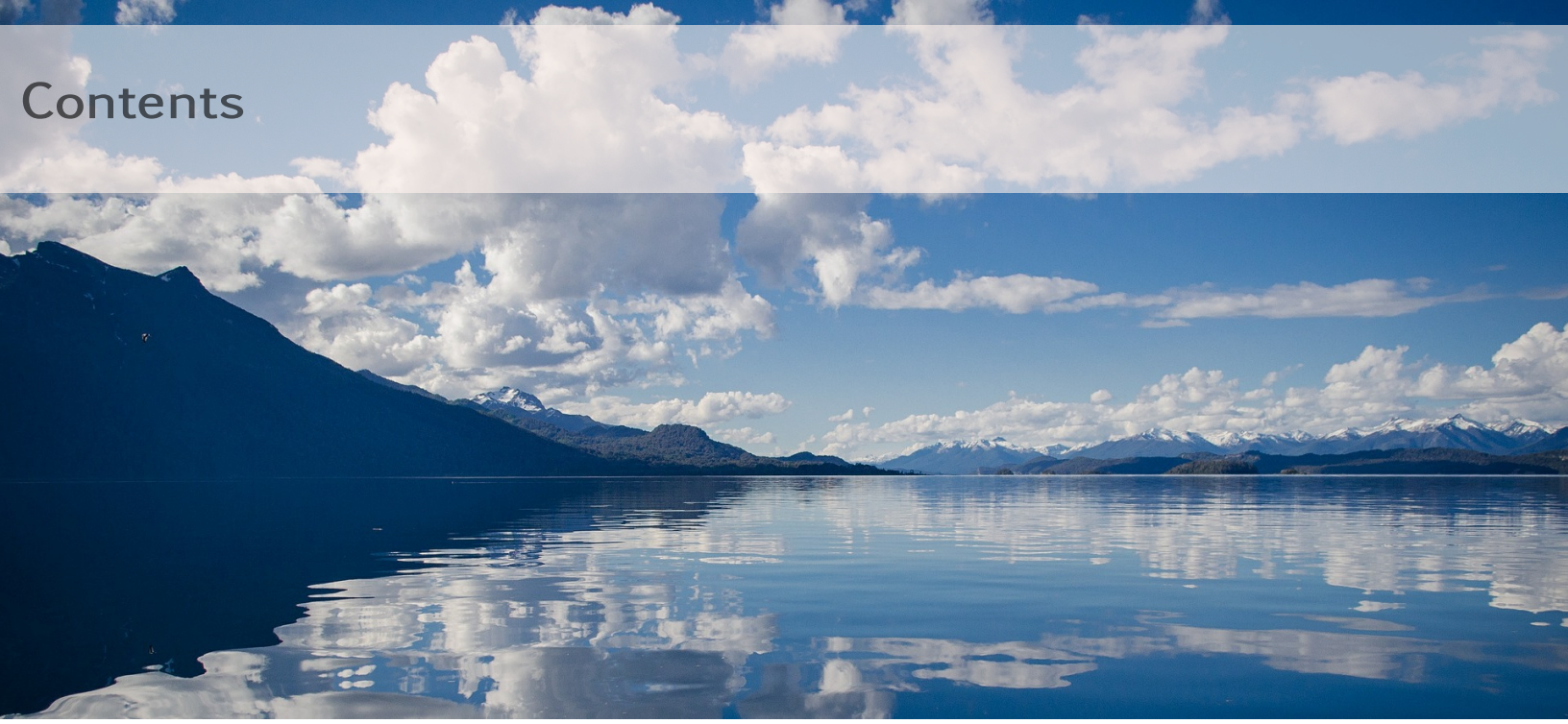
All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, or by any information storage or retrieval system, without the prior written permission of the publisher.

Art. No xxxxx
ISBN xxx–xx–xxxx–xx–x
Edition 2.0

Cover design by Tom Schrijvers

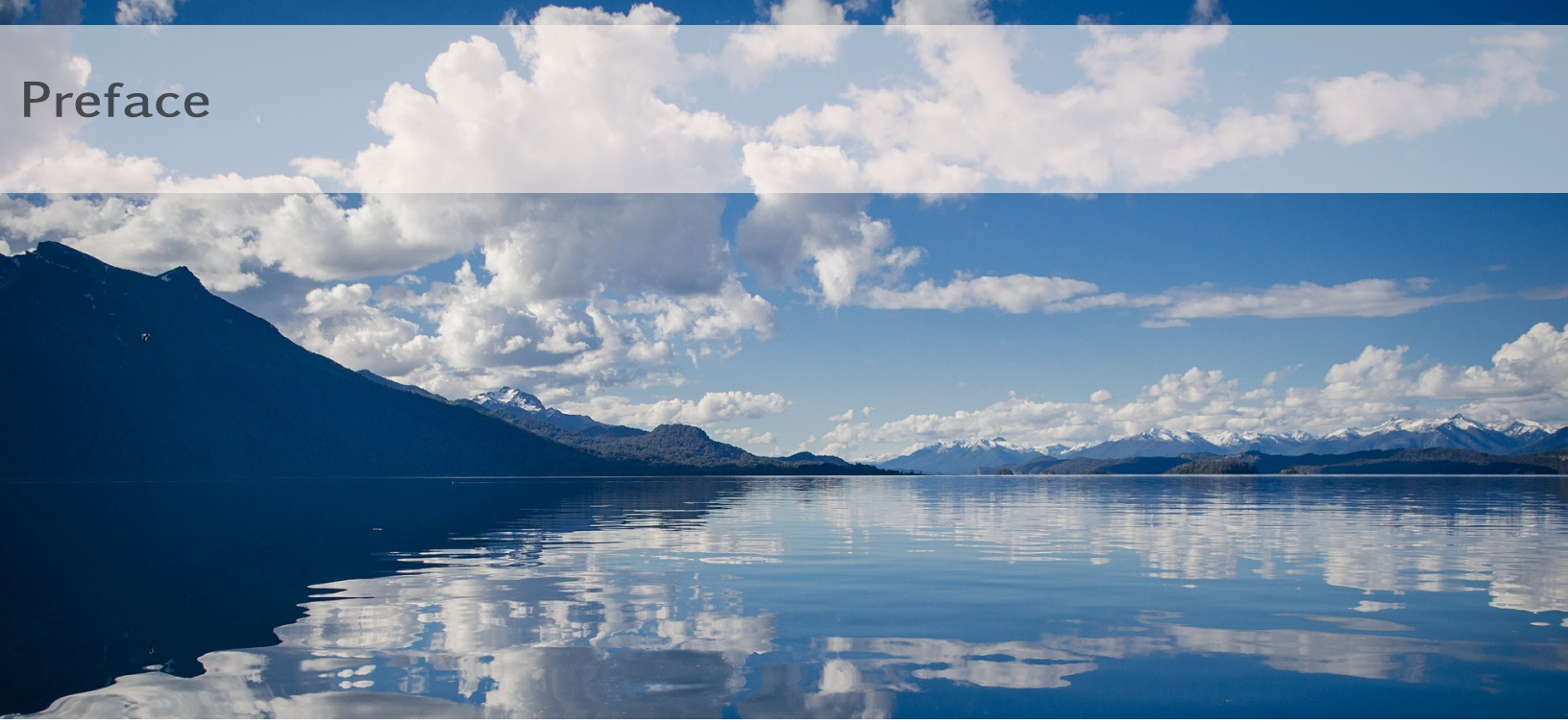
Published by Scientica
Printed in Leuven

Contents



Part II – Advanced Topics

Preface



Bug Bounty

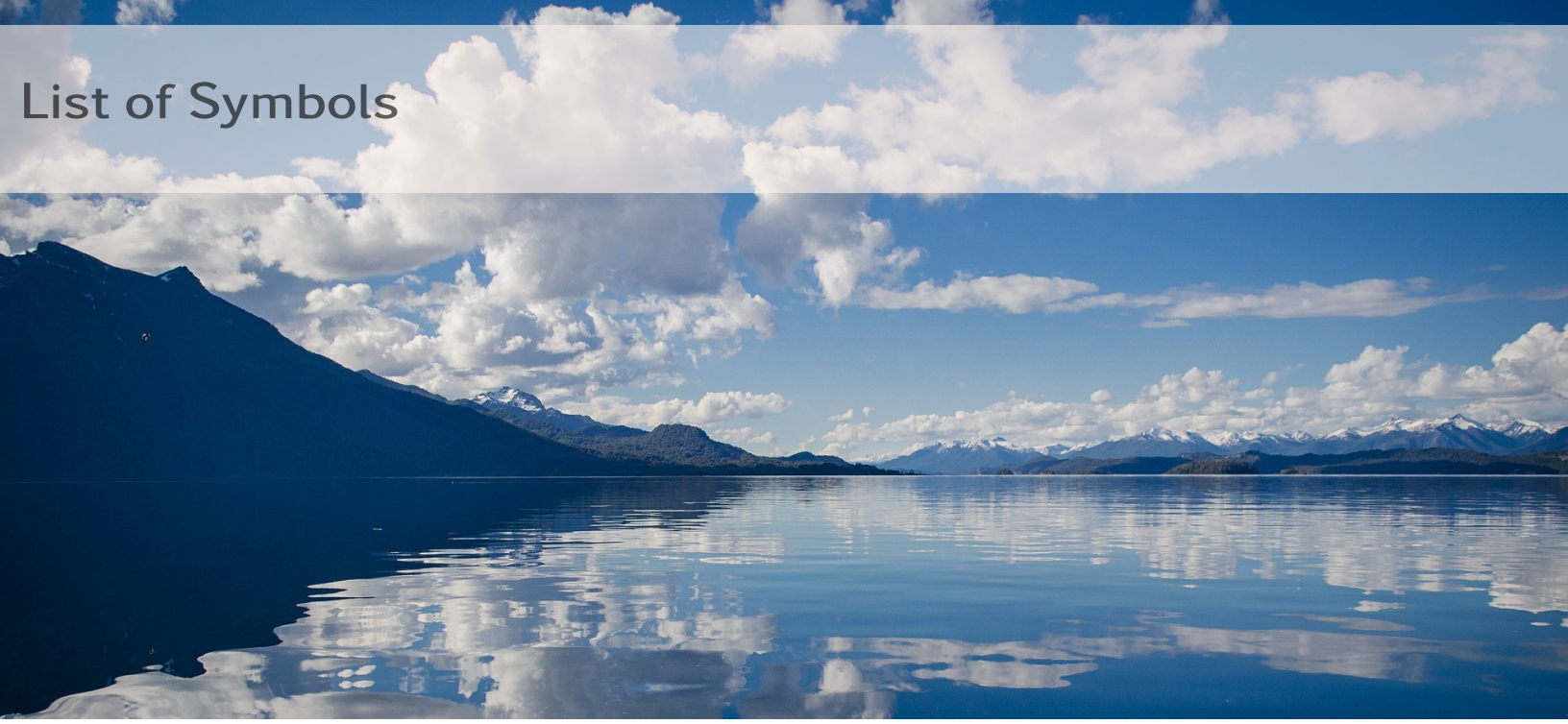
Please report any typos and other mistakes to the author at `tom.schrijvers@kuleuven.be`.
In exchange you will be forever acknowledged in the list below.

Acknowledgments

I am grateful to the following kind people for reporting annoying mistakes in this text.

Toon Willems
Pieter De Cremer
Tom Naessens
Yasser Deceukelier
Mathijs Rogiers
Nick Van Haver
Benoit Desouter
Bram De Deyn
Louis Van Ackere
Thijs Goemaere
Jorn Van denbussche
Matthias Van Eeghem
Jan Vermeulen
Jens Spitaels
Hannes Mareen
Jan De Schryver
Gregory Braekevelt
Jonas Devlieghere
Frederik De Smedt
Ali Bayraktar
Thor Galle
Nathan Van Laere
Wouter Baert
Koen Jacobs
Willem Seynaeve
Jonas Kapitzke
Li-Chiun Lin
Thomas Peeters
Sven Thijssen
Nathan Cornille
João Mesquita Pimentel
Jeroen Ooge
Gints Engelen
Wouter Mertens
Ruben Van Laer
Seppe Lesschaeve
Ben Fidlers
Ruben Kindt
Robin Spiltijns
Yinqi Liang
Sam Vervaeck
Valentin Ringlet
Max Yousif

List of Symbols



Borrowing notation and concepts from mathematics is an integral part of the Haskell and functional programming culture. To immerse ourselves in this culture, we are using the mathematical notation for Haskell source code that you will find in many papers and resources about Haskell.

This will be unusual at first, but you will get the hang of it quickly. Here is a conversion table to help you along.

Symbolic	ASCII	Name
$x \equiv y$	`x == y`	equal
$x \not\equiv y$	`x /= y`	not equal
$x \leqslant y$	`x <= y`	less than or equal
$x \geqslant y$	`x >= y`	greater than or equal
$a \to b$	`a -> b`	function type
$x \leftarrow xs$	`x <- xs`	generator in list comprehension
$\lambda x \to e$	`\x -> e`	anonymous function aka lambda abstraction
$l_1 +\!\!+ l_2$	`l1 ++ l2`	list append
$f \circ g$	`f . g`	function composition
$m \gg\!= f$	`m >>= f`	monadic bind

1. Introduction

This chapter discusses several practicalities related to the Haskell course and provides a background on Functional Programming in general and on Haskell specifically.

Before we dive into practical programming with the functional programming language Haskell, we briefly consider the mathematical foundation of (functional) programming languages. Even though a number of concepts may seem initially rather abstract, we will gradually discover that they provide the underlying philosophy around which Haskell is built. Without insight into this philosophy, we cannot understand the essence of functional programming.

We also briefly summarize the history and impact of the Haskell language, and mention a number of useful Haskell-related sources and tools.

1.1 Course Setup

The Haskell course which this book is a part of is organised around practicing—the only way to learn programming—uses a flipped classroom teaching style for that.

Before the lecture: You prepare for the lecture by reading—in advance—a chapter of this book and by making some basic preparatory exercises.

During the lecture: We focus on getting a deeper understanding of the material by addressing any questions and difficulties you have putting what you have read into practice by solving (basic and advanced) exercises together, discussing alternative solution strategies, ...

After the lecture: You get additional practice by making additional exercises more indenpently.

To help you along we have a whole support system in place:

The e-systant platform: We have created the `http://esystant.be` exercise platform for this course. The platform allows you to make Haskell exercises in the browser without having to install anything. E-systant contains all the course's exercise assignments. It gives automatic feedback on your solutions, allows you to improve them and also

provides model solutions. You can keep track of your progress relative to your peers on the leaderboard. You can use e-systant at the exam; it includes all the electronic documents that you can consult at that time.

Slack workspace: Since the corona crisis we are using a slack workspace both during and outside of the lectures. Students loved this and strongly recommended that we keep this for you.

We use the workspace during and outside the lectures (it's there 24/7) to exchange code snippets, ask questions, give clarifications, discuss, vote, post memes, ...

Coaches: Some courses have teaching assistants; we have coaches. They are there to help you become better Haskell programmers. Be sure to talk to them regularly and discuss code you have written. Whether you are stuck or think it is already perfect, our coaches will help you go further.

1.2 Beyond the Course

This course only covers the basics of Haskell and functional programming. For students who have a taste for more, I recommend reading the Advanced Topics part of this book and consider one or more follow-up activities:

- The Capita Selecta: Artificial Intelligence course consists of several tracks, one of which is "Language Engineering in Haskell".
- You can conduct further research into Haskell and other (functional) programming languages during a master thesis, and possibly during a subsequent PhD.
- Utrecht University organises a yearly summer school where you can expand your practical experience with Haskell.
- Haskell has a very active open source community that is welcoming to newcomers. The community participates every year in the Google Summer of Code. Just like former students of this book you can participate.
- There are interesting international internship opportunities. For instance, one former student of this book has done an internship at Tsuru Capital in Tokyo, a second at Google in Zurich and a third at CentralApp in Brussels.
- There are interesting career opportunities using Haskell. Former students of this course work at CentralApp, Fugue, PieSync, Standard Chartered, and Tweag.

Note that knowledge of Haskell is also an asset for companies that do not make use of functional programming, but are knowledgeable about the domain. It is often used as a filter to identify the better programmer; of course everyone knows Java, but Haskell is a different matter.

1.3 The λ-calculus

Functional programming goes back to the time immediately before the development of the first computer and the advent of programming languages. The first computer scientists, who

were mathematicians, were already preparing their arrival. They were studying at a fairly abstract level what could and could not be computed with a computer. For this purpose **Alonzo Church** developed in 1936 the first version of the λ-calculus.

This λ-calculus captures the essence of a programming language, but is on the one hand too *minimalistic* and on the other hand too *abstract* to be able to speak of a proper programming language.

The minimalism means that this calculus has just enough features to study computability without needlessly complicating the mathematical study of this phenomenon. As a consequence, the calculus can *in theory* express practical computations, but is *in practice* too impractical to do this effectively. Of course we cannot condemn the calculus for that, as it was never its intended use. Concretely the calculus consists of only 3 different expression forms and 3 different calculation rules.

1.3.1 Expressions

A program in the calculus is an *expression* E. Such an expression can be one of three different possible forms.

E	::=	x	variables
	|	$\lambda x.E$	(function) abstractions
	|	$E_1\, E_2$	(function) applications

The central form[1] is the function abstraction $\lambda x.E$, which is also called λ-abstraction, abstraction or simply function. This λ-abstraction represents a function in 1 parameter x and a function body E that may refer to the parameter x.

A parameter x presents a variable that can be used in the body of a function. An example of a very simple function is $\lambda x.x$, which is the function that just returns its parameter. This function is known as the *identity function*.

The third expression form is the function application, or application for short, $E_1\, E_2$. This form applies a function expression E_1 to another expression E_2. An example of such an application is the application of the identity function to the variable y: $(\lambda x.x)\, y$.

The λx part of the notation is called the λ-*binder*, because it binds the parameter x to a value during function application. Within an expression E we distinguish the *bound variables* from the *free variables*. A bound variable is one that occurs in E "under" a binder that binds it. A free variable is one that is not bound by a binder. For example, in $\lambda x.(\lambda y.(x\, y))\, z$ both x and y are bound variables, while z is free.

Sometimes there can be multiple binds for the same variables. For example, there are two binders for x in $\lambda x.(\lambda x.x)$. In this case, every occurrence of a variable is bound by its nearest (i.e., innermost) binder. We say that the inner binder shadows the outer binder. If there are multiple occurrences of a variable, each occurrence may be bound by a different binder. For example, in $\lambda x.((\lambda x.x)\, x)$ the x on the left is bound by the inner binder, while the x on the right is bound by the outer binder. The same variable can also occur both free and bound. For instance, in $(\lambda x.x)\, x$ the x on the left is bound, while the one on the right is free.

[1] It is this form that gives functional programming its name.

1.3.2 Calculation Rules

On their own the expression forms are not very useful: we can put together various expressions, but have not much to say about them at this point. The expressions are only notation (*syntax*) without meaning (*semantics*).

The calculation rules assign a kind of meaning to the expressions. There are three calculation rules for the λ-calculus, given in the form of equations:

$\lambda x.E$	$\equiv$	$\lambda y.[y/x]E$	α-conversion
$(\lambda x.E_1)\,E_2$	$\equiv$	$[E_2/x]E_1$	β-reduction
$\lambda x.(Ex)$	$\equiv$	E	η-conversion

α-conversion You have probably already developed the following intuition in the context of other programming languages you know: the names of parameters are irrelevant, you can freely change them without affecting the behavior of your program. This is also true in the λ-calculus. Whether you write the identity function as $\lambda x.x$ or as $\lambda y.y$ makes no difference: the two expressions are equivalent.

The α-conversion rule expresses this formally. The rule makes use of the generic notation $[y/x]E$ for renaming variables. This notation expresses that we apply the substitution to the expression E. This means that we replace in E every *free* occurrence of x with y. For example, $[y/x]x$ results in y, but $[y/x](\lambda x.x)$ remains $\lambda x.x$ because x does not occur free.

While reasoning about the equivalence of two expressions, the α-conversion is used to bridge the trivial differences in variable naming. A second important application of α-conversion is to ensure that every λ-binder in an expression E binds a variable with a different name. This avoids confusion and unintended shadowing.

β-reduction The rule for β-reduction is closely connected to the evaluation of programs in an actual programming language. It allows us to *evaluate* an expression by simplifying or *reducing* it. The rule replaces a function application $(\lambda x.E_1)\,E_2$ by the function body E_1 in which the parameter x has been replaced by the argument E_2. Just like the previous rule, this rule expresses the replacement in terms of substitution. This means that only the free occurrences of the parameter x are replaced.

η-conversion The η-conversion rule is an additional rule for comparing expressions without evaluating them. The rule expresses the notion of *extensionality*: the two expressions $\lambda x.(E\,x)$ (where x does not occur freely in E) and E behave in the same way modulo β-reduction. If we β-reduce $(((\lambda x.)E\,x)\,E')$, for any E', we get $E\,E'$. Hence, modulo β-reduction we cannot distinguish the two expressions. This fact is captured in the rule for η-conversion.

This rule is often applied to write expressions more compactly.

Equivalence The three calculation rules are expressed as axioms of the equivalence relation $\equiv$. Three additional, usually implicitly assumed, defining axioms of an equivalence relation are *reflexivity*, *symmetry* and *transitivity*. A final special axiom is *Leibniz's law*. This

law expresses that we may replace a subexpression E_1 of a larger expression E by an equivalent subexpression E_2 and still preserve equivalence of the whole.

E	$\equiv$	E		reflexivity
E_2	$\equiv$	E_1	if $E_1 \equiv E_2$	symmetry
E_1	$\equiv$	E_3	if $E_1 \equiv E_2$ en $E_2 \equiv E_3$	transitivity
$[E_1/x]E$	$\equiv$	$[E_2/x]E$	if $E_1 \equiv E_2$	Leibniz' law

Variable Capture A problematic situation may arise during β-reduction that inadvertently changes the meaning of an expression. Consider the expression $(\lambda y.(\lambda x.(x\,y)))\,x$. If we carelessly apply β-reduction to the outermost application, we obtain $\lambda x.(x\,x)$. Yet, this renders the second occurrence of x, which was originally free, into a bound variable. We say that the variable has been captured by the λ-binder. This does not happen when we start from an α-renamed variant of the original expression $(\lambda y.(\lambda z.(z\,y)))\,x$, which reduces to $\lambda z.(z\,x)$. We consider the first variable-capturing β-reduction to be invalid. In practice, we always make sure to first rename bound variables to avoid variable capture from happening.

1.4 Haskell

This book teaches functional programming in Haskell. Haskell is the mainstream functional programming language that most closely embodies the principles of functional programming. That is why we call it a *pure* functional language; it is not diluted by concessions to the imperative paradigm of computer hardware and mainstream programming languages. An additional feature of Haskell is its *laziness*, which we discuss at length in a later chapter.

1.4.1 History

The '80s were abuzz with research into lazy functional programming languages. Many researchers built their own experimental language implementation, of which the commercial Miranda system (1985) is by far the most well-known. At the end of the '80s many researchers decided that it made more sense to join forces than to continue dabbling with their individual systems. They founded the *Haskell committee* to design a common lazy functional programming language.

The Haskell committee released Haskell 1.0 in 1990. In subsequent years new versions of the language were released until finally a stable standard was published in 1998: Haskell'98. At that point the committee was effectively disbanded, but the development of different Haskell systems did of course continue. Recently the Haskell committee was re-instituted by the Haskell' (pronounced "Haskell *Prime*") movement to release updates to the Haskell standard on a yearly basis.

1.4.2 Prominent Language Properties

There are many interesting aspects to the Haskell language and its ecosystem.

What's in a name The name Haskell has been derived from the American logician *Haskell B. Curry* (1900-1982). He studied the essence of programming by means of minimal calculi known as *combinators*. We will encounter his name twice more, when we discuss the *Curry-Howard isomorphism* and the concept of *currying*.[1] Haskell Curry has also given his name to another pure functional language, Curry, which contains elements of logic programming.

[1] The name currying is actually inappropriate as the concept had been invented previously by Schönfinkel.

Types Haskell is known for its sophisticated type system. The language itself is *strongly and statically* typed, and really puts the types to work for the programmer instead of the opposite.

Strong typing means that every value in the program has a unique associated type and that only operations that conform to that type can be applied to it. There is no wiggle room.

Static typing means that the types and possible type errors are established during compilation, and thus before execution. In contrast to most statically typed languages, the programmer does not have to declare much type information. The Haskell compiler can mostly determine the types on its own using a process known as *type inference*.

Abstraction Haskell has a large capacity for abstraction. Many repetitive code patterns can be isolated in named entities and reused under that name.

The threshold for code reuse is a lot lower than with the typical unit of abstraction in object-oriented languages, the class. Hence, a Haskell program is often assembled from many small entities that are quickly composed into various new combinations. Large screen-filling code blocks and copy-paste programming are simply not done in Haskell.

As a consequence, Haskell does not need books about *design patterns*, but instead provides libraries with reusable abstractions. Haskell programs are thus small and make ample use of libraries.

Domain-Specific Languages The emphasis on small reusable abstractions, combined with advanced syntax capabilities, make Haskell a good stomping ground for *embedded domain-specific languages* (EDSLs). The core idea is to offer in Haskell a combination of abstractions that are useful for solving problems in a particular problem domain (e.g., drawing figures, managing financial contracts or controlling robots). By composing these abstractions in various ways new problems in the problem area can be solved quickly. Thanks to suitable syntax and cohesive abstractions it seems as if the programmer is no longer working in Haskell, but in a custom domain-specific programming language.

Learning Curve The Haskell learning curve looks quite different from that of Java. On the one hand it is much steeper. This means that the language requires more effort and insight of the programmer to reach a certain skill level. For instance, the programmer has to be able to handle more abstract concepts, and identify their potential application to concrete problems. On the other hand Haskell provides a much richer learning experience. Haskell facilitates continued learning about programming concepts and different ways to tackle programming problems. Even if you do not end up programming in Haskell in the long run, many of the ideas learnt in Haskell can be applied in the context of other languages. Moreover, once

mastered, Haskell is very powerful. For instance, Haskell programs are typically 10× shorter than the corresponding Java programs.

1.4.3 Industrial Impact

The general presence and visibility of Haskell in the software industry is rather limited. Nevertheless, the language has a substantial indirect influence on mainstream programming. Many ideas and concepts are first conceived, developed and fine-tuned in the context of Haskell before the most successful ones trickle down and are adopted by the mainstream. A good example is Scala, which was mainly influenced by Haskell, and which in turn influences Java and C#. In general it is fair to say that the development of mainstream programming languages is very sluggish, and that Haskell is one of the trailblazers far ahead of the pack.

Due to its steep learning curve Haskell is less suitable for (large) companies with a lower-educated programmer population. Instead, Haskell can often be found in small, highly productive programmer teams (inside or outside of large companies), such as at AT&T, Bluespec, Facebook, Galois, Linspire, Qualcomm and Google. In addition, Haskell also thrives in application niches with highly educated programmers. The most prevalent such area is that of *investment banks*: ABN AMRO, Barclays Capital, Credit Suisse, Deutsche Bank, Standard Chartered and Tsuru Capital.

Part I

Basic Topics

2. First Steps in Haskell

This chapter covers the basic features of Haskell and enables you to write simple programs.

2.1 Functions

The term *functional programming* says it all: functions are the core unit of code in Haskell.

2.1.1 My First Function

Let us look right away at a first example of a Haskell function definition:

```
f x = x + 1
```

This notation is very similar to the mathematical notation for the same function definition:

$$f(x) = x + 1$$

In both cases we speak of a function (named) f with a parameter x. The only difference between the Haskell notation and the mathematical notation is that Haskell does not require parentheses around the parameter x.

Let us also briefly emphasize the role of the = symbol. Just like in mathematics this symbol expresses that f x is equivalent to x + 1. Importantly, this equality goes both ways. Of course the evaluation uses the equality from left to right. However, the equation also works the other way around: we can replace x + 1 anywhere in a program by f x without changing the program's meaning.

Now we illustrate how to call this function in GHCi, the interactive environment of GHC.

```
▶ f 0
1
```

Just like in the function definition we separate the function name from the (now actual) parameter with a space. GHCi prints the result of the function call, in this case 1, on the next line. Based on the function definition it should be intuitively clear how GHCi obtains this result. We return later to the technical details of this process.

Here is another call:

```
▶ f 1
2
```

We can obtain the same result by applying f twice in succession to the value 0. This gives us a slightly larger expression:

```
▶ f (f 0)
2
```

As you can see, we do use parentheses here. Yet, they are not used to separate the function name from its parameters. Instead they are used to group a subexpression. If we had written f f 0, then Haskell would not have understood that 0 is a parameter of the second f and f 0 a parameter of the first f. The parentheses group the unit f 0 to disambiguate the expression.

Parentheses in Haskell have exactly the same grouping role as parentheses have in arithmetic expressions like $(1 + 2) \times 3$.

2.1.2 Multiple Parameters

A function with multiple parameters is defined in much the same way as a function with a single parameter:

```
g x y = x + 2 * y
```

We separate the parameters from one another with a space. The same happens in function calls:

```
▶ g 1 2
5
▶ f (g 1 2)
6
▶ g 1 (f 2)
7
```

2.1.3 Naming

There are two kinds of function names in Haskell: 1) conventional alphanumeric names, and 2) operator names.

Conventional Alphanumeric Names Conventional alphanumeric names start with a lower case letter (a-z) or an underscore (_). In the latter case more characters have to follow; in the former case this is optional. Subsequent characters may be lower case letters (a-z), upper case letters (A-Z), digits (0-9), underscore (_) and apostrophe (').

These are examples of valid names:

```
f
f'
f''
_f
f'_'f
fUNny
```

It is good style to use camel case for function names and to limit the use of digits, underscores and apostrophes.

```
concatMap
zipWith
```

Digits and apostrophes are sometimes used as suffixes of existing function names to identify related functions. However, this practice is only used for illustration purposes and is not suitable for production code.

Operator Names In contrast to many other programming languages, Haskell allows the programmer to define custom operators. An operator name consists of one or more of the following symbols:

```
!#$%&*+./<=>?@\^|-~:
```

Many operators already have a predefined meaning, like:

```
$  +  ++  >>=
.  -  &&  <*>
<  *  ||  <$>
>  /  >>
```

but there are still many unused combinations.

Function Application Syntax The notation for the application of a function to a number of parameters depends on the type of function name. In case of a conventional function name, the function is placed in front of its parameters. This is known as *prefix* notation.

```
g x y = f x y
```

If the function name is an operator, the operator is placed between the two parameters. This is known as *infix* notation.

```
x @@ y = x + y
```

Conversion It is possible to use a conventional function name as if it were an operator and vice versa. We achieve the former by placing the function name between *backquotes* (a.k.a. *backticks*). This allows us to use conventional functions in infix position.

```
x `g` y = x `f` y
```

Conversely parentheses turn an operator into a conventional funtion name that can be used in prefix position.

```
(@@) x y = (+) x y
```

2.2 Primitive Data

The λ-calculus establishes that, in theory, functions are sufficient to express all computational tasks. That is of course not very practical. Hence, Haskell offers also various other datatypes besides functions.

2.2.1 `Int`: Integer Numbers

The type `Int` is the name for integer numbers in Haskell. We have in fact already used integer numbers in the above functions `f` and `g`. This fact has perhaps gone unnoticed because the notation for `Int` values is entirely standard: `0`, `1`, `-22`, ...

For addition, subtraction and multiplication Haskell provides the usual operators that can be used in infix position:

```
▶ 5 + 3
8
▶ 10 - 3
7
▶ 7*9
63
```

Less conventional is the use of a regular function `div` for integer division:

```
► div 8 2
4
► div 8 3
2
```

There are various other predefined Int functions. Here is one more:

```
► mod 8 3
2
► (div 8 3) * 3 + (mod 8 3)
8
```

2.2.2 Double: Floating-Point Numbers

Floating-point numbers are also written in the usual way: 0.0, 3.14159, – 99.99, ... By default Haskell uses *double* precision. Hence the type name Double. We will see later how to obtain single precision numbers of type Float.

Again Haskell provides operators +, –, * and now also / for division.

```
► 8.0 / 2.0
4.0
► 8.0 / 3.0
2.6666666666666665
```

2.2.3 Bool: Boolean Values

Finally, we mention the Boolean values of type Bool: True and False.

The operators for conjunction and disjunction are the conventional ones:

```
► True && False
False
► True || False
True
```

The function not implements negation:

```
► not True
False
► not False
True
```

You can obtain Boolean values by comparing numbers:

```
► 1 + 1 ≡ 2
True
► 1 ≢ 1
False
► 2 > 3
False
► 3.1 < 4.7
True
```

Of course you can also define your own functions that return Boolean values:

```
passingGrade p = p > 9
```

and use them at will:

```
► passingGrade 9
False
► passingGrade 10
True
```

2.3 Conditional Expressions

Conditional control structures are found in nearly all programming languages.

2.3.1 If-then-else

The *Flafius* bank offers an interest rate that depends on the savings amount. Their policy is captured in the following function:

```
rate a =
  if a > 250000
    then 1.00
    else 1.40
```

If the savings amount is more than 250,000 euro, you get a lower interest rate. The function rate makes use of the built-in conditional expression **if – then – else**.

You can nest multiple occurrences of this conditional expression form to distinguish more than two cases. Consider for instance the Flafius *Plus* account that distinguishes three different cases:

```
plusRate a =
  if a > 250000
    then 1.00
    else if a > 50000
      then 1.40
      else 1.90
```

2.3.2 Guards

The if – then – else notation quickly becomes unwieldy when there are more than two cases. For this reason Haskell provides an interesting alternative known as *guards*. With guards we can write the function plusRate as follows:

```
plusRate a
  | a > 250000 = 1.00
  | a > 50000  = 1.40
  | otherwise  = 1.90
```

This formulation is more compact and readable. A guard is a Boolean expression like a > 250000 that "guards" a case of the function. If the test succeeds, the corresponding result is returned. If the test fails, the function tries the next guard. The final case is usually a default case guarded by otherwise. Note that otherwise is just a more readable synonym for True.

The guard notation has been inspired by the corresponding mathematical notation:

$$plusRate(a) = \begin{cases} 1.00 & ,250000 < a \\ 1.40 & ,50000 < a < 250000 \\ 1.90 & ,otherwise \end{cases}$$

2.4 Lists and Recursion

A list is an ordered collection of data that is used in a wide range of applications. Lists in Haskell deserve special attention because they are used more frequently, also in situations where other languages would not use them.

2.4.1 Lists

A list containing the elements 1, 2 and 3 is denoted as [1,2,3]. Lists with other types of elements are written in a similar fashion. For instance, [True,False,False,True] is a list of Boolean values and [0.0,0.25,0.5,0.75,1.0] contains doubles.

Three useful functions to inspect lists are null, head and tail. The first of these checks whether a list is empty.

```
► null [1,2,3]
False
► null [ ]
True
```

The functions head and tail are only defined for non-empty lists. In such cases head returns the first element and tail the list of remaining elements.

```
► head [1,2,3]
1
► tail [1,2,3]
[2,3]
```

Applying either function on an empty list results in a runtime exception.

```
► head [ ]
***Exception: Prelude.head: empty list
► tail [ ]
***Exception: Prelude.tail: empty list
```

2.4.2 Primitive Notation

The notation [1,2,3] is actually *syntactic sugar* meant to "sweeten" the use of lists. This syntactic sugar is internally translated by Haskell into a more general and primitive notation. This general notation is based on an *inductive* definition of lists.

- The base case is the empty list [ ], and
- the inductive case x:xs sticks the element x in front of the list xs.

We call [ ] and : the *constructors* of lists; they allow us to build lists.

```
► 1:[ ]
[1]
► 1:(2:[ ])
[1,2]
► 1:2:[ ]
[1,2]
```

As you can see GHCi prints lists we build with the primitive constructors using the syntactic sugar notation. Observe that we can omit parentheses because the right-most occurrence of : has the highest priority.

2.4.3 Recursive Functions on Lists

Now that we have seen how to build and inspect lists, nothing can stop us from writing various functions on lists. For instance, we can write a simple function that creates a singleton list from the given element thus:

```
singleton x = [x]
```

However, functions like singleton that do all of their work "at once" in the function body are, due to their simplicity, not terribly exciting.

Functions become more fanciful and interesting when they outsource part of their work to another function. For instance, the function two creates a list with two copies of the given element by delegating part of the work to singleton.

```
two x = x : singleton x
```

Yet, ultimately nothing very deep arises from delegation to simple functions. After all, we can simply inline the auxiliary functions (i.e., expand the function definitions) to get the same behavior.

```
two' x = x : [x]
```

Hence, while this kind of delegation affords us valuable *code reuse*, we do not gain any *expressive power*.

The situation changes dramatically when functions delegate part of their work to themselves. This outrageously useful concept, whereby a function is defined in terms of itself, is known as *recursion*.

Recursively Producing Lists As a first example of a recursion, we write the function fromTo that builds the lists of numbers in the given interval.

```
▶ fromTo 1 10
[1,2,3,4,5,6,7,8,9,10]
▶ fromTo 5 8
[5,6,7,8]
▶ fromTo 8 5
[]
```

We define this function compactly as follows:

```
fromTo from to
  | from > to  = []
  | otherwise = from : fromTo (from + 1) to
```

We distinguish between two cases:

- in case the interval is empty we return the empty list, and
- in case the interval is non-empty, we return the non-empty list that starts with the first element of the interval and follows with the remaining interval.

We call this function *recursive* because, in the case of a non-empty interval, it calls itself recursively to create the tail of the list.

Observe that we could not have written the function fromTo without recursion. Indeed, recursion greatly increases the expressive power and makes the Haskell language *Turing complete*.[1] This makes recursion a key concept which is used pervasively in most Haskell programs and is vital to master: it is for functional programming what loops are for imperative programming.[2]

Termination Observe that not all of the work is left to the recursive call: the function itself adds the first element of the interval. Clearly, if all of the work were left to the recursive call, nothing would ever be achieved! Indeed, the most direct way of writing an infinite loop in Haskell is by writing a function that does nothing more than call itself recursively.

```
loop x = loop x
```

In practice, programmers end up writing more convoluted forms of such *non-terminating* recursion by accident. Hence, it is essential to make sure all uses of recursion are terminating. The key idea is to ensure that

(a) the amount of work for the function is larger, in a sense to be identified by the programmer, than that for the recursive call, and that

(b) starting from any amount of work a base case is reached in a finite number of steps; a base case is one that is handled directly without recursive calls.

In the case of the fromTo function we can show termination by characterising the amount of work with the size of the interval to be generated, which is a natural number. We can verify that condition (a) is satisfied as the size of the interval clearly decreases by 1 in the recursive call. Condition (b) is also met because the interval becomes empty after a finite number of steps[3] and an empty interval is generated straight away without further recursion.

Recursively Consuming Lists Now we write a function that takes a list apart. This function sum sums the numbers in the list.

[1] If you don't know what this means, do look it up.

[2] Arguably recursion is more fundamental than loops, as it also arises frequently in mathematics and in art, e.g., see Matryoshka Dolls and the Droste effect.

[3] the number of steps is equal to the size of the interval

```
► sum [ ]
0
► sum [1,2,3,4,5]
15
► sum (fromTo 1 10)
55
```

Again we write a recursive function.

```
sum list
  | null list    = 0
  | otherwise = head list + sum (tail list)
```

This function uses guards to distinguish between the empty list, the base case, and the non-empty list, the recursive case. The recursive call is applied to a component of the original list, in this case its tail. This is known as *structural recursion*.

There are two prominent reasons why structural recursion is a very common pattern for defining functions. Firstly, it is very natural and easy to define a function that is guided along by the shape of its input data structure. Secondly, structural recursion is very well-behaved as it guarantees termination. Indeed, the recursive call has obviously less work because a component of the input is obviously smaller than the input itself, and ultimately the input data structure is so small it has no further components.

2.4.4 Pattern Matching

Haskell simplifies the case distinction for lists by means of *pattern matching*. Instead of using the null function for case analysis we can use *patterns*. A pattern is a particular shape of list that we are interested in. In the case of sum we distinguish between the pattern [ ] and the pattern (x : xs). This results in the following definition:

```
sum [ ]      = 0
sum (x : xs) = x + sum xs
```

The role of a pattern is two-fold. Firstly, it selects a particular case, just like a guard. Secondly, it defines a number of variables that can be used within the definition of the case. In the example above the pattern (x : xs) defines the variables x and xs that can be used in x + sum xs.

Pattern matching has three important advantages:

1. The combination of the two roles means that we can write the function sum *more compactly*.

2. Pattern matching improves the *readability* because the patterns are written in the same way as the list values that are supplied as input to the function. Hence we can immediately see which case is applicable to a particular input.

3. Pattern matching promotes *equational reasoning*: each of the cases is an independent equation that we can exploit during reasoning about the equality of two expressions.

We can easily define the predefined functions null, head and tail ourselves in terms of pattern matching.

```
null []       = True
null (x:xs)   = False
head (x:xs) = x
tail (x:xs)   = xs
```

In the case of head and tail there is no sensible result for the empty list []. Hence this case is not defined for both functions. That is why we call them *partial functions*, functions that are not defined for their entire domain of inputs.

2.5 Types

So far we have said very little about *types*, while Haskell is actually quite renowned for its advanced type system. Yet, in contrast to many other statically typed programming languages, the programmer does not have to extensively annotate code with type information (parameter types, return types, type of local variables, ...). A particular part of a Haskell system, the *typechecker*, is able to automatically derive the necessary type information in a process called *type inference*.[4] Based on this derived type information the typechecker validates whether all operations in the program make sense. If not, it signals type errors.

2.5.1 Type Signatures

Even though you do not have to supply type information, there nevertheless are compelling reasons to do so for your user-defined functions in the form of *type signatures*:

- They are useful documentation of the code. A signature indicates how the function is to be used (what parameters to supply and what result to expect) and also gives an indication of what the function may or may not do.

  For example, the online search engine *Hoogle*[5] facilitates finding predefined Haskell functions based on a desired type.

- The signature provides an (abstract) specification of the function being defined. If the specification does not agree with the definition, then the typechecker signals this discrepancy. In this way it is possible to detect bugs early in the development process.

Now consider how to write type signatures. We repeat our first function, now with an explicit type signature:

[4] Type Inference was developed by Hindley and Milner in the 70s.
[5] `http://hoogle.haskell.org/`

```
f :: Int → Int
f x = x + 1
```

The type signature is written on the line above the function definition. It consists of the name of the function f followed by :: and then the function type Int → Int. This type Int → Int is the type of functions that take a value of type Int and map it to a value of type Int.

Here are more of our functions, now with a type signature.

```
g :: Int → Int → Int
g x y = x + 2 * y

passingGrade :: Int → Bool
passingGrade p = p > 9

rate :: Int → Double
rate b =
  if b > 250000
    then 1.00
    else 1.40
```

As you can see the types of the parameters and results can be different. Multiple parameters are separated by arrows →.

2.5.2 Algebraic Datatypes

Haskell programmers are not limited to the predefined datataypes like Int and Bool, but can also create their own new types. This happens in the form of *algebraic datatypes* (ADTs).[6]

Alternatives A new Haskell ADT is defined by means of a **data**-declaration that enumerates the alternative constructors. Here is an ADT for the colors of a traffic light:

```
data Light = Red | Orange | Green
```

The name of the type is Light. It distinguishes three different values, Red, Orange and Green, based on the corresponding constructors.

The following function makes use of the Light ADT:

```
next :: Light → Light
next Red    = Green
next Orange = Red
next Green  = Orange
```

[6] Do not confuse these with abstract datatypes, also abbreviated by ADT.

The function next illustrates that pattern matching is also available for user-defined datatypes.

We call Light a *sum type*: the values of this type are the "sum" (or union) of the values that can be constructed by the three different constructors.

$$\text{Light} = \{\text{Red}\} \cup \{\text{Orange}\} \cup \{\text{Green}\}$$

In languages like C++ and Java we call the above often enum-types or enumerations of values.

Observe that the predefined type Bool is another example of such a sum type. You can define it yourself as follows:

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

Combinations A quite different kind of ADT is the following Person ADT that represents people with their name and age.

```
data Person = P String Int
```

The name of this type is Person. It provides a single constructor P. This constructor has two parameters or *fields*. The first field is of the predefined type String and represents the name of the person. The second field is of type Int and represents the person's age. Here is a possible value:

```
tom :: Person
tom = P "Tom" 34
```

The order of the fields is important and has to conform to the order of the field types in the **data**-declaration.

Here are a few functions over people:

```
mkPerson :: String → Int → Person
mkPerson name age = P name age

getName :: Person → String
getName (P n a) = n

getAge :: Person → Int
getAge (P n a) = a
```

An ADT like Person is also called a *product* type because there is a value of this type for every element in the (Cartesian) product of the field types.

$$\text{Person} = \{\text{P n a} \mid (\text{n}, \text{a}) \in \text{String} \times \text{Int}\}$$

Sums of Products In general we can define an ADT as a sum of products: multiple constructors can each have a number of fields. For instance, here is an ADT for geometric shapes that represent circles and rectangles. Circles have a radius and rectangles have a width and height.

```
data Shape
  = Circle Double
  | Rectangle Double Double
area :: Shape → Double
area (Circle radius)          = radius * radius * pi
area (Rectangle width height) = width * height
```

The set of all possible values of type Shape is:

$$\text{Shape} = \{\text{Circle r} \mid \text{r} \in \text{Double}\} \cup \{\text{Rectangle w h} \mid (\text{w}, \text{h}) \in \text{Double} \times \text{Double}\}$$

Recursively Defined ADTs Finally, in the most general form, new ADTs can also be defined in terms of themselves. Here is an example, an ADT of binary trees.

```
data Tree
  = Empty
  | Branch Int Tree Tree
```

A tree is either Empty or it is a Branch with an element of type Int and two subtrees. These subtrees in turn are either empty or branches... The set of trees of type Tree satisfies the recursion equation:

$$\text{Tree} = \{\text{Empty}\} \cup \{\text{Branch e l r} \mid (\text{e}, \text{l}, \text{r}) \in \text{Int} \times \text{Tree} \times \text{Tree}\}$$

Mathematically speaking this is not a unique definition for the set Tree, because Tree appears both on the left- and right-hand side of the equation. For now we restrict ourselves to the obvious solution of this equation, but later we come back to another solution.

The obvious solution is that trees are *inductively* defined. This means that every non-empty tree is built from "smaller" subtrees that in turn are defined of yet smaller trees. In other words, every tree is finite.

Recursively defined ADTs naturally give rise to recursive functions over them. For instance, the recursive function sumTree sums the numbers in the given tree.

```
sumTree :: Tree → Int
sumTree Empty          = 0
sumTree (Branch n l r) = n + sumTree l + sumTree r
```

More precisely, sumTree is defined by structural recursion. Its second clause, handling a branch, features two recursive calls, one for each of the two subtrees of the branch.

2.5.3 Parametric Polymorphism

Parametric polymorphism is known in the context of object-oriented languages as *genericity*. It is an important abstraction mechanism of Haskell.

The Problem Consider the following definition:

```
id x = x
```

This function is the identity function; it simply returns its parameter.

What is the type signature of id? Based on what we have seen so far, many signatures are possible: Int → Int, Double → Double, Light → Light, ... All these signatures are valid because the function does not exploit any essential property of its parameter.

If we need an identity function for many different types, we would have to write different versions of the same function, each with a different type signature but the same implementation:

```
id1 :: Int → Int
id1 x = x
id2 :: Double → Double
id2 x = x
id3 :: Light → Light
id3 x = x
```

This repetition of code is of course sinful. Moreover, id causes a problem for Haskell's type inference. What type signature should be inferred automatically? That is ambiguous, and of course it is rather annoying if type inference were to pick one arbitrarily, if it did not correspond to what the programmer needs.

The Solution The solution to the above problem is to extend the simple type system we have seen so far with a new concept that allows us to express that the identity function works at all types. This is the concept of *type parameters*. We denote these type parameters with lower case letters like a, b, ... A type parameter is a placeholder for any type. We use them as in the type signature below:

```
id :: a → a
id x = x
```

This signature uses one type parameter a that occurs twice. At every call of the function id the type parameters are *implicitly* filled in by the typechecker. For example, at a call id True the parameter a is instantiated by Bool and at a call id Green it is instantiated by Light.

We call a signature that contains type parameters *polymorphic*, in contrast to *monomorphic* signatures that do not contain type variables.

This polymorphism also solves the ambiguity problem for type inference. Even though there are several monomorphic signatures for id, the polymorphic one is more general as it covers all the monomorphic ones. We call this polymorphic signature the *principal* signature as it covers any other valid signature we can provide for id.

Here is an example of a polymorphic function with two type parameters:

```
const :: a → b → a
const x y = x
```

Polymorphic ADTs Not just functions can be polymorphic, but ADTs too. A good example is the recursive Tree type we saw earlier. The elements of this type of tree are of type Int. What if we also want to have trees with elements of type Bool, Light or . . . ? Instead of writing a new **data**-declaration for every variant, we can just write a single polymorphic one:

```
data Tree a = Empty | Branch a (Tree a) (Tree a)
```

The type Tree is parameterised in an element type a that we can instantiate in whatever way we want. For example Tree Int corresponds to our earlier monomorphic version and Tree Bool is a tree of Bool values.

Observe that Tree itself is not a proper type. We call it a *type constructor*. It first has to be applied to an element type before we obtain a proper type.

Of course polymorphic ADTs easily lead to polymorphic functions. Here is an example:

```
tlength :: Tree a → Int
tlength Empty          = 0
tlength (Branch x l r) = 1 + tlength l + tlength r
```

2.5.4 Predefined ADTs

It is useful to be aware of several predefined ADTs in Haskell.

Unit The *unit* type is a type with a single value. Its pseudo-declaration is:

```
data () = ()
```

In other words, the type has the special name () and a single value also written (). At first sight, this type is pretty useless because its value is known a priori. Yet, we will see some useful applications later.

Tuples Haskell provides a series of predefined generic tuple types. The pseudo-definition of the binary tuple type is:

```
data (a, b) = (a, b)
```

with predefined projection functions:

```
fst :: (a, b) → a
snd :: (a, b) → b
```

Tuples are useful to return multiple results from a function. An example of this practice is the quotRem function that returns both the quotient and remainder after division.

```
▶ quotRem 10 3
(3, 1)
```

Haskell also supports tuples with more than two components. However, no projection functions are defined for them.

```
info :: (String, Int, Double)
info = ("Tom", 34, 1.94)
```

Generic Sum Type The predefined generic sum type is Either.

```
data Either a b
  = Left a
  | Right b
```

In contrast to tuples, the type Either is not used often. In practice, new custom ADTs are defined with meaningful constructor names. Perhaps one reason is that Haskell does not provide special syntax for Either.

Lists The pseudo-definition of the polymorphic list type is:

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

2.5.5 Type Synonyms

Besides defining *new* ADTs, it is also possible to introduce an additional name for an existing type. This additional name is known as *type synonym* or *type alias*. It can be freely exchanged for the original name of the type. Here is an example:

```
type Distance = Int
```

Now we can write Distance wherever Int is expected, and vice versa.

Type synonyms are used to indicate that a particular generic type, e.g., Int, is used to denote a specific concept, e.g., distance. Implicitly and informally this entails that not every operation on this use of the existing type makes sense. For instance, it does not make sense to add a non-distance to a distance. Yet, type synonyms do not disallow such improper operations. To make an explicit distinction between Distance and Int, a new ADT has to be defined.

```
data Distance = D Int
```

The disadvantage of this approach is that all useful functions for Ints, like addition, have to be redefined for Distance. This is where the advantage of type synonyms lies: such redefinition is not needed.

Hence there is clearly a trade-off between type synonyms and new ADTs. Either one has less programming work, or one gets more support from the typechecker. For that reason type synonyms are often used in the *rapid prototyping* phase of software development, while custom ADTs are created in the development of robust software.

Strings A prominent application of type synonyms in Haskell are Strings. Most languages have a separate type for strings to represent text. However, in Haskell strings are just lists of characters. The String type synonym is simply defined as:

```
type String = [Char]
```

The special notation "Tom" is just syntactic sugar for the regular list notation ['T','o','m']. The advantage of strings as lists is that all polymorphic list functions also apply to strings.[7]

```
▶ head "Tom"
'T'
▶ reverse "Tom"
"moT"
▶ length "Tom"
3
```

2.6 List Comprehensions

The well-known *comprehension* notation is used in mathematics to define sets. For instance, assume $S = \{1,2,3,4,5,6,7,8,9,10\}$. Then we can write the R of squares of elements of S

[7] The disadvantage is that strings in Haskell are not represented in memory in the same compact way as in other programming languages. For that reason, Haskell also features alternative libraries with more compact representations like Text.

smaller than 5 in this way:

$$R = \{n * n \mid n \in S \wedge n < 5\}$$

Of course R is equal to $\{1, 4, 9, 16\}$.

In Haskell we can use essentially the same comprehension notation to define lists.

```
s :: [Int]
s = [1,2,3,4,5,6,7,8,9,10]
r :: [Int]
r = [n * n | n ← s, n < 5]
```

The syntax `n ← s` in the comprehension is called a *generator* and the syntax `n < 5` is called a *guard*. The generator generates candidate values `n` from the list `s` and the guard selects only the valid ones. These selected values are returned in the resulting list.

A beautiful example of list comprehensions is the following definition of the well-known quicksort algorithm.

```
qs :: [Int] → [Int]
qs []           = []
qs (pivot : xs) = qs [l | l ← xs, l ⩽ pivot]
                  ++
                  [pivot]
                  ++
                  qs [h | h ← xs, h > pivot]
```

A list comprehension can also consist of multiple generators and guards. Here is an example with two generators that computes the Cartesian product of two lists.

```
cp :: [a] → [b] → [(a, b)]
cp l r = [(a, b) | a ← l, b ← r]
```

Observe the order in which the pairs are generated:

```
► cp [1,2] ['a','b']
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

As you can see, for every element produced by the first generator, all elements of the second generator are enumerated. The effect is similar to that of a nested loop in imperative programming languages.

3. Functions

This chapter shows that functions are first-class citizens in Haskell. We call them that because they are the equal of any other type of value in the language. This means that they can appear anywhere where other values do: as parameters to functions, as the result of a function, as value stored in a variable, in a datastructure, ...

The most prominent of these first-class rights is the possibility to pass around functions as parameters.

3.1 Function Parameters

Here are the definitions of two functions sum and prod that compute the sum and product of a list of integers respectively.

```
sum :: [Int] → Int
sum [ ]      = 0
sum (x : xs) = x + sum xs

prod :: [Int] → Int
prod [ ]      = 1
prod (x : xs) = x * prod xs
```

What immediately jumps out is how similar the definitions of these two functions are. They clearly have the same inductive pattern: there is a base case (0 or 1) for the empty list, and for the non-empty list the recursive result (sum xs or prod xs) is combined with the first element x to obtain the overall result by means of a function (+ or *).

We can capture this common pattern in a new function foldr:

```
foldr f z [ ]      = z
foldr f z (x : xs) = f x (foldr f z xs)
```

We have introduced two parameters to deal with the points where sum and prod differ: z for the base case and f to combine the recursive result with the first element.

Now we can define sum and prod in terms of this common pattern.

```
sum l = foldr (+) 0 l
prod l = foldr (*) 1 l
```

These two definitions are very compact because the inductive logic has been isolated in the foldr function. It is instrumental to consider the type signature of this function:

```
foldr :: (e → a → a) → a → [e] → a
```

This signature reflects that foldr has three parameters, which are in reverse order of appearance:

- the list of type [e], where the type of the elements e is unconstrained;
- the result z for the base case of any type a, which is also the result type of foldr as a whole; and
- the function f with function type e → a → a, because it is applied to an element in the list (e) and a recursive result (a) to yield a new result (a).

Because of the first function parameter f we call foldr a *higher-order function*, and a first-order function in particular.

The order of a function is defined (inductively, how else?) as follows:

- a **0th-order function** has only non-function parameters; and
- an **(n+1)th-order function** has one or more function parameters, the highest order of which is *n*.

In practice we most often encounter 0th-order and 1st-order functions, while 2nd- and higher-order functions are rather rare. That is why the particular order is usually not relevant and we typically only speak of higher-order functions when we want to indicate that there is a function parameter.

The Behavior of foldr The function name foldr is short for *fold right*. In other functional languages this function is also called *reduce*, because it reduces a list datastructure to a single value (e.g., a sum). The recursion scheme that is captured in this function is not restricted to lists, but can be easily generalized to other inductive datatypes.

We can diagrammatically clarify how this recursion scheme affects a list as follows:

```
    a : (b : (c : []))

==> (foldr f z)

    a `f` (b `f` (c `f` z ))
```

As we can see the impact of foldr is to replace the list constructors (:) and [ ] with the custom values f and z. Using terminology from *category theory* we call the combination of f and z an *algebra* for lists. By means of foldr the algebra assigns a (new) meaning to the list, which on its own is a meaningless or uninterpreted datastructure.

Let us apply this principle, to replace the constructors by parameters, to the previously presented type of trees:

```
data Tree a = Empty | Branch a (Tree a) (Tree a)
foldTree :: (e → a → a → a) → a → Tree e → a
foldTree f z Empty          = z
foldTree f z (Branch x l r) = f x (foldTree f z l)
                                  (foldTree f z r)
```

Function Definitions in Terms of foldr Here we provide a recipe for defining functions in terms of a call to foldr,

```
h l = foldr f z l
```

where the parameters f and z have to be instantiated with the appropriate expressions. This recipe is known as *the universal property of* foldr. Formally:

$$\texttt{h l = foldr f z l} \quad \Leftrightarrow \quad \begin{cases} \texttt{h [ ]} & = & \texttt{z} \\ \texttt{h (x : xs)} & = & \texttt{f x (h xs)} \end{cases}$$

If we know what h is, we can attempt to solve the system of two equations on the right for z and f.

To make the recipe concrete, we illustrate it on the function h ≡ subsequences. The function subsequences :: [a] → [[a]] returns all ordered sublists of the given list. The order in which the sublists are returned is not important. Here is an example:

```
> subsequences [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

We anticipate the following shape for the function definition:

```
subsequences :: [a] → [[a]]
subsequences l = foldr f z l
```

We determine the two parameters f and z thus:

- The z parameter is the easiest to determine. It is obtained from the first of the two equations, with h ≡ subsequences:

$$\texttt{subsequences [ ] = z}$$

In other words, z is the expected result for the empty list. Clearly an empty list has only one sublist: the empty list itself. Hence we expect:

```
> subsequences [ ]
[[ ]]
```

and we partially instantiate the definition accordingly:

```
subsequences :: [a] → [[a]]
subsequences l = foldr f [[ ]] l
```

- To determine the parameter f, we solve the second equation.

$$\text{subsequences } (x : xs) = f\ x\ (\text{subsequences } xs) \tag{3.1}$$

This equation is concerned with how the function behaves itself with respect to a list of the form x : xs. In particular we have to characterize the behavior in terms of x and the result r that we obtain from recursively applying the function to the tail xs (i.e., r ≡ subsequences xs). The way x and r are combined determines the function f.

In our example we have to figure out how we obtain the subsequences of the list x : xs given x and subsequences xs. This follows from the following two insights:

  - Every subsequence of xs is also a subsequence of x : xs.
  - If we stick x in front of a subsequence of xs, we obtain a subsequence of x : xs.

Hence, we can say that:

$$\text{subsequences } (x : xs) = [x : ys \mid ys \leftarrow \text{subsequences } xs] +\!\!+ \text{subsequences } xs$$

If we fill in this insight in Equation 3.1, we get:

$$[x : ys \mid ys \leftarrow \text{subsequences } xs] +\!\!+ \text{subsequences } xs = f\ x\ (\text{subsequences } xs)$$

By abbreviating, the recursive result subsequences xs to r, we get:

$$[x : ys \mid ys \leftarrow r] +\!\!+ r = f\ x\ r$$

If we turn this equation around, we get a valid definition for f:

```
f :: a → [[a]] → [[a]]
f x r = [x : ys | ys ← r] ++ r
```

Lastly, we would like to point out that not every function can be written as a call to foldr.

3.1.1 Map

A second frequent recursion scheme is that of map:

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

This higher-order function transforms every element in a list by means of the given function f. For instance, capitalize replaces every letter in a String by the corresponding capital using the predefined function toUpper :: Char → Char.

```
capitalize :: String → String
capitalize s = map toUpper s
```

3.1.2 Filter

A third useful higher-order function is filter:

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

This higher-order function removes all elements from a given list that do not satisfy the given *predicate* p.

3.2 Support for Higher-Order Functions

Because Haskell wants to encourage the use of higher-order functions, it provides special support that facilitates their use. This support allows to quickly write new functions (typically function parameters) in terms of existing functions, and is based on ideas from the λ-calculus.

3.2.1 Local Definitions

Suppose that we want to square all elements in a list. We immediately identify the map recursion scheme as appropriate for this task. The function to apply to every element is the squaring function. We write the auxiliary function square to capture this.

```
squarelist :: [Int] → [Int]
squarelist l = map square l

square :: Int → Int
square x = x * x
```

Now square is a toplevel definition that can also be used elsewhere. Often this is not intended. When writing small auxiliary definitions—e.g., to serve as a parameter of a higher-order function—we only intend them to be used locally and do not wish to create any additional dependencies further away that, e.g., would make it more onerous to refactor the program. For this purpose Haskell provides local definitions that have a limited scope. In fact, Haskell provides two different syntactic forms: **where** blocks and **let**…**in**… expressions. This is what the program looks like with both styles of local definitions:

```
squarelist :: [Int] → [Int]
squarelist l    = map square l
  where
    square x = x * x
```

and

```
squarelist :: [Int] → [Int]
squarelist l =
  let square x = x * x
  in  map square l
```

There is no recommended form; both are equally 'good'. There are some small differences in their use. The **where** block is local to another definition, in this case that of squarelist, is visible everywhere in that definition. The **let**…**in**… expression can be used anywhere an expression can be written. The definitions it creates are visible everywhere in the **let**…**in**….

Both support multiple definitions, simply by writing them on subsequent lines.For example,

```
squarelist :: [Int] → [Int]
squarelist l    = result
  where
    square x = x * x
    result   = map square l
```

3.2.2 Anonymous Functions

The function square is actually rather trivial, and it makes little sense to define and name a special-purpose function for the job. For this reason and inspired by the λ-calculus, Haskell allows to compactly write anonymous functions for specifying function parameters.

```
squarelist :: [Int] → [Int]
squarelist l = map (λx → x * x) l
```

The notation $\lambda x \to x * x$ approximates the symbolic λ-calculus notation $\lambda x.x * x$. The parameters of the anonymous function are listed between the backslash and the arrow, while the body of the function is given after the arrow.

3.2.3 Currying and Partial Application

As first-class citizens functions can be returned from other functions just like any other type of value. Here is an example.

```
add :: Int → (Int → Int)
add x = λy → y + x
```

The function add takes a number x and returns an (anonymous) function of type Int → Int that adds x to the function parameter y.

This way we obtain for instance the increment function:

```
inc :: Int → Int
inc = add 1
```

```
► inc 5
6
```

Compare the above higher-order definition of add to the following similar function add′:

```
add′ :: Int → Int → Int
add′ x y = y + x
```

The function add′ is seemingly a 0-order function with two parameters that is fundamentally different from add. Yet in fact there is no difference between both functions in Haskell! We call this concept *currying*: every function with n parameters is actually simulated by an $n-1$th-order function that takes the first parameter and returns an $n-2$th-order function that processes the remaining parameters.

Haskell conveniently allows us to write n-parameter functions in 0th-order style. Yet it is very useful to be aware of the actual underlying higher-order representation as we can exploit it to compactly derive *new* functions from existing ones.

We can for instance define inc in terms of add′:

```
inc = add′ 1
```

This technique of applying a function to fewer than all of its parameters is called *partial application*.

3.2.4 Operator Sections

The partial application of operators is possible too. We can for instance define inc directly in terms of the operator +.

```
inc = (+) 1
```

A function that checks whether a number is strictly positive cannot be defined easily in terms of partial application of >, because we want to supply the second argument but not the first one. Fortunately, there is special syntax for just that, called *operator section*.

```
positive :: Int → Bool
positive = (>0)
```

This is a right operator section. A left operator section is possible too: (0>).
Here are a few examples:

```
► map (>0) [-1,0,1]
[False,False,True]
► map (0>) [-1,0,1]
[True,False,False]
► map (*2) [1..5]
[2,4,6,8,10]
```

3.2.5 Eta-reduction

A final convenient feature is η-reduction, which Haskell has copied from the λ-calculus. With η-reduction we can rewrite this:

```
► map (λx → signum x) [-10,0,20]
[-1,0,1]
```

into the more compact form:

```
► map signum [-10,0,20]
[-1,0,1]
```

because $\lambda x \rightarrow$ signum x and signum are equivalent.
We can also apply η-reduction to conventional function definitions. Instead of:

```
sum l = foldr (+) 0 l
```

we can simply write:

```
sum = foldr (+) 0
```

3.3 Stylish Higher-Order Code

A very compact, but very readable[1] programming style aims to write function definitions without mentioning function parameters explicitly. This style is called *point-free* because it does not mention the so-called *points* or values.

3.3.1 Function Composition

Curiously the point-free style makes heavy use of "points":

```
(∘) :: (b → c) → (a → b) → (a → c)
f ∘ g = λx → f (g x)
```

This dot-operator implements function composition[2] in Haskell.

We illustrate the point-free style with an example: counting the number of words in a text. First we write the traditional *point-wise* solution:

```
wc :: String → Int
wc str = length (words str)
```

where the function words :: String → [String] splits a string into a list of words and the function length :: [a] → Int yields the length of a list.

In point-free style we obtain a more compact definition by not mentioning the parameter str:

```
wc' :: String → Int
wc' = length ∘ words
```

We say that the function wc' is defined as the composition of length and words. Obviously the order of composition matters: the function length is applied to the result of words.

3.3.2 Pipelines

Of course more than two functions can be composed in this way. The result is called a *pipeline*. The data flows from right to left through such a pipeline. Here is small example:

[1] after some familiarisation

[2] well-known in mathematics where it is written as ∘

```
squareSumEvens :: [Int] → Int
squareSumEvens = sum ∘ map (λx → x * x) ∘ filter even
```

It may take some getting used to this style, but once you have mastered it, you will appreciate how easy it is to read this kind of definition.

3.3.3 Point-wise Pipelines

The point-wise counterpart of function composition (∘) is function application. In Haskell function application is denoted with a space, like the space in `f x`. If you want to apply multiple functions in succession, you have to use parentheses to group the function calls as in `f (g x)` because `f g x` is equivalent to `(f g) x` which has an entirely different meaning.

However, Haskell offers also an explicit operator ($) for function application:

```
($) :: (a → b) → a → b
f $ x = f x
```

This operator is very convenient because it has a very low priority and associates to the right. This means that `f $ g x` denotes `f (g x)`, but uses 1 fewer character to do so.

Hence, we can write squareSumEvens in an analogous point-wise style:

```
squareSumEvens :: [Int] → Int
squareSumEvens l =
  sum $ map (λx → x * x) $ filter even l
```

As you can see, it is easy to switch between point-wise and point-free style with ($) and (∘).

3.3.4 Dataflow Inversion

Finally, we mention a variant of ($) that is popular in the functional language F#:

```
(▷) :: a → (a → b) → b
x ▷ f = f x
```

This operator associates to the left with low priority and swaps the parameters of ($). This yields a more conventional left-to-right dataflow, similar to bash pipelines.

```
squareSumEvens :: [Int] → Int
squareSumEvens l =
  l ▷ filter even ▷ map (λx → x * x) ▷ sum
```

In the same way you can define an operator for function composition where the data flows from left to right. In the area of category theory this operator is sometimes denoted as ; such that $f;g \equiv g \circ f$. Unfortunately, ; is a reserved symbol in Haskell that cannot be used as an operator. Hence, you need to come up with a different operator name for the job.

4. Type Classes

Type classes are one of the most remarkable and famous features of Haskell.

4.1 Adhoc Overloading

Many programming languages overload operators like + and ∗. For example in Java + can be used for adding both integers and floating-point numbers. We say that the operator + is *overloaded* because it provides different implementations for different types of parameters. We say that the overloading is *adhoc* because the behavior of the operator is not uniformly defined for every type, as compared to parametric polymorphism, but is individually defined for every type.

Overloaded operators are the cause of much envy in programmers if the language does not permit them to use + for the addition of user-defined datatypes (e.g., Matrix). Then the developers of the language are privileged, and they have decided once and for all which operators are overloaded and which types they support. Even though it would be convenient to use + for custom types, most languages simply do not permit this.

There are however languages that do support user-defined overloading, and C++ is perhaps the best-known example. Yet, the way Haskell supports user-defined adhoc overloading is particularly remarkable because of its elegance.

The following examples illustrate the overloading of ≡ in Haskell:

```
► 'a' ≡ 'b'
False
► 1.0 ≡ 1.0
True
► True ≡ True
True
```

We can apply ≡ to parameters of different types: Char, Double, Bool, . . . but not all types.

4.2 Type Classes

To indicate that ≡ is an overloadable function, it is declared in a type class.

```
class Eq a where
  (≡) :: a → a → Bool
```

This declaration defines the type class with name Eq. The type class has one type parameter a and one function (≡). Type class functions are called *methods*. Only the type signature of the method (≡) is given. Every concrete type T can supply a different implementation of the method (≡); the only requirement is that this implementation has type T → T → Bool, which we obtain by substituting the type a with T in the original signature.

4.2.1 Instances

To indicate that a particular type implements the methods of a type class, we provide an *instance* of that type class for that type. Here we see the instance of Eq for the type Bool:

```
instance Eq Bool where
  True ≡ True   = True
  False ≡ False = True
  _    ≡ _      = False
```

The type class Eq is already predefined in Haskell, and instances for most predefined types are provided as well. However, when defining a new ADT, a corresponding Eq instance does not come into existence automatically. Consider the Light ADT; if we want to compare values of this type, GHCi points out that there is no applicable implementation of (≡):

```
▶ Green ≡ Green
<interactive>:1:7:
  No instance for (Eq Light)
    arising from a use of ≡
  Possible fix: add an instance declaration for (Eq Light)
  In the expression: Green ≡ Green
  In an equation for 'it': it = Green ≡ Green
```

At the same time we get advice for how to solve the problem: simply declare an Eq Light instance. Let us do so:

```
instance Eq Light where
  Red    ≡ Red    = True
  Green  ≡ Green  = True
  Orange ≡ Orange = True
  _      ≡ _      = False
```

Observe that we have a lot of freedom in how we want to define this instance. The above example implements structural equality. However, this is not required; we can opt for a more semantic notion of equality instead. For example, we can define evaluation for a datatype of expressions:

```
data Exp = Lit Int | Add Exp Exp
eval :: Exp → Int
eval (Lit n)       = n
eval (Add e₁ e₂) = eval e₁ + eval e₂
```

Now we can state that two expressions are equal if they evaluate to the same result:

```
instance Eq Exp  where
  e₁ ≡ e₂ = eval e₁ ≡ eval e₂
```

Observe that in the above declaration the first occurrence of ≡ denotes equality of expressions, while the second occurrence denotes equality of integers.

4.2.2 Laws

Do we have complete freedom in how we implement a particular type class instance? No, not every well-typed definition of ≡ makes sense as a notion of equality. We expect to have only implementations that actually capture equivalence relations. The fact that ≡ should be an equivalence relation, can be captured in three *laws*:

$$x \equiv x \equiv \mathsf{True} \qquad \text{(Reflexivity)}$$
$$x \equiv y \equiv y \equiv x \qquad \text{(Symmetry)}$$
$$x \equiv y \mathbin{\&\&} y \equiv z \equiv x \equiv z \mathbin{\&\&} y \equiv z \qquad \text{(Transitivity)}$$

This is the minimal sensible specification of (≡). Unfortunately, these laws cannot be expressed in the Haskell language itself, nor can they be enforced by the Haskell compiler. Just like in other languages we have to include this supplementary specification in comments and code documentation. Nevertheless, these laws are deemed highly important and it is expected that the implementer of an instance verifies that his implementation satisfies them.

4.2.3 Multiple Methods

A type class is not limited to a single method. Typically multiple related methods are bundled in the same type class. The Eq class is an example of this practice. It not only contains the equality test (≡) we have shown above, but also the inequality test (≠).

```
class Eq a  where
  (≡) :: a → a → Bool
  (≠) :: a → a → Bool
```

This method comes of course with an additional law, which expresses the connection between the two methods:

$$x \equiv y \equiv \mathsf{not}\ (x \not\equiv y) \qquad \text{(Consistency)}$$

4.2.4 Default Implementations

To lighten the burden of implementing type class instances, Haskell makes it possible to provide so-called *default implementations* of methods in the type class declaration. This is mainly useful if the type class contains multiple methods that can be defined in terms of each other. For example, the full declaration of the Eq type class makes use of default implementations to define $(\equiv)$ and $(\not\equiv)$ in terms of each other following the (Consistency) law.

```
class Eq a where
  (≡) :: a → a → Bool
  x ≡ y = not (x ≠ y)
  (≠) :: a → a → Bool
  x ≠ y = not (x ≡ y)
```

When a new instance is defined, it need not implement both methods. It is sufficient to implement only one of the two; the other comes for free thanks to the default implementation. Of course it is still possible to implement both methods; this is useful when there exists a more efficient implementation for the type at hand than the default implementation. If we forget to implement any method, the instance assumes both default implementations. This is of course problematic because their mutual definition leads to an infinite loop at runtime.

To mitigate this problem it is common to state in the documentation of the type class what the minimal sets of methods are that need to be implemented. For Eq there are two such (singleton) sets: $\{(\equiv)\}$ and $\{(\not\equiv)\}$. Implementing more methods is allowed, but not less.

4.2.5 Constrained Polymorphism

Type class methods are of course to be used within other function definitions. An example is the function nub, which removes duplicates from a list.

```
nub [ ] = [ ]
nub (x : xs) = x : nub (filter (≠ x) xs)
```

What is the most general type signature of this function? In particular, for what types of list elements does this function make sense? If we consider the implementation of nub, then the only restriction on the type a of elements is that it should provide an implementation of the inequality test $(\not\equiv)$. Multiple types do so (e.g., Int and Bool), but certainly not all types (e.g., Int → Int).

Hence, the restriction on the type a is precisely expressed by saying that "there has to be an instance of type class Eq for a". In Haskell we express this restriction with the notation

Eq a.[1] We add this restriction as follows to the type signature:

```
nub :: Eq a ⇒ [a] → [a]
```

We call the function type a *qualified type* because it is subject to a qualification (i.e., an additional condition). The term *constrained polymorphism* is also used to express that the type is not purely polymorphic, but polymorphism with additional constraints on the type parameters. We call Eq a in this role a *type constraint*.

4.2.6 Propagation of Type Constraints

We just saw that nub has a type constraint in its signature because it uses the type class method (≠). In turn nub also "infects" other functions with its type constraint. The function nbDistinct is defined in terms of nub:

```
nbDistinct :: Eq a ⇒ [a] → Int
nbDistinct l = length (nub l)
```

Because nbDistinct calls nub on a list l of type [a], the type constraint Eq a is also imposed in the signature of nbDistinct.

Hence type constraints are contagious, and the type class methods are the ground zero of the contagion; their stand-alone signatures already feature the type constraint. Consider:

```
(≡) :: Eq a ⇒ a → a → Bool
```

The only way to prevent further propagation of a constraint is to fill in a concrete monomorphic type for which a type class instance exists:

```
nbDistinctInts :: [Int] → Int
nbDistinctInts l = nbDistinct l
```

Here we call nbDistinct on a list of type [Int]. Because there is an Eq Int instance, the constraint does not appear in nbDistinctInts.

4.2.7 Polymorphic Instances

We can write instances for polymorphic data types like [a] too. For example, the Eq instance for this type is already predefined as:

[1] Here the name of the type class, such as Eq, plays the role of a *logical predicate* that expresses: this type has an instance of this type class.

```
instance Eq a ⇒ Eq [a] where
  [] ≡ []           = True
  (x : xs) ≡ (y : ys) = x ≡ y && xs ≡ ys
  _ ≡ _             = False
```

Note that the instance itself is subject to the type constraint Eq a, which appears in the head of the instance. After all, to compare lists we have to be able to compare their elements.

By combining the instances Eq Int and Eq Light with this instance for lists, we can perform the following checks.

```
► [1,2,3] ≡ [1,2,3]
True
► [Green, Orange, Red] ≡ [Green, Orange]
False
► [[Green], [Orange], [Red]] ≡ [[Green], [Orange], [Red]]
True
```

Another example of a predefined polymorphic instance is that for tuples:

```
instance (Eq a, Eq b) ⇒ Eq (a, b) where
  (x1, y1) ≡ (x2, y2) = x1 ≡ x2 && y1 ≡ y2
```

Hence two tuples are equal if their components are. Because this requires the equality tests for the two component types, two type constraints arise: Eq a for the first component, and Eq b for the second.

```
► (1, Red) ≡ (1, Red)
True
► (Green, 2) ≡ (Green, 3)
False
```

4.2.8 Show

An important and well-known type class is Show:

```
class Show a where
  show :: a → String
  ..  -- additional methods
```

This class captures the conversion of values of type a to a textual String representation. (The omitted methods are for advanced uses and not discussed here. We can safely ignore them because they have a default implementation in terms of show.)

All predefined types, with the exception of functions, have a Show instance. In fact, we have already made use of these instances without knowing. Indeed, the interactive GHCi environment uses show behind the scenes to obtain textual representations of results that it can print on the screen.

In this example

```
▶ True && False
False
```

the expression True && False is evaluated to the value False. GHCi obtains the text "False" that appears on the screen by evaluating show (True && False). Yet, if we evaluate a trivial value of a new type, GHCi does not know how to display it:

```
▶ Red
<interactive>:1:1:
  No instance for (Show Light)
    arising from a use of Red
  Possible fix: add an instance declaration for (Show Light)
  In a stmt of an interactive GHCi command: print it
```

In order to display the value, we have to write a Show instance first:

```
instance Show Light where
  show Green  = "green light"
  show Orange = "orange light"
  show Red    = "red light"
```

Now GHCi can display the result:

```
▶ Red
red light
```

4.2.9 Subclasses

New type classes can extend existing ones. This is illustrated by the type class Ord a, which states that elements of type a are totally ordered. The class extends Eq a. After all, equality is obviously defined for ordered elements. Hence, the Ord class is defined as a *subclass* of Eq.

```
class Eq a ⇒ Ord a where
  compare :: a → a → Ordering
  (<)     :: a → a → Bool
```

```
(⩾)     :: a → a → Bool
(>)     :: a → a → Bool
(⩽)     :: a → a → Bool
max     :: a → a → a
min     :: a → a → a
```

with

```
data Ordering = LT | EQ | GT
```

where LT is short for "less-than", EQ for "equals", and GT for "greater than".

The class contains many methods, but luckily only one has to be implemented explicitly: either compare or (⩽). The others come for free thanks to default implementations.

Because Ord is a subclass of Eq, it is only possible to create a new Ord instance for a type, if that type already has an Eq instance. That is the case for Light, and so we can write:

```
instance Ord Light where
  compare Green  Green  = EQ
  compare Red    Red    = EQ
  compare Orange Orange = EQ
  compare Green  _      = LT
  compare _      Green  = GT
  compare Red    _      = GT
  compare _      Red    = LT
```

We do not elaborate the laws for Ord, but simply summarize them by saying that Ord should provide a total order.

4.2.10 Automatically Derived Instances

For a limited number of predefined type classes Haskell knows how to automatically derive new instances following a generic recipe. This is useful when the programmer does not want to waste time writing the instance and the generic recipe of Haskell is exactly what is required.

You can make use of this feature by ending a **data** declaration with **deriving** (...), where ... is a list of type class names whose instances should be generated.

For example, we can write for Light:

```
data Light = Green | Orange | Red deriving (Eq, Ord, Show)
```

Observe that automatic derivation of instances only applies to a limited number of predefined type classes, and certainly not for user-defined type classes.[2]

[2] although there is an advanced feature to teach the compiler how to do so

5. I/O

We have not yet studied how a Haskell program communicates with the outside world (other processes, the file system, the user, ...). This chapter shows the exceptional way in which Haskell deals with such I/O tasks.

5.1 Problems with I/O in Haskell

At first blush it seems easy to perform I/O. We simply adopt the approach of other programming languages. This means that we build a number of primitive functions into the language that perform their I/O task through system calls to the operating system. For instance, to read a character from the standard input stream we provide a function getChar :: Char that performs its I/O through system calls and that returns a character which we can use in the program.

While there seems to be no cloud in the sky there are nevertheless two serious issues with this naive approach to I/O. The first—and foremost—issue is a matter of principles and the second of a more pragmatic nature.

5.1.1 Reasoning about I/O

One of the important principles that Haskell has adopted from the λ calculus is that it should be easy to reason about programs. Two properties of the language make this possible.

- **Evaluation preserves equality** With the help of a number of calculation rules, in the form of equalities, we can easily derive equivalent representations of the same program. If we systematically apply this approach we can establish that a function call f x is equivalent to its result v. After all, we obtain the result by rewriting the call in a number of equivalence-preserving steps.

- **The Leibniz Principle** The Leibniz Principle states that expressions e_1 and e_2 are equivalent if they cannot be distinguished in any context g. We can capture this in the following formula:

$$\forall e_1, e_2 . e_1 \equiv e_2 \rightarrow \forall g . g\ e_1 \equiv g\ e_2$$

The combination of these two properties is known as *referential transparency*: the result of a function f only depends on its parameter x, and not on any other factors like its context or how often f has already been called.

An important application of referential transparency is *common subexpression elimination* (CSE): if the same subexpression occurs more than once inside a larger expression, then we can eliminate the duplicates. The following code illustrates this situation:

```
(f x, f x)
```

where the subexpression f x occurs twice. With CSE we can rewrite this expression into the equivalent form

```
let y = f x
in (y, y)
```

You can check that CSE indeed preserves equivalence for all expressions we have covered so far. There is only one exception: the naive getChar function that we have just introduced. Suppose that the standard input consists of the sequence of two characters "ab". Then we likely expect the following result:

```
► (getChar, getChar)
('a', 'b')
```

However, if we apply CSE, then we can rewrite the expression to:

```
► let y = getChar in (y, y)
('a', 'a')
```

Obviously we do not expect that this expression reads more than one character! We clearly have a problem: the two expressions only make sense if they are not equivalent, but if we hold on to our principles they have to be equivalent. Obviously, we are not willing to give in on either account.

5.1.2 Predictability of I/O

Many functional programming languages give up on their principles when it comes to I/O and opt for the pragmatic getChar :: Char approach. Yet, even if we would also prefer a pragmatic solution to a principled one for Haskell, we would nevertheless not be satisfied with the getChar :: Char approach. The reason is Haskell's unusual *lazy evaluation* mechanism. Lazy evaluation is discussed extensively in Chapter 8. Here we will just say that it evaluates just enough of a computation to produce its result, ignoring any unnecessary subexpressions that do not contribute towards the result.

Assume again that the input consists of the sequence "ab", and that we define the following function.

```
twoChars :: (Char, Char)
twoChars = (getChar, getChar)
```

If we evaluate this function, we get the result:

```
► twoChars
('a','b')
```

However, because of lazy evaluation, it may be the case that the tuple is not always fully evaluated. For instance, in this case

```
► fst twoChars
'a'
```

the second component is not needed and hence not evaluated. Similarly, in this case the first component is not evaluated:

```
► snd twoChars
'a'
```

But now the result is `'a'` instead of the expected `'b'`. How can this be? Well, the first component is not evaluated and hence does not consume the first character on the standard input. So, when we read a character during the evaluation of the second component, we obviously get the first character in line: `'a'`.

A similarly unexpected situation arises when we flip the components of the tuple.

```
► (snd twoChars, fst twoChars)
('a','b')
```

We do not get `('b','a')` because the components of the result are evaluated in the opposite order.

The observations we have made on these small hypothetical examples can be generalized. The native approach to I/O in Haskell is not practical in combination with lazy evaluation. In general it is impossible for the programmer to predict when an I/O action is performed. Moreover, the order does not depend on the I/O code itself, but on its context. This is of course unmanageable.

5.2 The IO Type

As we have just seen, the execution of I/O actions in Haskell is problematic, both on a principled and on a pragmatic level. Haskell's ingenious solution to the problem is to simply ban the problematic execution of I/O actions from the language.

5.2.1 Plans of I/O Actions

Instead of executing I/O actions itself, Haskell leaves the dirty work to a "third party". A Haskell program does not execute any I/O actions, but computes a *plan of I/O actions* to be executed. It is up to a third party to execute this plan. This third party is typically the Haskell runtime environment implemented in C or assembler.

The important conceptual change of perspective is that within a Haskell program we build plans of I/O actions instead of executing them. In light of this new perspective the signature `getChar :: Char` is unsuitable because it does not designate `getChar` as the plan for an I/O action that yields a character. Indeed, the type `Char` is the type of a character and not the type of a plan to obtain a character.

Instead, Haskell uses the following actual signature:

```
getChar :: IO Char
```

where the type constructor `IO` indicates a plan for an I/O action, and not the actual execution of said action.

For the sake of brevity we omit "plan for" from now on and simply say IO action when we are talking about a value of type `IO a`.

5.2.2 Building I/O Actions

The Haskell language specification stipulates that the `IO` type is *abstract*. This means that the programmer has no access to its definition. Hence, it is for instance not possible to pattern match on values of the `IO` type.

Primitive Functions The only way to create values of type `IO a` is through primitive functions of this type. The function `getChar :: IO Char` is an example of such a function. Another primitive function is `putChar :: Char → IO ()`, that sends a character to the standard output (e.g., the terminal). Note that the type `()` is used here. It is used to indicate that the I/O action does not yield a noteworthy result; the only reason to execute it is for its side effect.

Another trivial I/O action is created with `return :: a → IO a`. Given a value of type `a`, it builds a plan to produce that value. The execution of that plan is meant to simply return the given value without any actual I/O.

Composing I/O Actions The programmer can obtain more complex I/O actions by composing the primitive functions. For this, there is the primitive combinator `(≫=) :: IO a → (a → IO b) → IO b`. For example,

```
echo :: IO ()
echo = getChar ≫= (λc → putChar c)
```

builds an I/O action that fetches a character and then outputs it. The `(≫=)` operator covers the "and then" part of the previous sentence.

The `return` primitive is the neutral element of composition. For example, `return 'a' ≫= (λc → putChar c)` is the same as `putChar 'a'`. Also, `getChar ≫= (λc → return c)` is the same as `getChar`.

Having such a neutral element is often convenient as a base case. For instance, putStr outputs a String, which is just a list of characters.

```
putStr :: String → IO ()
putStr [ ]       = return ()
putStr (c : cs) = putChar c ≫= (λ_ → putStr cs)
```

In the case of the empty string, there is nothing to be output. This "do nothing" is captured by return (). When composed with outputting the other characters, what happens is just outputting the other characters.

5.2.3 The do Notation

Using the ≫= operator gets unwieldy when chaining more than two actions. Because it is a common pattern, Haskell provides special syntactic sugar, called the do notation, to facilitate it.

Here is an example using this do notation.

```
echo2 :: IO ()
echo2 = do c1 ← getChar
           c2 ← getChar
           putChar c1
           putChar c2
```

The do keyword signals the start of a do-block where every line contains an I/O action that follows the previous one. The result of an I/O action can be given a name with the ← arrow. Subsequent lines can refer to names bound in previous lines.

More useful examples are:

```
echo :: IO ()
echo = do c ← getChar
          putChar c

putStr :: String → IO ()
putStr [ ]       = return ()
putStr (c : cs) = do putChar c
                     putStr cs

putStrLn :: String → IO ()
putStrLn str = do putStr str
                  putChar '\n'
```

5.2.4 Reasoning about Plans of I/O Actions

Simply talking about plans of I/O actions does not result in a conceptual problem.

Two plans are equivalent if they denote the same action. When we have two equivalent plans, we can share the plan (CSE):

```
(getChar, getChar)  ≡  let y = getChar in (y, y)
```

Both expressions have the type (IO Char, IO Char), a tuple of plans.

5.2.5 Executing I/O Actions

Conceptually the execution of I/O actions happens outside the Haskell program. The program itself is only responsible for delivering the plan. Hence, a Haskell program is expected to have a function with the following signature:

```
main :: IO ()
```

The runtime system invokes this function to obtain the plan and then executes it.
An elementary Haskell program looks like:

```
main :: IO ()
main = putStrLn "Hello, World!"
```

You can compile this program with the GHC compiler:

```
$ ghc --make Hello.hs
[1 of 1] Compiling Main                ( Hello.hs, Hello.o )
Linking Hello ...
```

and then execute it:

```
$ ./Hello
Hello, World!
```

You can also try out I/O actions in the GHCi interpreter. In contrast to other types of values, GHCi does not print values of type IO a. Instead it executes them.

```
▶ putStrLn "Hello, World!"
Hello, World!
```

5.2.6 The Shell/Core Metaphor

As we have seen, I/O actions are only executed once they have been computed by the program. Conceptually we say that a Haskell program consists of an outer IO shell, the IO plan that is returned, and a core that consists of pure computations used by the IO plan.

Thin Shell, Fat Core To stick as closely as possible to Haskell's pure principles, it is good style to keep the IO shell of your program as thin as possible, and to allocate as much of the functionality as possible in its pure core. Moreover, tight coupling between both parts of the program should be avoided.

Here is an example of bad style:

```
bad :: Int → IO ()
bad n = go n 0
  where go :: Int → Int → IO ()
        go 0 acc = print acc
        go n acc = do x ← readLn
                      go (n – 1) (acc + x)
```

This is better style:

```
good :: Int → IO ()
good n = do xs ← readNumbers n
            print (sum xs)
```

```
readNumbers :: Int → IO [Int]
readNumbers 0 = return [ ]
readNumbers n = do x  ← readLn
                   xs ← readNumbers (n – 1)
                   return (x : xs)
```

As you can see, this program cleanly separates the program logic, i.e., summing a sequence of numbers, from reading and writing numbers. This approach is not only easier to read, but also promotes reuse. We can now use the pure library function sum to add up the numbers and we can later reuse readNumbers in other applications that do not necessarily add up the numbers they read.

Playing with IO Values As long as IO values have not been embedded in the program shell, they are not executed by the runtime system. This means we can freely manipulate them, for example, in the process of building complex IO values.

For instance, we can split readNumbers into a number of more primitive functions, that are more widely reusable:

1. The central action is readLn :: IO Int which reads a number.
2. We need this action n times:

```
replicate :: Int → a → [a]
replicate 0 _ = [ ]
replicate n x = x : replicate (n – 1) x
```

Note that replicate has a polymorphic type that is not IO-specific. Hence it is highly reusable.

3. Finally, we need to sequence a list of IO actions and collect their results in a list.

```
sequence :: [IO a] → IO [a]
sequence []       = return []
sequence (m : ms) = do x  ← m
                       xs ← sequence ms
                       return (x : xs)
```

The above three functions are all Haskell library functions, just like the common composition of replicate and sequence.

```
replicateM :: Int → IO a → IO [a]
replicateM n m = sequence (replicate n m)
```

By using these library functions, we can write readNumbers much more compactly:

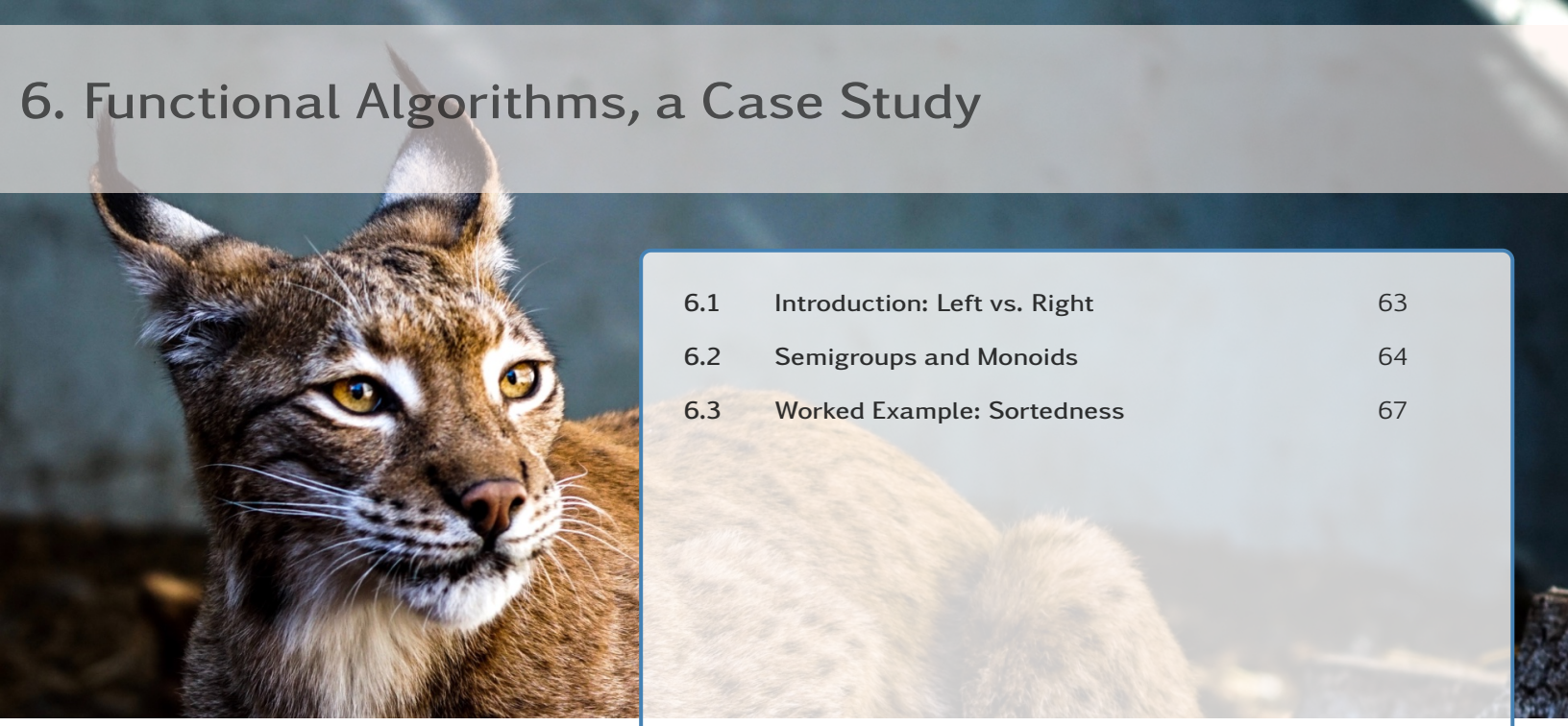
```
readNumbers :: Int → IO [Int]
readNumbers n = replicateM n readLn
```

In the Haskell library, sequence and replicateM have more general types. The library uses the Monad type class to support all types, not just IO, that have return and ≫= methods. This abstraction is discussed more extensively in Chapter 9.

Part II

Advanced Topics

6. Functional Algorithms, a Case Study



In this chapter, we explore an approach to designing algorithms that leverages general functional programming techniques (abstraction, higher-order functions, and the adaptation of algebraic concepts to programming) and Haskell language features like type classes.

6.1 Introduction: Left vs. Right

Haskell offers two structural recursion schemes over lists, the left and the right fold.

```
foldl :: (b → a → b) → b → [a] → b
foldl cl nl []       = nl
foldl cn nl (x : xs) = foldl cn (cn nl x) xs
```

```
foldr :: (a → b → b) → b → [a] → b
foldr cr nr []       = nr
foldr cr nr (x : xs) = cr x (foldr cr nr xs)
```

In the case where the types a and b are the same, we can compare their behavior for the same c and n—the pair of c and n is called the *algebra*.[1]

$$\texttt{foldl c n } [x_1, x_2, x_3, x_4] = \texttt{c (c (c (c n } x_1\texttt{) } x_2\texttt{) } x_3\texttt{) } x_4$$
$$\texttt{foldr c n } [x_1, x_2, x_3, x_4] = \texttt{c } x_1 \texttt{ (c } x_2 \texttt{ (c } x_3 \texttt{ (c } x_4 \texttt{ n)))}$$

The first accumulates values from the left and the second from the right. This can yield very different results. For example,

[1] The mathematical field of *Algebra* studies algebras, and folding is a computer science application of this.

```
> foldl (-) 0 [1,2,3,4]
-10
> foldr (-) 0 [1,2,3,4]
2
```

However, there are algebras where accumulating form the left or the right yields the same output for any list. For example,

```
> foldl (+) 0 [1,2,3,4]
10
> foldr (+) 0 [1,2,3,4]
10
```

or

```
> foldl (++) [] [[1,2],[],[3],[4]]
[1,2,3,4]
> foldr (++) [] [[1,2],[],[3],[4]]
[1,2,3,4]
```

Challenge Can you think of other algebras where accumulating from the left and the right yields the same result?

6.2 Semigroups and Monoids

There are two key properties of the algebra that are responsible for making it impervious to the side of accumulation.

6.2.1 Associativity

The first key property is *associativity*. An operator $(<>) :: A \to A \to A$ is associative, when changing the bracketing of an expression has no impact on the result:

$$x <> (y <> z) \equiv (x <> y) <> z$$

Because the bracketing does not matter for an associative operator, we often drop it and just write $x <> y <> z$.

As we can see in the above examples, $(+)$ and $(++)$ are both associative operators while $(-)$ is not.

In the field of Abstract Algebra,[2] a set A with an associative operator $(<>)$ is known as a *semigroup*. Haskell has adopted this concept in its semigroup type class.

[2] Abstract Algebra is a subfield of Algebra that classifies algebras based on their properties.

```
class Semigroup a where
  (<>) :: a → a → a
```

Any instance of this type class is of course required to provide an associative implementation for the "diamond" operator (<>).

For example,

```
instance Semigroup [a] where
  (<>) = (++)
```

Challenge How many associative Int operators can you think of? Try to get to 10.

6.2.2 Aside: Newtype Wrappers

Two obvious associative operators for Int are (+) and (∗). This poses a problem, because Haskell does not allow two instances of the same type class for the same type. Hence, we can't create two Semigroup instances for Int with these two operators. That makes sense of course, as Haskell could then not know whether (+) or (∗) is intended when it sees the diamond operator (<>) at type Int. It is to avoid this kind of ambiguity that Haskell does not allow multiple instances for the same type.

How then should we deal with this situation when there are multiple sensible algebras? The common approach in Haskell is to create distinct types, that are copies of the original type and that each have their own instance.

We can create a copy of Int by means of a new datatype with one constructor that has one field:

```
data IntCopy = MkIntCopy Int
```

In fact, Haskell has a separate keyword **newtype** for this special case of datatype.[3] Types created with **newtype** are called "newtype wrappers".

```
newtype IntCopy = MkIntCopy Int
```

Typically, the new types are named after their algebra:

```
newtype Sum = Sum Int
newtype Product = Product Int
```

Because they are separate types, they can each have their own Semigroup instance:

[3] **newtype** and **data** behave slightly differently under laziness, but that is beyond this course.

```
instance Semigroup Sum  where
  Sum x <> Sum y = Sum (x + y)
instance Semigroup Product  where
  Product x <> Product y = Product (x * y)
```

In fact, because (+) and (*) are meant to be associative for all instances of Num, the module Data.Semigroup readily provides more general definitions:

```
newtype Sum a = Sum a
newtype Product a = Product a
instance Num a ⇒ Semigroup (Sum a)  where
  Sum x <> Sum y = Sum (x + y)
instance Num a ⇒ Semigroup (Product a)  where
  Product x <> Product y = Product (x * y)
```

It is worth browsing the documentation of Data.Semigroup to learn about other available semigroup instances.

6.2.3 Neutral Element

Associativity on its own is not enough. For example, (++) is associative, but:

```
> foldl (++) [0] [[1,2],[],[3],[4]]
[0,1,2,3,4]
> foldr (++) [0] [[1,2],[],[3],[4]]
[1,2,3,4,0]
```

Hence, also the initial value n of the accumulator matters. Indeed, it must be the *neutral element* of the operator, both on the left and the right.

$$n <> x \equiv x \equiv x <> n$$

In abstract algebra, a semigroup with a neutral element is known as a *monoid*. Hence, Haskell features the Monoid subclass of Semigroup.

```
class Semigroup a ⇒ Monoid a  where
  mempty :: a
```

For example,

```
instance Monoid [a]  where
  mempty = []
```

Challenges

- What are the neutral elements of other associative operations?
- Not all associative operators have a neutral element; can you think of some without a neutral element?

6.2.4 Beyond left vs. right

As we have seen, by using a monoid we get the same behavior from foldr and foldl. However, it goes further than that. Indeed, bracketing all the way to the left and all the way to the right are just the two extremes. We can sprinkle neutral elements anywhere over our expression and choose a corresponding bracketing.

A prominent strategy is *divide-&-conquer* where we recursively split the list in two halves that we process independently and then combine their results. When the processing of the parts happens in parallel, the time complexity even goes from linear to logarithmic![4]

```
foldDC :: Monoid m ⇒ [m] → m
foldDC [ ]   = mempty
foldDC [x] = x
foldDC l     = foldDC l₁ <> foldDC l₂
  where  (l₁, l₂) = splitAt (length l `div` 2) l
```

The computation tree of foldDC is binary:

```
foldDC [x₁, x₂, x₃, x₄, x5, x6, x7, x8]
 =
((x₁ <> x₂) <> (x₃ <> x₄)) <> ((x5 <> x6) <> (x7 <> x8))
```

Here the independent parts of the tree can be processed in parallel. In other words, if we can express an algorithm as a fold with a monoid, it can be parallellized!

6.3 Worked Example: Sortedness

Consider the function isSorted that checks whether a list is sorted.

```
isSorted :: Ord a ⇒ [a] → Bool
isSorted [ ]   = True
isSorted [x] = True
isSorted (x : y : xs) = x ⩽ y && isSorted (y : xs)
```

We would like to obtain a parallellizable version of this function. This basically means that we should come up with a monoid that underpins a divide-&-conquer approach to isSorted.

[4] Provided the splitting does not take linear time, which it does for Haskell lists.

At first blush, we might take Bool as the type of the monoid, && as its associative operator with neutral element True. However, after a little thought, it should be obvious that this won't do. Indeed, from knowing whether two adjacent sublists xs and ys are sorted, we can't always tell whether xs ++ ys is sorted. For example, [1,2] and [3,4] are sorted and so is [1,2] ++ [3,4], yet [3,4] ++ [1,2] is not.

Hence, we need a more crafty type for the monoid, that allows us to compare two adjacent sorted sublists. After exercising our little gray cells some more, we realise that the minimal information we need for that are the first (i.e., smallest) and last (i.e., largest) element of the sorted sublist.

```
data Sortedness a = NotSorted | Sorted a a
instance Ord a ⇒ Semigroup (Sortedness a) where
  NotSorted   <> _           = NotSorted
  _           <> NotSorted   = NotSorted
  Sorted l₁ u₁ <> Sorted l₂ u₂
                | u₁ ⩽ l₂    = Sorted l₁ u₂
                | otherwise  = NotSorted
```

While Sortedness a forms a semigroup, it lacks a neutral element. (Mind that NotSorted is an *absorbing* element, quite the opposite of a neutral element.) So we add one for good measure, which also denotes an empty list, which obviously has no (smallest and largest) elements.

```
data Sortedness a = EmptySorted | NotSorted | Sorted a a
instance Ord a ⇒ Semigroup (Sortedness a) where
  EmptySorted <> s           = s
  s           <> EmptySorted = s
  NotSorted   <> _           = NotSorted
  _           <> NotSorted   = NotSorted
  Sorted l₁ u₁ <> Sorted l₂ u₂
                | u₁ ⩽ l₂    = Sorted l₁ u₂
                | otherwise  = NotSorted
instance Ord a ⇒ Monoid (Sortedness a) where
  mempty = EmptySorted
```

Finally, we can rewrite isSorted with a parallellizable fold.

```
isSorted' :: Ord a ⇒ [a] → Bool
isSorted' = toBool ∘ foldDC ∘ map singleSorted
```

This approach consists of three steps:

1. We turn every element (every singleton list, if you will) into a Sortedness value.

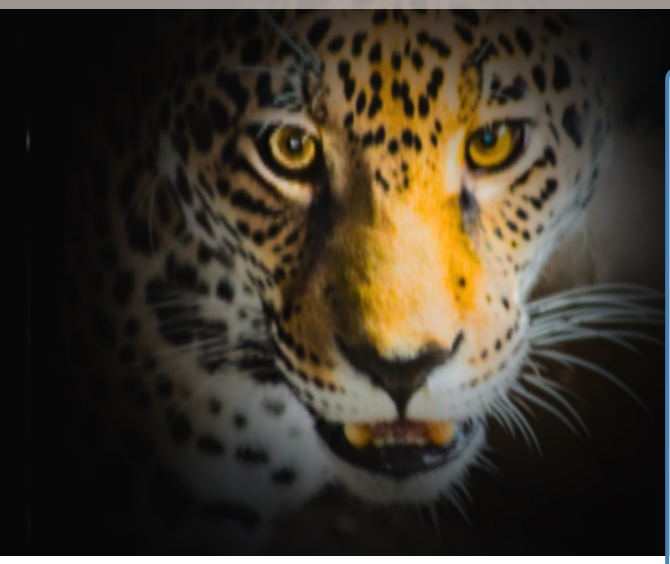
```
singleSorted :: Sortedness a
singleSorted x = Sorted x x
```

2. We accumulate (also called *crush*) all the monoid values with foldDC, which can be done in parallel.
3. We extra the boolean from the resulting monoid value.

```
toBool :: Sortedness a → Bool
toBool EmptySorted = True
toBool NotSorted   = False
toBool (Sorted _ _)  = True
```

That is all there is to it.

7. Equational Reasoning



The pure functional nature of Haskell enables a very simple approach to reasoning about the equivalence of programs. This technique is called *equational reasoning*. This chapter illustrates the power of this technique on a number of examples.

7.1 Proving Simple Lemmas

Equational reasoning is essentially a technique for proving lemmas about the equality of two expressions. The technique consists of proving $E_0 \equiv E_n$ by means of n *trivial* steps $E_0 \equiv E_1$, $E_1 \equiv E_2$, ... and $E_{n-1} \equiv E_n$ that immediately follow from known equalities. Because of the transitivity of equality, together these consecutive steps form a proof of the target equation.

7.2 Proof by Structural Induction

Equational reasoning is very powerful in combination with structural induction. Structural induction is a proof mechanism for properties defined over recursive datatypes, typically involving functions defined by structural induction.

7.2.1 Basic Examples

We illustrate the technique on a few basic properties of (++), which is defined by structural recursion on its first argument, which happens to be a value of the recursive datatype [a].

```
(++) :: [a] → [a] → [a]
[]       ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

A first trivial property is that [] is the neutral element of (++) on the left.

Theorem 7.1

$$[\,] +\!\!+ l \equiv l$$

This follows immediately from the definition of $(+\!\!+)$. Indeed, we establish the equation by considering the first rule in the definition of $(+\!\!+)$ as an axiom.

Less trivial to prove is that $[\,]$ is also the neutral element on the right.

Theorem 7.2

$$l +\!\!+ [\,] \equiv l$$

We prove this theorem by means of structural induction on the list l. This means that we prove the theorem independently for every possible constructor from which l can be built: either l is of the form $[\,]$, or it is of the form $(x : xs)$. In the latter case we are allowed to make use of the *induction hypothesis* $xs +\!\!+ [\,] \equiv xs$ which states that the property holds for the recursive subterm xs.

Base case: $l = [\,]$

```
  [] ++ []
≡   [[ definition of (++)  ]]
  []
```

Inductive case: $l = (x : xs)$

```
  (x:xs) ++ []
≡   [[ definition of (++)  ]]
  x:(xs ++ [])
≡   [[ induction hypothesis  ]]
  x:xs
```

The third property we prove is the associativity of $(+\!\!+)$.

Theorem 7.3

$$(l_1 +\!\!+ l_2) +\!\!+ l_3 \equiv l_1 +\!\!+ (l_2 +\!\!+ l_3)$$

We prove this property by structural induction on l_1.

Base case: $l_1 = [\,]$

```
  ([] ++ l2) ++ l3
≡   [[ definition of (++)  ]]
```

```
  l2 ++ l3
≡   [[ definition of (++)  ]]
  [] ++ (l2 ++ l3)
```

Inductive case: $l_1 = (x : xs)$

```
  ((x : xs) ++ l2) ++ l3
≡   [[ definition of (++)  ]]
  (x : (xs ++ l2)) ++ l3
≡   [[ definition of (++)  ]]
  x : ((xs ++ l2)) ++ l3)
≡   [[ induction hypothesis  ]]
  x : (xs ++ (l2 ++ l3))
≡   [[ definition of (++)  ]]
  (x : xs) ++ (l2 ++ l3)
```

7.2.2 Application: Optimization Correctness

We start from a simple function that reverses lists:

```
reverse :: [a] → [a]
reverse []       = []
reverse (x : xs) = reverse xs ++ [x]
```

This definition is known as *naive reverse* because it has an unnecessarily bad time complexity: $\mathcal{O}(n^2)$. The computation consists essentially of a nested loop: the outer loop traverses the list and builds a new list by repeatedly extending a new list by appending an element x to the back. This appending operation is itself a loop over a list.

A more efficient definition of reverse is the following:

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

This definition makes use of an auxiliary function go that takes an additional parameter acc. The function go assembles the result in this additional parameter by adding elements to the front. Because this additional parameter accumulates the result we call it an *accumulating parameter* or *accumulator* for short. This function clearly has a linear time complexity. This difference from reverse is due to adding elements to the front of a list in constant time instead of adding them to the back in linear time.

Intuitively it should be clear that reverse and reverse′ are equivalent functions. This means that they yield the same output for the same input. Formally:

Theorem 7.4

$$\text{reverse} \equiv \text{reverse}'$$

Instead of directly proving the theorem, we first prove a more general auxiliary theorem that relates go and reverse.

Theorem 7.5

$$\text{go}\ l\ \text{acc} \equiv \text{reverse}\ l +\!\!+ \text{acc}$$

We prove this theorem by structural induction on the list l.

Base case: $l = [\,]$

```
  go [] acc
≡   [[ definition of go  ]]
  acc
≡   [[ definition of (++)  ]]
  [] ++ acc
≡   [[ definition of reverse  ]]
  reverse [] ++ acc
```

Inductive case: $l = (x : xs)$

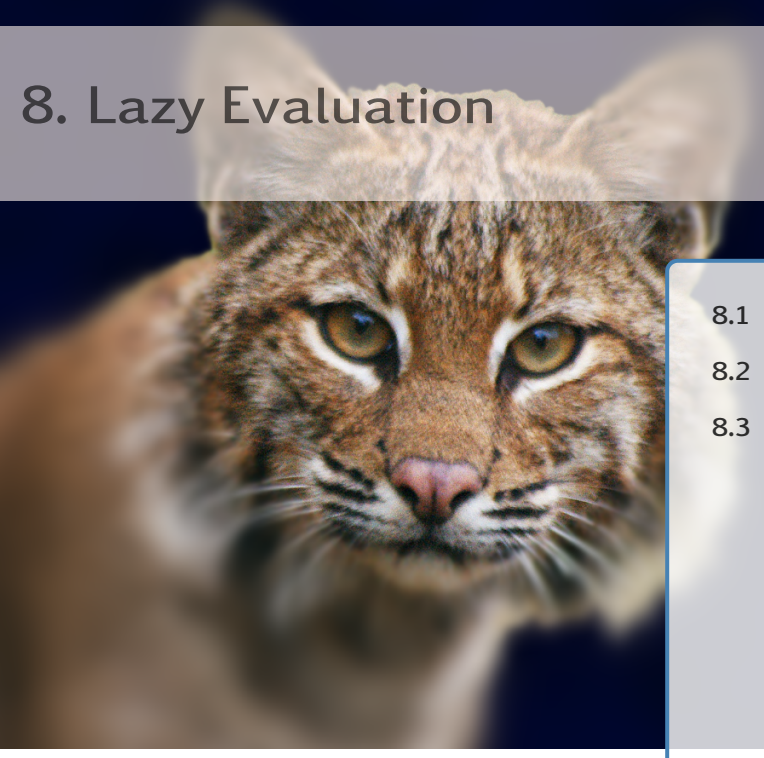
```
  go (x:xs) acc
≡   [[ definition of go  ]]
  go xs (x:acc)
≡   [[ induction hypothesis  ]]
  reverse xs ++ (x:acc)
≡   [[ definition of (++)  ]]
  reverse xs ++ ([x] ++ acc)
≡   [[ associativity of (++)  ]]
  (reverse xs ++ [x]) ++ acc
≡   [[ definition of reverse  ]]
  reverse (x:xs) ++ acc
```

Now we can easily prove the main theorem.

```
  reverse' l
≡   [[ definition of reverse'  ]]
```

```
    go l []
  ≡   [[ auxiliary theorem  ]]
    reverse l ++ []
  ≡   [[ property of (++)  ]]
    reverse l
```

8. Lazy Evaluation



This chapter discusses the lazy evaluation strategy of Haskell known as *call by need*.

8.1 Evaluation Strategies

Chapter 1 has presented the λ-calculus, which forms a foundation for Haskell. The three computation rules provide equivalence-preserving ways of transforming expressions: α-conversion, β-reduction and η-conversion. Of these three results, the one for β-reduction conforms to our intuition about evaluating programs. A β-reduction step simplifies a function call. By repeatedly applying this step until no more function call appears in the expression, we can obtain a final result.

Here is an example of a non-trivial expression that is turned into a final result by repeated application of β-reduction.

$$
\begin{aligned}
&((\lambda x.x\ 5)\ (\lambda y.(\lambda z.y)))\ 7 \\
\rightarrowtail_\beta\ &((\lambda y.(\lambda z.y))\ 5)\ 7 \\
\rightarrowtail_\beta\ &(\lambda z.5)\ 7 \\
\rightarrowtail_\beta\ &5
\end{aligned}
$$

8.1.1 Call by Value

Not every example is as straightforward as the one above. Stating that every program is evaluated by means of repeated β-reduction still leaves a degree of freedom. Namely there can be multiple reducible function applications within an expression. Then we can choose which subexpression to reduce first.

Here is a simple example with two reducible subexpressions. We choose to reduce the one on the right first.

```
  (λy.y) ((λz.z) 5)
↣β
  (λy.y) 5
↣β
  5
```

The above examples illustrates the conventional evaluation strategy, used by most programming languages. It is known as *call-by-value*, because arguments are first reduced to values before they are passed to a function.

8.1.2 Call by Name

There is an alternative reduction order in the running example, where the application on the left is reduced first:

```
  (λy.y) ((λz.z) 5)
↣β
  ((λz.z) 5)
↣β
  5
```

This strategy is known as *call-by-name*, because it passes the arguments to the function before evaluating them. Hence the arguments are passed in their original form (i.e, with their original name).

It may seem a coincidence that both call-by-value and call-by-name yield the same result in the example, but it is not. In general, if both evaluation strategies yield a value, that value is identical.

If both strategies yield the same result, then why is it important to distinguish between them?

There are two important reasons for this:

- Firstly, you have to read the "fine print": *if they yield a result*, it is the same. It may be the case though that one strategy does not yield a result, but the other one does. Here not yielding a result means that the evaluation does not terminate.

  Let us illustrate this with an example in Haskell notation. Consider the following two function definitions:

```
loop = loop
const x y = x
```

  With call-by-value the evaluation of the following expression does not terminate:

```
  const 42 loop
⟶CBV
  const 42 loop
⟶CBV
  ...
```

With call-by-name the evaluation does terminate:

```
  const 42 loop
⟶CBN
  42
```

We see that call-by-value always evaluates the arguments, even if they are not needed to obtain the final result. In contrast, call-by-name only evaluates the arguments if and when they are actually needed to obtain the final result.

This observation can be generalized: the termination behavior of call-by-name is better than that of call-by-value. Every expression that terminates with call-by-value, also terminates with call-by-name, but not vice versa.

- Secondly, even if the two strategies obtain the same result, they may not necessarily do so with the same time and space characteristics. We illustrate the difference in runtime with `const 42 (sum [1..1000])`. With call-by-name the result `42` is obtained after 1 reduction step, while call-by-value first needs about 1000 reduction steps to compute the sum.

  At first sight call-by-name always needs fewer reduction steps than call-by-value, but that is not the case. This perhaps unexpected weakness of call-by-name is mitigated by call-by-need.

8.1.3 Call by Need

Call-by-name has the annoying performance problem of sometimes doubling the amount of work. Here we count three reduction steps for call-by-value:

```
  (λx → x + x) (1 + 2)
⟶CBV
  (λx → x + x) 3
⟶CBV
  3 + 3
⟶CBV
  6
```

In contrast call-by-name requires four steps:

$$
\begin{aligned}
&(\lambda x \to x + x)\ (1 + 2)\\
&\rightarrowtail_{CBN}\\
&(1 + 2) + (1 + 2)\\
&\rightarrowtail_{CBN}\\
&3 + (1 + 2)\\
&\rightarrowtail_{CBN}\\
&3 + 3\\
&\rightarrowtail_{CBN}\\
&6
\end{aligned}
$$

We can arbitrarily increase the number of steps with the example $(\lambda x \to x + x)\ (\mathsf{sum}\ [1..n])$. What is the source of the problem? The function $\lambda x \to x + x$ doubles the work of the argument x, because x appears twice in the body of the function. That is why call-by-name copies the argument expression and subsequently has to reduce both copies.

Call-by-need solves this problem of call-by-value by remembering the connection between the two copies. Instead of duplicating the work, it is *shared*. If one copy is evaluated, the other copy is immediately replaced by the result. This is illustrated thus:

$$
\begin{aligned}
&(\lambda x \to x + x)\ (1 + 2)\\
&\rightarrowtail_{need}\\
&(1 + 2) + (1 + 2)\\
&\rightarrowtail_{need}\\
&3 + 3\\
&\rightarrowtail_{need}\\
&6
\end{aligned}
$$

As you can see, call-by-need only needs three reduction steps. What is not shown in this textual representation is that call-by-need remembers that the two occurrences of $1 + 2$ are copies of each other. In the second step the left occurrence is evaluated to 3, and the right occurrence is immediately replaced by the same result.

In summary call-by-need has the same good termination behavior as call-by-name, better than call-by-value. Besides that call-by-need evaluates every expression at most once. In contrast, call-by-value evaluates every expression exactly once, and call-by-name may do this an arbitrary number of times.

8.2 Lazy Evaluation in Practice

8.2.1 Control Abstractions

An interesting aspect of lazy evaluation is that Haskell makes it easy to define control abstractions, like `if-then-else`.

```
ifthenelse :: Bool → a → a → a
ifthenelse True  e1 e2 = e1
ifthenelse False e1 e2 = e2
```

This definition behaves in the same way as the built-in `if then else`:

```
► (λx → ifthenelse (x > 0) (10 / x) 0) 0
0
```

If we would evaluate the same example with call-by-value, `ifthenelse` would not behave as expected.

8.2.2 Infinite Data Structures

Infinite data structers are a remarkable application of lazy evaluation.

Infinite Data Structures The most important example of infinite data structures are infinite lists. This example shows an infinite list of natural numbers:

```
► [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,
23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,
43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,
63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,
83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100,101,
102,103,104,105,106,107,108,109,110,111,112,...
```

If we had not cut off the output here, GHCi would have gone on forever.
We can implement the above example ourselves:

```
from :: Int → [Int]
from n = n : from (n + 1)
```

```
► from 1
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,
23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,
43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,
63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,
83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100,101,
102,103,104,105,106,107,108,109,110,111,112,...
```

On its own an infinite list looks like an oddity. No practical program can actually process an infinite amount of data. Every terminating program can only consider a finite part of such infinite amount of data. Yet, thanks to lazy evaluation it is practical to do so with an infinite list. For example:

```
► take 10 [1..]
[1,2,3,4,5,6,7,8,9,10]
```

We take the first ten elements of the infinite list. Because the remainder of the list is not needed for the final result, it is not evaluated. In this way lazy evaluation makes a seemingly non-terminating program terminate.

Compositionality What makes infinite lists interesting is the purity of their definition. For example:

```
nats :: [Int]
nats = [1..]
```

defines the natural numbers in a very compact way, without considering termination. Of course every practical terminating program can only consider a finite subset of the natural numbers, but nats need not be concerned with the logic for selecting a particular subset. This logic can be defined externally to nats. In this way we can work in a highly compositional fashion: 1) we can apply various kinds of selection to nats, and 2) we can apply the same selection to various kinds of infinite lists.

```
► take 5 nats
[1,2,3,4,5]
► takeWhile (<7) nats
[1,2,3,4,5,6]
```

Moreover, we can define new infinite lists in terms of existing ones.
The even numbers are:

```
evens :: [Int]
evens = map (2*) nats
```

```
► take 10 evens
[2,4,6,8,10,12,14,16,18,20]
```

An alternative definition for the even numbers is:

```
evens' :: [Int]
evens' = zipWith (*) nats (repeat 2)
```

where

```
repeat :: a → [a]
repeat x = x : repeat x
zipWith :: (a → b → c) → [a] → [b] → [c]
zipWith f [ ]      _      = [ ]
zipWith f _      [ ]      = [ ]
zipWith f (x : xs) (y : ys) = f x y : zipWith f xs ys
```

The factorials are:

```
facs :: [Int]
facs = scanl1 (*) nats
```

where

```
scanl1 :: (a → a → a) → [a] → [a]
scanl1 f [ ] = [ ]
scanl1 f (x : xs) = go xs x
  where go [ ]      acc = [acc]
        go (x : xs) acc = acc : go xs (f acc x)
```

```
► take 10 facs
[1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

Dataflow Dependencies An astounding alternative definition of nats is:

```
nats' :: [Int]
nats' = 1 : map (1 +) nats'
```

This infinite list is defined in terms of itself. This is seemingly paradoxical: to define the natural numbers, we already have to have a definition of the natural numbers. Fortunately, there is not actually a paradox. Indeed, the definition defines the first number in the list directly with self-dependency. Based on this first number, we can determine the second number, and so on:

```
nats' ≡ 1 : map (1 +) nats'
      ≡ 1 : map (1 +) (1 : ???)
      ≡ 1 : 1 + 1 : map (1 +) (1 + 1 : ???)
      ≡ 1 : 2 : map (1 +) (2 : ???)
```

In fact every element depends on the previous one, and the first element is given. Hence, there is an evaluation order in which all the elements can be effectively computed, and lazy evaluation dynamically figures out this evaluation order.

The powers of 2 constitute a similar example, where every number is double the previous one:

```
pows2 :: [Int]
pows2 = 1 : map (2*) pows2
```

One more example are the Fibonacci numbers, where every number is the sum of the two previous ones:

```
fibs :: [Int]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

8.3 Discussion

Even though call-by-need has multiple interesting applications and properties, it is not the most widely preferred evaluation strategy. There are actually a number of disadvantages:

- There is more overhead associated with the implementation of call-by-need (the management of so-called *thunks*), than with call-by-value. This means that programs where call-by-value and call-by-need use the same number of evaluation steps, call-by-value is typically faster.

  The overhead is partially mitigated by optimized compilation: *strictness analysis* determines when call-by-value can be used.

- It is harder for the programmer to reason about evaluation characteristics (space and time) of call-by-need than call-by-value.

- The combination of call-by-need with I/O and other side effects is quite complex, as we will see later.

9. Monads

This chapter introduces a powerful functional programming abstraction for dealing with side effects.

9.1 Kinds and Type Constructors

We use types to classify values. The values True and False are of type Bool, the value () is of type (), 'a' and 'b' are of type Char, and so on.

Yet some types, like Maybe, are not inhabited by values. To be clear, there are of course values of type Maybe (), namely

$$\{\text{Nothing}, \text{Just } ()\}$$

or of type Maybe Bool, namely

$$\{\text{Nothing}, \text{Just True}, \text{Just False}\}$$

but there are not values of type Maybe. The type has to be completed with another type in order to be inhabited by some values.

With *kinds* we can classify "proper" types, who denote a set of values, and improper or incomplete types who do not represent any values. Kinds κ have an inductive structure that is much simpler than that of types. There are only two syntactic forms:

- Kind $*$ is the base case. It is the kind of all proper types. We say for instance that Bool has kind $*$ and write this as $\text{Bool} :: *$.
- The inductive case is kind $\kappa_1 \to \kappa_2$, where κ_1 and κ_2 are themselves kinds. This is the kind of improper types. These types are also called *type constructors*, in analogy with data constructors, because they need to be applied to other types to obtain a proper type. A type constructor of kind $\kappa_1 \to \kappa_2$ takes a parameter of kind κ_1 and yields a type of kind κ_2.

  In the simplest case κ_1 and κ_2 are just the kind $*$. An example of this situation is $\text{Maybe} :: * \to *$. When we apply Maybe to Bool of kind $*$ we obtain Maybe Bool of kind $*$. Another example of a type constructor is that of lists $[\,] :: * \to *$.

A type constructor with a more complex kind is the predefined type Either, which has two type parameters. Recall:

```
data Either a b = Left a | Right b
```

The kind of Either is $* \to (* \to *)$ or simply $* \to * \to *$.

9.2 Functor

We have used type classes to provide a common interface to proper types that support a similar set of operations. Now we can do the same for type constructors.

An important class of type constructors with similar operations are generic "collections" of kind $* \to *$. These are data structures like lists and trees that collect values of a given type. Because not all data structures support all possible kinds of operations on collections, we use different type classes for different operations.

9.2.1 Generalized map

We focus here on a well-studied operation, that is both very abstract and very concrete: the map function of lists. Collections that support this operation are called *functors* and in general we call the operation fmap:

```
class Functor f where
  fmap :: (a → b) → (f a → f b)
```

The best known instance is that for lists:

```
instance Functor [ ] where
  fmap f l = map f l
```

9.2.2 The Functor Laws

Not any implementation of fmap will do for other types of collections f. Indeed, we expect that implementations of fmap respect our intuition and properly generalize the map function for lists. This expectation is captured in two *laws*:

1. Firstly, we expect that fmap preserves the shape of the collection and only modifies the elements individually with the given function f. This property is partially captured in the *identity law*:

   $$\text{fmap id} \equiv \text{id}$$

   This law expresses that if we do not modify the individual elements ($f \equiv \text{id}$), then fmap id does not modify the collection as a whole.

2. The second law is the *fusion law*:

$$\mathsf{fmap}\ f \circ \mathsf{fmap}\ g \equiv \mathsf{fmap}\ (f \circ g)$$

   The law expresses that two successive applications of fmap can be merged (or fused) into a single one. This is a form of optimisation in which two loops are replaced by a single one.

Lawfulness of Lists Let us check whether the list instance is lawful and abides by the two laws. As a reminder, here is the definition of map:

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

We prove the first Functor law by means of structural induction on the list.

Base case: $l \equiv [\,]$

```
  map id []
≡   [[ def. of map   ]]
  []
≡   [[ def. of id    ]]
  id []
```

Induction case: $l \equiv (x : xs)$

```
  map id (x : xs)
≡   [[ def. of map   ]]
  id x : map id xs
≡   [[ induction hypothesis   ]]
  id x : id xs
≡   [[ def. of id    ]]
  x : xs
≡   [[ def. of id    ]]
  id (x : xs)
```

We prove the second law in a similar way:

Base case: $l \equiv [\,]$

```
  map f (map g [])
≡   [[ def. of map   ]]
  map f []
```

```
    ≡   [[ def. of map   ]]
      [ ]
    ≡   [[ def. of map   ]]
      map (f ∘ g) [ ]
```

Induction case: $l \equiv (x : xs)$

```
      map f (map g (x : xs))
    ≡   [[ def. of map   ]]
      map f (g x : map g xs)
    ≡   [[ def. of map   ]]
      f (g x) : map f (map g xs)
    ≡   [[ def. of (∘)   ]]
      (f ∘ g) x : map f (map g xs)
    ≡   [[ induction hypothesis   ]]
      (f ∘ g) x : map (f ∘ g) xs
    ≡   [[ def. of map   ]]
      map (f ∘ g) (x : xs)
```

In conclusion, the list instance is indeed lawful. Seen from another perspective, the two Functor laws capture two essential properties of lists.

9.2.3 Example Instances of Functor

Just like lists, trees also have a lawful Functor instance:

```
data Tree a = Leaf a | Fork (Tree a) (Tree a)
instance Functor Tree  where
  fmap f (Leaf x)  = Leaf (f x)
  fmap f (Fork l r) = Fork (fmap f l) (fmap f r)
```

Another instance of Functor is that of Maybe, which can be seen as a tiny collection with either 0 or 1 element:

```
instance Functor Maybe  where
  fmap f Nothing = Nothing
  fmap f (Just x)  = Just (f x)
```

Two more esoteric minimal Functor instances are that for the singleton collection I and the empty collection Empty.

```
data I a        = I a
data Empty a = Empty
instance Functor I  where
  fmap f (I x) = I (f x)
instance Functor Empty  where
  fmap f Empty = Empty
```

Exercise 9.1 Prove that the above four instances are lawful.

9.3 Monads as Collections

The Monad concept and corresponding Haskell type class build on the notion of a Functor.

There are two ways to think concretely about the abstract notion of a monad. The first one is the most simple. It views monads as collections that support two operations in addition to that of Functor. Unfortunately, because this is not the main view on monads in Haskell, it is not reflected in the standard Monad type class. Hence, for the sake of studying the collections view, we define our own type class Monad† and come back to Haskell's Monad type class when we consider the second view on monads.

So, without further ado, here is the Monad† type class:

```
class Functor c ⇒ Monad† c  where
  unit :: a → c a
  join :: c (c a) → c a
```

Note that every Monad† instance must also have a Functor instance with an accompanying implementation for fmap. This is captured in the Functor super class constraint on the Monad† declaration. In addition to the fmap implementation, Monad† instances supply implementations for two more methods.

1. The unit :: a → c a method creates a singleton collection of type c a from the given element of type a.

2. The join :: c (c a) → c a method collapses the two levels of a nested collection of type c (c a) to obtain a flat collection of type c a.

To make the above abstract descriptions more concrete let us consider the instance for the list type:

```
instance Monad† [ ]  where
  unit x    = [x]
  join xss = concat xss
```

where the Prelude function concat is predefined as

```
concat :: [[a]] → [a]
concat [ ]       = [ ]
concat (xs : xss) = xs ++ concat xss
```

As we can see, unit x creates the singleton list [x]. Also join [[1,2],[3],[ ],[4,5,6]] collapses the nested list into the flat list [1,2,3,4,5,6].

Example Instances Here are a few more example instances for the types introduced in Section 9.2.3.

```
instance Monad† Tree where
  unit x = Leaf x
  join (Leaf t)   = t
  join (Fork l r) = Fork (join l) (join r)
instance Monad† Maybe where
  unit x = Just x
  join Nothing = Nothing
  join (Just m) = m
```

Verify that these instances implement the two methods appropriately.

Exercise 9.2 Provide instances for the esoteric types I and Empty of Section 9.2.3.

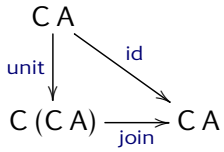
9.3.1 Lawful Instances

The Monad† type class comes with four laws that all instances must satisfy.

1. The first law regulates the interaction between join and unit.

$$\mathsf{join} \circ \mathsf{unit} \equiv \mathsf{id}$$

In point-free style this law can be rather cryptic. To give more insight in what the law expresses, we present it in a different form known as a *commuting diagram*.



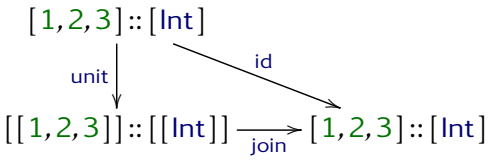
Every node in the diagram is a type and every arrow denotes a function from one type to another type. Hence, a path in the diagram represents the function that is the composition of the functions denoted by the initial arrows.

We say that the diagram commutes because all paths with the same start and end points are equivalent. In this case join ∘ unit and id are equivalent, as already expressed by

the textual law. What makes the commuting diagram more informative (and more attractive) than the textual law is the fact that it makes the involved types explicit.

In this case the diagram makes it more apparent that if we first wrap a singleton collection around an existing collection with unit and then flatten the resulting nested collection with join we end up with the original collection.

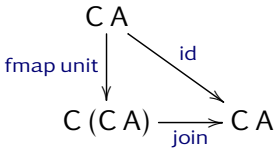
In order to illustrate the law, we show a special form of the commuting diagram where we replace the types with example values.



2. The second law also considers a different interaction between join and unit.

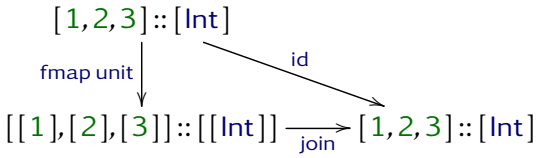
$$\mathsf{join} \circ \mathsf{fmap}\ \mathsf{unit} \equiv \mathsf{id}$$

The corresponding commuting diagram is:



This law is very similar to the first one. The essential difference is that we now do not create a singleton layer around the original collection with unit, but around the individual elements inside the collection with fmap unit. Flattening with join again yields the original collection.

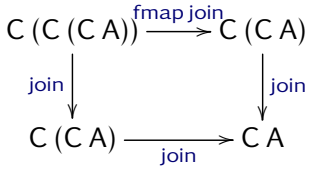
Here is a concrete example.



3. The third law considers the interaction of two joins.

$$\mathsf{join} \circ \mathsf{join} \equiv \mathsf{join} \circ \mathsf{fmap}\ \mathsf{join}$$

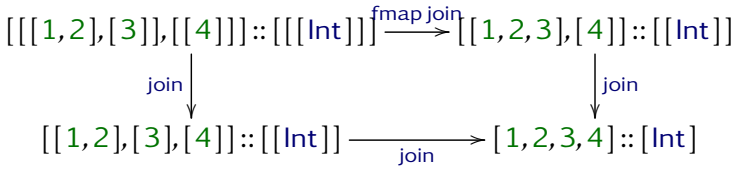
The corresponding commuting diagram is:



Both paths in the diagram turn a collection with 3 levels of nesting into a flat collection by successively collapsing 2 levels. The path on the left first collapses the two outer

levels, while the path on the right first collapses the two inner levels. The law states that it does not matter which one you pick.

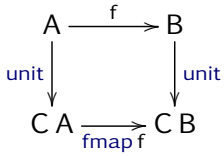
Here is an example.



4. The final law governs the interaction between fmap and unit.

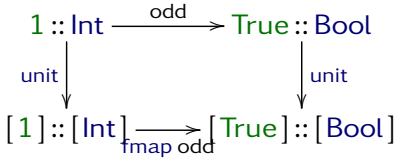
$$\mathsf{fmap}\ f \circ \mathsf{unit} \equiv \mathsf{unit} \circ f$$

where $f :: A \to B$ can be any function. The corresponding commuting diagram is:



The law states that we can choose whether we first transform a value with f and then wrap it in a singleton collection, or first wrap the value in a singleton collection and the transform it with fmap f.

This is an example.



Exercise 9.3 Prove that the Maybe instance is lawful.

Exercise 9.4 Prove that the list instance is lawful.

Exercise 9.5 Prove that the Tree instance is lawful.

Exercise 9.6 Draw the commuting diagrams for the two Functor laws.

9.4 Monads for Computations with Side Effects

The main use of monads in Haskell is for modelling computations with side-effects.

9.4.1 Motivating Example

Consider a datatype Exp for representing simple arithmetic expressions.

```
data Exp = Lit Int | Add Exp Exp
```

The Lit constructor creates simple integer literals (or constants), and the Add constructor denotes the addition of two subexpressions. For instance, Add (Lit 1) (Lit 2) denotes the expression 1 + 2.

Of course Exp only captures the syntax of arithmetic expressions. We capture their semantics, i.e., the evaluation of an expression tree, in a function:

```
eval :: Exp → Int
eval (Lit n)      = n
eval (Add e1 e2) = eval e1 + eval e2
```

Note that eval is what we call a *pure* function: it is a function in the mathematical sense that maps every input expression tree onto an integer.

```
► eval (Add (Lit 1) (Lit 2))
3
```

Expressions with Division To spice things up, we extend the expression syntax with a constructor Div for division.

```
data Exp = Lit Int | Add Exp Exp | Div Exp Exp
```

This obviously requires an extended evaluation function.

```
eval :: Exp → Int
eval (Lit n)      = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Div e1 e2)  = eval e1 `div` eval e2
```

Unfortunately, this definition is no longer a pure function.

```
► eval (Div (Lit 1) (Lit 0))
*** Exception: divide by zero
```

It raises an exception upon division by zero and hence does not return a result for every possible expression. This makes eval a *partial* function: its result is not defined for every value in its domain.

While this exception mechanism is common to many programming languages, it is considered a violation of the purely functional programming philosophy where we want to stay as close as possible to the mathematical ideal of a total function that is defined for all values of its domain.

Explicit Partiality Fortunately, instead of relying on the impure exception mechanism, we can model the partiality of eval in a purely functional way. The key is to be explicit about the partiality in the return type. For this purpose we use the Maybe type constructor.

```
eval :: Exp → Maybe Int
```

It signals that eval returns either Just a value or Nothing. Now eval can be defined as a total function that explicitly returns Nothing when there is no sensible result.

```
eval (Lit n)      = Just n
eval (Add e1 e2) = case eval e1 of
                     Just n1 → case eval e2 of
                                 Just n2 → Just (n1 + n2)
                                 Nothing → Nothing
                     Nothing → Nothing
eval (Div e1 e2)  = case eval e1 of
                     Just n1 → case eval e2 of
                                 Just n2 → if (n2 ≡ 0)
                                             then Nothing
                                             else Just (n1 `div` n2)
                                 Nothing → Nothing
                     Nothing → Nothing
```

Observe in the above definition how the Maybe result type forces us to explicitly anticipate the potential failure at all (recursive) invocations of eval. This is a good thing because it forces us to pause and think about how we want to handle failure.

The downside is that the explicit handling of partiality makes the definition of eval highly verbose. Luckily we can remedy this issue with a healthy dose of abstraction. Observe that the following pattern occurs four times in the above code.

```
case ... of
  Just x    → ... x ...
  Nothing → Nothing
```

Let us capture this pattern in a function and call it bind.

```
bind :: Maybe a → (a → Maybe b) → Maybe b
bind m f = case m of
```

```
    Just x    → f x
    Nothing → Nothing
```

This function takes two partial computations. The first, m, maybe returns a result of type a. The second, f, depends on the result of the first computation and maybe returns a result of type b. The bind function composes or *glues* these two computations together: if m produces a result, it is passed to f; if m fails, the composition fails.

The definition of eval in terms of bind is much more concise.

```
eval (Lit n)       = Just n
eval (Add e₁ e₂) = eval e₁ `bind` λn₁ →
                     eval e₂ `bind` λn₂ →
                     Just (n₁ + n₂)
eval (Div e₁ e₂)  = eval e₁ `bind` λn₁ →
                     eval e₂ `bind` λn₂ →
                     if (n₂ ≡ 0)
                       then Nothing
                       else Just (n₁ `div` n₂)
```

Note that the partiality of recursive eval calls is no longer handled explicitly; the bind function fully encapsulates this. This means that the code is not only shorter, but also expresses only the interesting aspect of the computation.

9.4.2 Generalisation

We can generalise the use of Maybe to model partiality in a purely functional fashion to using other monads for modelling other kinds of side-effects. Every time the idea is that the type m a models a computation with results of type a that can have additional, impure behavior modelled by the monad m.

Bind-Centrism The function bind plays a central role in this approach. It allows us to compose two computations with side-effects m a and a → m b where the latter depends on the former. In fact, bind m f acts as function application for computations with side-effects where m is the argument and f is the function. In contrast with regular pure function application, which is written with a space in Haskell, both the argument and the function can produce side effects. The order of the arguments of bind indicates in which order the side effects happen: first those of the argument and then those of the function.

In order to express bind, the Monad† type class we have introduced earlier is enough:

```
bind :: Monad† m ⇒ m a → (a → m b) → m b
bind m f = join (fmap f m)
```

Verify that this boils down to the earlier definition when m is instantiated to Maybe.

However, given the central role of bind for modelling side effects, Haskell does not use the Monad† type class. Instead it uses the following bind-centric interface:

```
class Monad m where
  return :: a → m a
  (≫=) :: m a → (a → m b) → m b
```

Here bind is actually written with the infix operator (≫=), but still pronounced *bind*. Also our earlier unit function has been renamed to return, which is more apt when talking about computations that immediately produce a result without generating any side effect.

Note that the Monad type class does not have Functor as an explicit super class.[1] Even so, every Monad is necessarily a Functor and we can express fmap for any Monad in terms of (≫=) and return thus:

```
fmap_M :: Monad m ⇒ (a → b) → (m a → m b)
fmap_M f m = m ≫= λx → return (f x)
```

Exercise 9.7 Verify that this definition coincides with the fmap implementation given earlier for Maybe.

Dynamism We can also express join in terms of (≫=).

```
join_M :: Monad m ⇒ m (m a) → m a
join_M m = m ≫= λn → n
```

This definition highlights the dynamic character of monadic computations. A computation of type m (m a) is one that produces another computation as a result. With join we can run this dynamically produced computation immediately after it has been produced. Hence, the plan for a dynamic computation does not have to be known in advance, it can be made up along the way.

The Monad Laws We have already expressed two Functor laws and four Monad† laws. We can reformulate these in terms of three concise Monad laws.

1. The *left unit* law states that return is the unit of (≫=) on the left.

   $$\texttt{return}\ x \gg\!= f \equiv f\ x$$

   This is obvious since no side effects happen in return.

2. For similar reasons the *right unit* law states that return is also the unit of (≫=) on the right.

   $$m \gg\!= \texttt{return} \equiv m$$

[1] In recent versions of Haskell, this situation has been remedied. In fact, the remedy also involves a type class Applicative that provides a more expressive interface than Functor but less expressive than Monad.

3. The *associativity* law states that $(\gg=)$ is associative.

$$(m \gg= f) \gg= g \equiv m \gg= (\lambda x \to f\ x \gg= g)$$

Since the nesting does not matter, we often omit the parentheses and write simply $m \gg= f \gg= g$.

Exercise 9.8 Prove the Monad laws in terms of the Monad† and Functor laws, and the definition of $(\gg=)$ in terms of fmap and join.

Exercise 9.9 Prove the Functor and Monad† laws in terms of the Monad laws, and the definitions of fmap and join in terms of return and $(\gg=)$.

The do Notation In order to facilitate the use of monadic computations, Haskell provides syntactic sugar that makes the use of $(\gg=)$ less cumbersome. This syntactic sugar allows writing a pipeline of $(\gg=)$ applications in the form of a do block:

$$m \gg= \lambda x_1 \to f_1\ x_1 \gg= \lambda x_2 \to \cdots \gg= \lambda x_n \to f_n\ x_1 \ldots x_n \equiv$$

```
do x1 ← m
   x2 ← f1 x1
   ...
   xn ← ...
   fn x1 ... xn
```

Every step of the pipeline is written on its own line. If a step produces a result that is used in later lines, it is given a name with the left arrow notation $x \leftarrow$.

To illustrate the do notation, we rewrite the definition of the monadic eval function in terms of it.

```
eval :: Exp → Maybe Int
eval (Lit n)      = return n
eval (Add e1 e2) = do n1 ← eval e1
                      n2 ← eval e2
                      return (n1 + n2)
eval (Div e1 e2)  = do n1 ← eval e1
                      n2 ← eval e2
                      if (n2 ≡ 0)
                        then Nothing
                        else return (n1 `div` n2)
```

Here are two important caveats for novice users of the do notation:

1. The syntax is sensitive to indentation: all rules of a do block have to be neatly aligned and start in the same column. If a line is indented further than its predecessor, the compiler generates a syntax error of the form `parse error on input `<-'`.

2. The last line in a **do** block can never be of the form x ← m because there are no further lines that could refer to x. If you make a mistake against this rule, the compiler treats you to the error message `The last statement in a 'do' block must be an expression`.

From now on we often use the **do** notation when writing monadic code.

The (≫) Operator The predefined (≫) operator is a special form of (≫=).

```
(≫) :: m a → m b → m b
ma ≫ mb = ma ≫= λ_ → mb
```

The monadic computation m ≫ n sequentially composes m and n without passing the result of m to n. Instead the intermediate result is simply discarded. Hence the only reason to execute m is for its side effects.

A good use of (≫) is in combination with the guard function.

```
guard :: Bool → Maybe ()
guard b =
  if b then return ()
       else Nothing
```

This function returns an uninteresting result of type (), but has the interesting side effect of failing when the given boolean is False.

We can use this as follows in our running example.

```
eval (Div e1 e2) = do n1 ← eval e1
                      n2 ← eval e2
                      guard (n2 ≠ 0) ≫ return (n1 `div` n2)
```

Moreover, we do not have to mention the (≫) operator explicitly. It is used implicitly by the **do** notation whenever we do not name the result of a step.

```
eval (Div e1 e2) = do n1 ← eval e1
                      n2 ← eval e2
                      guard (n2 ≠ 0)
                      return (n1 `div` n2)
```

9.4.3 The State Monad

We now present the State monad which models the *mutable state* side effect.

List of Random Numbers Consider the problem of generating a list of random numbers in Haskell. For this purpose we can use the RandomGen type class for random number generators.

```
class RandomGen g where
  next :: g → (Int, g)
    -- other methods omitted
```

In fact, this type class provides an interface for pseudo-random number generators, as actual randomness cannot be obtained in a purely functional language or even a fully deterministic computer. The idea is to approximate true randomness by pseudo-randomness, which uses a sufficiently complex deterministic algorithm that only seems to generate random results.

The idea is that the generator has a state, in the above interface represented by a value of type g. The next function produces, based on some algorithm, from that state a pseudo-random number and a new state. Given the same initial state, the next function always generates the same number and next state; there is no actual randomness involved, only purely functional behavior. The way to obtain multiple (typically different) pseudo-random numbers, is to call next on the second state, third state, and so on.

We capture this approach in the randomList function that produces a list of n random numbers.

```
randomList :: RandomGen g ⇒ Int → g → ([Int], g)
randomList n g0
  | n > 0
  = let (x, g1) = next g0
        (xs, g2) = randomList (n − 1) g1
    in (x : xs, g2)
  | otherwise
  = ([ ], g0)
```

Observe that the explicit threading of the generator state is rather clumsy. It not only clutters the definition of randomList, but also makes it easy to introduce logical errors. For instance, if we mistakenly write the recursive call to randomList as randomList (n − 1) g_0, i.e., mentioning the initial generator state, we always obtain a list of n identical numbers.

Threading State with Bind We can clean up the verbose threading of state by encapsulating it in a function. As a starting point, we introduce a type synonym for a state-passing computation.

```
type SP s a = s → (a, s)
```

In words, a state-passing computation is one that takes an input state of type s and produces an output value of type a alongside an output state of type s. The bind_{SP} function

glues together two state-passing computations, whether the latter depends on the result of the former.

```
bindSP :: SP s a → (a → SP s b) → SP s b
bindSP m f = λs0 → let (x, s1) = m s0
                   in  f x s1
```

Observe how $\mathsf{bind}_{\mathsf{SP}}$ threads the state through both computations.

We can also represent a pure computation (i.e., one that is independent of the state) as a state-passing computation.

```
pureSP :: a → SP s a
pureSP x = λs → (x, s)
```

The two building blocks for state-passing computations nicely unclutter our definition of randomList.

```
randomList :: RandomGen g ⇒ Int → SP g [Int]
randomList n
  | n > 0
  = next                    `bindSP` λx →
    randomList (n – 1) `bindSP` λxs →
    pureSP (x : xs)
  | otherwise
  = pureSP [ ]
```

The State Monad Clearly, state passing is a computational effect that forms a monad. In order to make it a proper instance of the monad type class we do need a type constructor m of kind $* \to *$ that can be applied to the result type $a :: *$ of the computation. The type $s \to (a, s)$ obviously cannot be split up into m a for some m. Also the type synonym SP s a cannot be used, because[2] the Haskell language does not allow type synonyms to be split; SP s is not a valid type in Haskell.

The solution to this conundrum is to define a new type in Haskell.

```
data State s a = State (SP s a)
```

Because this is a new type with the a parameter conveniently in the last position, it is perfectly valid to mention the type constructor State $s :: * \to *$ separately. At last we have a suitable type constructor to instantiate the Monad type class.

Note that State is just a simple wrapper around $s \to (a, s)$. Hence our instance basically replicates the earlier definitions, but adds a little bit of noise converting between the State type and the underlying representation.

[2] for technical reasons related to how type checking works

```
instance Monad (State s) where
  return x = State (pure_SP x)
  m >>= f  = State (bind_SP (runState m) (runState ∘ f))
runState :: State s a → SP s a
runState (State m) = m
```

This slightly affects the definition of randomList.

```
randomList :: RandomGen g ⇒ Int → State g [Int]
randomList n
  | n > 0
  = do x  ← State next
       xs ← randomList (n − 1)
       return (x : xs)
  | otherwise
  = return [ ]
```

Exercise 9.10 Prove that the Monad instance is lawful.

Getting and Putting The acts of reading and writing the state are usually encapsulated in two functions.

```
get :: State s s
get = State (λs → (s, s))
put :: s → State s ()
put s' = State (λs → ((), s'))
```

These allow us to avoid explicitly mentioning the State constructor in our running example.

```
randomList :: RandomGen g ⇒ Int → State g [Int]
randomList n
  | n > 0
  = do  x  ← next_S
        xs ← randomList (n − 1)
        return (x : xs)
  | otherwise
  = return [ ]
  where
    next_S :: RandomGen g ⇒ State g Int
    next_S =
      do g ← get
```

```
        let (x, g') = next g
        put g'
        return x
```

Avoiding the State constructor is a good thing because it avoids reliance on the actual representation of the state monad. However, it is also bad because the definition of $next_S$ is rather verbose. For this reason, a third primitive is often introduced to modify the current state.

```
modify :: (s → (a, s)) → State s a
modify f = State f
```

This leads us to our final definition of randomList.

```
randomList :: RandomGen g ⇒ Int → State g [Int]
randomList n
  | n > 0
  = do x  ← modify next
       xs ← randomList (n – 1)
       return (x : xs)
  | otherwise
  = return [ ]
```