Domain-Specific Languages of Mathematics

Patrik Jansson Cezar Ionescu Jean-Philippe Bernardy

WORK IN PROGRESS: DRAFT OF November 7, 2020

The main idea behind this textbook is to encourage the students to approach mathematical domains from a functional programming perspective: to identify the main functions and types involved and, when necessary, to introduce new abstractions; to give calculational proofs; to pay attention to the syntax of the mathematical expressions; and, finally, to organize the resulting functions and types in domain-specific languages.

Contents

0 About this book **8**

0.1 Introduction 8

0.2 About this book 11

0.3 Who should read this textbook? 12

0.4 Notation and code convention 13

0.5 Common pitfalls with traditional mathematical notation 14

0.5.1 A function or the value at a point? 14

0.5.2 Scoping 14

1 Types, Functions, and Complex numbers **16**

1.1 Types of **data** and functions 17

1.1.1 What is a type? 17

1.1.2 Functions 17

1.1.3 Types in Haskell: **type**, **newtype**, and **data** 21

1.2 Notation and abstract syntax for sequences 23

1.3 A DSL of complex numbers 25

1.3.1 A syntax for (complex) arithmetical expressions 30

1.4 Laws, properties and testing 32

1.4.1 Generalising laws 34

1.5 Exercises: Haskell, DSLs and complex numbers 36

2 Logic and calculational proofs **42**

2.1 An aside: Pure set theory 42

2.2 Propositional Calculus 43

2.2.1 An Evaluator for *Prop* 44

2.2.2 Truth tables and tautologies 45

2.2.3 Proofs for Propositional Logic 46

2.2.4 Implication, hypothetical derivations, contexts 48

2.2.5 The Haskell type-checker as a proof checker 50

2.2.6 Intuitionistic Propositional Logic and Simply Typed Lambda-Calculus, Curry-Howard isomorphism. 51

2.2.7 *Or* is the dual of *And*. 54

2.3 First Order Logic 55

2.3.1 Evaluator for Formulas and ∗Undecidability 56

2.3.2 Universal quantification 57

2.3.3 Existential quantification 59

2.3.4 Typed quantification 60

2.3.5 Curry-Howard for quantification over individuals 60

2.4 Examples 61

2.4.1 Proof by contradiction 61

2.4.2 Proof by cases 62

2.4.3 There is always another prime 62

2.5 Basic concepts of calculus 63

2.5.1 The limit of a sequence 66

2.5.2 Case study: The limit of a function 67

2.6 Exercises 69

2.6.1 Representations of propositions 69

2.6.2 Proofs 69

2.6.3 Continuity and limits 72

3 Types in Mathematics **75**
3.1 Types of functions, expressions and big operators 75
3.1.1 Expressions and functions of one variable 76
3.1.2 Scoping and Typing big operators 78
3.2 Detour: expressions of several variables 79
3.2.1 Partial functions 79
3.2.2 The data type of multiple variables expressions 80
3.3 Typing Mathematics: derivative of a function 82
3.4 Typing Mathematics: partial derivative 83
3.5 Type inference and understanding: Lagrangian case study . . . 85
3.6 Incremental analysis with types 89
3.7 Type classes 90
3.7.1 Numeric operations 91
3.7.2 Overloaded integer literals 92
3.7.3 Structuring DSLs around type classes 93
3.8 Computing derivatives 94
3.9 Exercises 97

4 Compositionality and Algebras **100**
4.1 Algebraic Structures 101
4.2 Homomorphisms 103
4.2.1 (Homo)morphism on one operation 103
4.2.2 Homomorphism on structures 104
4.3 Compositional semantics 106
4.3.1 Compositional functions are homomorphisms 106
4.3.2 An example of a non-compositional function 108
4.4 Folds 108
4.4.1 Even folds can be wrong! 110
4.5 Initial and Free Structures 112
4.5.1 A general initial structure 113
4.5.2 Free Structures 114

4.5.3 ∗A generic Free construction . . . 117
4.6 Computing Derivatives, reprise. . . . 118
4.7 Summary . . . 121
4.7.1 Structures and representations . . . 121
4.8 Beyond Algebras: Co-algebra and the Stream calculus . . . 122
4.9 ???? . . . 123
4.10 Exercises . . . 125

5 Polynomials and Power Series **130**
5.1 Polynomials . . . 130
5.2 Aside: division and the degree of the zero polynomial . . . 136
5.3 Polynomial degree as a homomorphism . . . 136
5.4 Power Series . . . 138
5.5 Operations on power series . . . 140
5.6 Formal derivative . . . 142
5.7 Helpers . . . 143
5.8 Exercises . . . 144

6 Higher-order Derivatives and their Applications **147**
6.1 Review . . . 147
6.2 Higher-order derivatives . . . 151
6.3 Polynomials . . . 153
6.4 Formal power series . . . 154
6.5 Simple differential equations . . . 155
6.6 Exponentials and trigonometric functions for *PowerSeries* . . . 157
6.7 Taylor series . . . 158
6.8 Associated code . . . 160
6.8.1 Not included to avoid overlapping instances . . . 161
6.8.2 This is included instead . . . 161
6.9 Exercises . . . 162

7 Elements of Linear Algebra **165**
7.1 Representing vectors as functions . . . 167
7.2 Linear transformations . . . 169
7.3 Dot products . . . 171
7.4 Examples of matrix algebra . . . 173
7.4.1 Polynomials and their derivatives . . . 173
7.4.2 ∗Dot product for functions and Fourier series . . . 174
7.4.3 Simple deterministic systems (transition systems) . . . 176
7.4.4 Non-deterministic systems . . . 179
7.4.5 Stochastic systems . . . 181
7.4.6 ∗Quantum Systems . . . 182
7.5 Monadic dynamical systems . . . 183
7.6 The monad of linear algebra . . . 184
7.7 Associated code . . . 186
7.8 Exercises . . . 189
7.8.1 Exercises from old exams . . . 189

8 Exponentials and Laplace **191**
8.1 The Exponential Function . . . 191
8.1.1 Exponential function: Associated code . . . 194
8.2 The Laplace transform . . . 195
8.3 Laplace and other transforms . . . 199
8.4 Exercises . . . 200
8.4.1 Exercises from old exams . . . 200

9 Probability Theory **202**
9.1 Sample spaces . . . 203
9.2 Bind and return . . . 206
9.3 Distributions . . . 206
9.4 Semantics of spaces . . . 207
9.5 Random Variables . . . 211
9.6 Events and probability . . . 212

9.7 Conditional probability 214
9.8 Examples 215
9.8.1 Dice problem 215
9.9 Drug test 216
9.10 Monty Hall 216
9.11 Advanced problem 217
9.12 Independent events 219
9.13 End 220

A Parameterised Complex Numbers **222**

Chapter 0

About this book

0.1 Introduction

This book started out as lecture notes aimed at covering the lectures and exercises of the BSc-level course "Domain-Specific Languages of Mathematics" (at Chalmers University of Technology and University of Gothenburg). The immediate aim of the book is to improve the mathematical education of computer scientists and the computer science education of mathematicians. We believe the book can be the starting point for far-reaching changes, leading to a restructuring of the mathematical training of engineers in particular, but perhaps also for mathematicians themselves.

Computer science, viewed as a mathematical discipline, has certain features that set it apart from mainstream mathematics. It places much more emphasis on syntax, tends to prefer formal proofs to informal ones, and views logic as a tool rather than (just) as an object of study. It has long been advocated, both by mathematicians [Wells, 1995, Kraft, 2004] and computer scientists [Gries and Schneider, 1995, Boute, 2009], that the computer science perspective could be valuable in general mathematical education. Until today, as far as we can judge, this perspective has been convincingly demonstrated (at least since the classical textbook of Gries and Schneider [1993]) only in the field of discrete mathematics. In fact, this demonstration has been so successful, that we see discrete mathematics courses being taken over by computer science departments. This is a quite unsatisfactory state of affairs, for at least two reasons.

First, any benefits of the computer science perspective remain within the computer science department and the synergy with the wider mathematical landscape is lost. The mathematics department also misses the opportunity to see more in computer science than just a provider of tools for numerical computations. Considering the increasing dependence of mathematics on software, this can be a considerable loss.

Second, computer science (and other) students are exposed to two quite different approaches to teaching mathematics. For many of them, the formal, tool-oriented style of the discrete mathematics course is easier to follow than the traditional mathematical style. Moreover, because discrete mathematics tends to be immediately useful to them, the added difficulty of continuous mathematics makes it even less palatable. As a result, their mathematical competence tends to suffer in areas such as real and complex analysis, or linear algebra.

This is a serious problem, because this lack of competence tends to infect the design of the entire curriculum.

We propose that a focus on *domain-specific languages* (DSLs) can be used to repair this unsatisfactory state of affairs. In computer science, a DSL "is a computer language specialized to a particular application domain" (Wikipedia), and building DSLs is increasingly becoming a standard industry practice. Empirical studies show that DSLs lead to fundamental increases in productivity, above alternative modelling approaches such as UML [Tolvanen, 2011]. Moreover, building DSLs also offers the opportunity for interdisciplinary activity and can assist in reaching a shared understanding of intuitive or vague notions. This is supported by our experience: an example is the work done at Chalmers in cooperation with the Potsdam Institute for Climate Impact Research in the context of Global Systems Science, Lincke et al. [2009], Ionescu and Jansson [2013a], Jaeger et al. [2013], Ionescu and Jansson [2013b], Botta et al. [2017b,a].

Thus, a course on designing and implementing DSLs can be an important addition to an engineering curriculum. Our key idea is to apply the DSL approach to a rich source of domains and applications: mathematics. Indeed, mathematics offers countless examples of DSLs: the language of group theory, say, or the language of probability theory, embedded in that of measure theory. The idea that the various branches of mathematics are in fact DSLs embedded in the "general purpose language" of set theory was (even if not expressed in these words) the driving idea of the Bourbaki project, which exerted an enormous influence on present day mathematics.

TODO: JP: forward references to chapters (and update text here to better fit actual content)

TODO: CI: what to cite: wikipedia?, Bourbaki association homepage?, first book(s)?

Hence, the topic of this book is *DSLs of Mathematics (DSLM)*. It presents classical mathematical topics in a way which builds on the experience of discrete mathematics: giving specifications of the concepts introduced, paying attention to syntax and types, and so on. For the mathematics students, the style of this book will be more formal than usual, as least from a linguistic perspective. The increased formality is justified by the need to implement (parts of) the languages. We provide a wide range of applications of the DSLs introduced, so that the new concepts can be seen "in action" as soon as possible.

TODO: JP: There seem to be a paragraph missing about the benefits for the CS student

In our view a course based on this textbook should have two major learning outcomes. First, the students should be able to design and implement a DSL in a new domain. Second, they should be able to handle new mathematical areas using the computer science perspective. (For the detailed learning outcomes, see Figure 1.)

To achieve these objectives, the course consists of a sequence of case studies in which a mathematical area is first presented (for example, a fragment of linear algebra, probability theory, interval analysis, or differential equations), followed by a careful analysis that reveals the domain elements needed to build a language for that domain. The DSL is first used informally, in order to ensure that it is sufficient to account for intended applications (for example, solving equations, or specifying a certain kind of mathematical object). It is in this step that the computer science perspective proves valuable for improving the students' understanding of the mathematical area. The DSL is then implemented in Haskell. The resulting implementation can be compared with existing ones, such as Matlab in the case of linear algebra, or R in the case of statistical computations. Finally, limitations of the DSL are assessed and the possibility for further improvements discussed.

- Knowledge and understanding
  - design and implement a DSL for a new domain
  - organize areas of mathematics in DSL terms
  - explain main concepts of elementary real and complex analysis, algebra, and linear algebra
- Skills and abilities
  - develop adequate notation for mathematical concepts
  - perform calculational proofs
  - use power series for solving differential equations
  - use Laplace transforms for solving differential equations
- Judgement and approach
  - discuss and compare different software implementations of mathematical concepts

Figure 1: Learning outcomes for this book

TODO: PJ: update the list to match contents, and perhaps include chapter references

In the first instances, the course is an elective course for the second year within programmes such as CSE[1], SE, and Math. The potential students will have all taken first-year mathematics courses, and the only prerequisite which some of them will not satisfy will be familiarity with functional programming. However, as some of the current data structures course (common to the Math and CSE programmes) shows, math students are usually able to catch up fairly quickly, and in any case we aim to keep to a restricted subset of Haskell (no "advanced" features are required).

To assess the impact in terms of increased quality of education, we planned to measure how well the students do in ulterior courses that require mathematical competence (in the case of engineering students) or software compentence (in the case of math students). For math students, we would like to measure their performance in ulterior scientific computing courses, but we have taught too few math students so far to make good statistics. But for CSE students we have measured the percentage of students who, having taken DSLM, pass the third-year courses *Transforms, signals and systems (TSS)* and *Control Theory (sv:*

[1]CSE = Computer Science & Engineering = Datateknik = D

Reglerteknik), which are current major stumbling blocks. We have compared the results with those of a control group (students who have not taken the course). The evaluation of the student results shows improvements in the pass rates and grades in later courses. This is very briefly summarised in Table 1 and more details are explained by Jansson et al. [2019].

	PASS	IN	OUT
TSS pass rate	77%	57%	36%
TSS mean grade	4.23	4.10	3.58
Control pass rate	68%	45%	40%
Control mean grade	3.91	3.88	3.35

Table 1: Pass rate and mean grade in third year courses for students who took and passed DSLsofMath and those who did not. Group sizes: PASS 34, IN 53, OUT 92 (145 in all).

The work that leads up to the current book started in 2014 with an assessment of what prerequisites we can reasonably assume and what mathematical fields the targeted students are likely to encounter in later studies. In 2015 we submitted a course plan so that the first instance of the course could start early 2016. We also surveyed similar courses being offered at other universities, but did not find any close matches. ("The Haskell road to Logic Math and Programming" by Doets and van Eijck [2004] is perhaps the closest, but it is mainly aimed at discrete mathematics.)

While preparing course materials for use within the first instance we wrote a paper [Ionescu and Jansson, 2016] about the course and presented the pedagogical ideas at several events (TFPIE'15, DSLDI'15, IFIP WG 2.1 #73 in Göteborg, LiVe4CS in Glasgow). In the following years we used the feedback from students following the standard course evaluation in order to improve and further develop the course material into complete lecture notes, and now a book.

Future work includes involving faculty from CSE and mathematics in the development of other mathematics courses with the aim to incorporate these ideas also there. A major concern will be to work together with our colleagues in the mathematics department in order to distill the essential principles that can be "back-ported" to the other mathematics courses, such as Calculus or Linear Algebra. Ideally, the mathematical areas used in DSLM will become increasingly challenging, the more the effective aspects of the computer science perspective are adopted in the first-year mathematics courses.

0.2 About this book

TODO: JP: The whole chapter could have this title. It seems that what follows should be folded seamlessly into the rest

Software engineering involves modelling very different domains (e.g., business processes, typesetting, natural language, etc.) as software systems. The

main idea of this course is that this kind of modelling is also important when tackling classical mathematics. In particular, it is useful to introduce abstract datatypes to represent mathematical objects, to specify the mathematical operations performed on these objects, to pay attention to the ambiguities of mathematical notation and understand when they express overloading, overriding, or other forms of generic programming. We shall emphasise the dividing line between syntax (what mathematical expressions look like) and semantics (what they mean). This emphasis leads us to naturally organise the software abstractions that we develop in the form of domain-specific languages, and we will see how each mathematical theory gives rise to one or more such languages, and appreciate that many important theorems establish "translations" between them.

Mathematical objects are immutable, and, as such, functional programming languages are a very good fit for describing them. We shall use Haskell as our main vehicle, but only at a basic level, and we shall introduce the elements of the language as they are needed. The mathematical topics treated have been chosen either because we expect all students to be familiar with them (for example, limits of sequences, continuous functions, derivatives) or because they can be useful in many applications (e.g., Laplace transforms, linear algebra).

In the first few years, the enrolment and results of the DSLsofMath course itself was as follows:

Year	'16	'17	'18	'19	'20
Student count	28	43	39	59	50
Pass rate (%)	68	58	89	73	68

Note that this also counts students from other programmes (mainly SE and Math) while Table 1 only deals with the CSE programme students.

0.3 Who should read this textbook?

The prerequisites of the underlying course may give a hint about what is expected of the reader. But feel free to keep going and fill in missing concepts as you go along.

TODO: JP: This was stated in different terms above, restructuring needed

The student should have successfully completed

- a course in discrete mathematics as for example Introductory Discrete Mathematics.
- 15 hec in mathematics, for example Linear Algebra and Calculus
- 15 hec in computer science, for example (Introduction to Programming or Programming with Matlab) and Object-oriented Software Development

- an additional 22.5 hec of any mathematics or computer science courses.

Informally: One full time year (60 hec) of university level study consisting of a mix of mathematics and computer science.

Working knowledge of functional programming is helpful, but it should be possible to pick up quite a bit of Haskell along the way.

TODO: JP: Reading guide as dependency graph

0.4 Notation and code convention

The book is a literate program: that is, it consist of text interspersed with code fragments. The source code of the book (including in particular all the Haskell code) is available on GitHub in the repository `https://github.com/DSLsofMath/DSLsofMath`.

TODO: citation

Our code snippets are typeset using `lhs2tex`, to hit a compromise between fidelity to the Haskell source and maximize readability from the point of view of someone used to conventional mathematical notation. For example, function composition is typically represented as a circle in mathematics texts. When typesetting, a suitable circle glyph can be obtained in various ways, depending on the typesetting system: `&#8728` in HTML, `\circ` in TEX, or by the RING OPERATOR unicode codepoint (`U+2218`), which appears ideal for the purpose. This codepoint can also be used in Haskell (recent implementations allow any sequence of codepoints from the unicode SYMBOL class). However, the Haskell Prelude uses instead the infix operator `.` (period), as a crude ASCII approximation, possibly chosen for its availability and the ease with which it can be typed. In this book, as a compromise, we use the period in our source code, but our typesetting tool renders it as a circle glyph. If, when looking at typset pages, any doubt should remain regarding to the form of the Haskell source, we urge the reader to consult the github repository.

The reader is encouraged to experiment with the examples to get a feeling for how they work. A more radical, but perhaps more instructive alternative would be to recreate all the Haskell examples from scratch.

Each chapter ends with exercises to help the reader practice the concepts just taught. Sometimes the chapter text contains short, inlined questions, like "Exercice 1.10: what does function composition do to a sequence?". In such cases there is some more explanation in the exercises section at the end of the chapter.

In several places the book contains an indented quote of a definition or paragraph from a mathematical textbook, followed by detailed analysis of that quote. The aim is to improve the reader's skills in understanding, modelling, and implementing mathematical text.

0.5 Common pitfalls with traditional mathematical notation

TODO: JP: BLOAT: this should go in intro chapter.

0.5.1 A function or the value at a point?

Mathematical texts often talk about "the function $f(x)$" when "the function f" would be more clear. Otherwise there is a risk of confusion between $f(x)$ as a function and $f(x)$ as the value you get from applying the function f to the value bound to the name x.

Examples: let $f(x) = x + 1$ and let $t = 5 * f(2)$. Then it is clear that the value of t is the constant 15. But if we let $s = 5 * f(x)$ it is not clear if s should be seen as a constant (for some fixed value x) or as a function of x.

Paying attention to types and variable scope often helps to sort out these ambiguities.

0.5.2 Scoping

The syntax and scoping rules for the integral sign are rarely explicitly mentioned, but looking at it from a software perspective can help. If we start from a simple example, like $\int_1^2 x^2 dx$, it is relatively clear: the integral sign takes two real numbers as limits and then a certain notation for a function, or expression, to be integrated. Comparing the part after the integral sign to the syntax of a function definition $f(x) = x^2$ reveals a rather odd rule: instead of *starting* with declaring the variable x, the integral syntax *ends* with the variable name, and also uses the letter "d". (There are historical explanations for this notation, and it is motivated by computation rules in the differential calculus, but we will not go there now. We are also aware that the notation $\int dx f(x)$, which emphasises the bound variable, is sometimes used, especially by physicists, but it remains the exception rather than the rule at the time of writing.) It seems like the scope of the variable "bound" by d is from the integral sign to the final dx, but does it also extend to the limits of the domain of integration? The answer is no, as we can see from a slightly extended example:

$$f(x) = x^2$$
$$g(x) = \int_x^{2x} f(x)dx \qquad\qquad = \int_x^{2x} f(y)dy$$

The variable x bound on the left is independent of the variable x "bound under the integral sign". We address this issue in detail in Section 3.1. Mathematics text books usually avoid the risk of confusion by (silently) renaming variables when needed, but we believe that this renaming is a sufficiently important operation to be more explicitly mentioned.

Acknowledgments

The support from Chalmers Quality Funding 2015 (Dnr C 2014-1712, based on Swedish Higher Education Authority evaluation results) is gratefully acknowledged. Thanks also to Roger Johansson (as Head of Programme in CSE) and Peter Ljunglöf (as Vice Head of the CSE Department for BSc and MSc education) who provided continued financial support when the national political winds changed.

Thanks to Daniel Heurlin who provided many helpful comments during his work as a student research assistant in 2017.

This work was partially supported by the projects GRACeFUL (grant agreement No 640954) and CoeGSS (grant agreement No 676547), which have received funding from the European Union's Horizon 2020 research and innovation programme.

Chapter 1

Types, Functions, and Complex numbers

In this chapter we exemplify our method by applying our method to the domain of arithmetic first, and complex numbers second, which we assume most readers will already be familiar with. However, before doing so, we introduce several central concepts in the book, as well as laying out methodological assumptions.

We will implement certain concepts in the functional programming language Haskell and the code for this lecture is placed in a module called *DSLsofMath.W01* that starts here[1]:

TODO: It would also be useful to split up these imports into those needed early (*Natural*, *Ratio*) and those (*CSem*) only needed in subsection Section 1.3 (to be introduced later).

```
{-# LANGUAGE InstanceSigs #-}
module DSLsofMath.W01 where
import DSLsofMath.CSem (ComplexSem (CS), (.+.), (.*.))
import Numeric.Natural (Natural)
import Data.Ratio (Ratio, (%))
```

These lines constitute the module header which usually starts a Haskell file. We will not go into details of the module header syntax here but the purpose is to "name" the module itself (here *DSLsofMath.W01*) and to **import** (bring into scope) definitions from other modules. As an example, the second to last line imports types for rational numbers and the infix operator $(\%)$ used to construct ratios ($1 \% 7$ is Haskell notation for $\frac{1}{7}$, etc.).

[1] As mentioned already in the introduction, the code is available on GitHub.

Figure 1.1: Humorously inappropriate type mismatch on a sign in New Cuyama, California. By I, MikeGogulski, CC BY 2.5, Wikipedia.

1.1 Types of data and functions

Dividing up the world (or problem domain) into values of different types is one of the guiding principles of this book. We will see that keeping track of types can guide the development of theories, languages, programs and proofs.

1.1.1 What is a type?

As mentioned in the introduction, we emphasise the dividing line between syntax (what mathematical expressions look like) and semantics (what they mean).

As an example we start with *type expressions* — first in mathematics and then in Haskell. To a first approximation one can think of types as sets. The type of truth values, *True* and *False*, is often called *Bool* or just $\mathbb{B}$. Thus the name (syntax) is $\mathbb{B}$ and the semantics (meaning) is the two-element set $\{False, True\}$. Similarly, we have the type $\mathbb{N}$ whose semantics is the infinite set of natural numbers $\{0, 1, 2, ...\}$. Other common types are $\mathbb{Z}$ of integers, $\mathbb{Q}$ of rationals, and $\mathbb{R}$ of real numbers.

So far the syntax is trivial — just names for certain sets — but we can also combine these, and for our purposes the most important construction is the function type.

1.1.2 Functions

For any two type expressions A and B we can form the function type $A \to B$. Its semantics is the set of (semantic) functions from the semantics of A to the

semantics of B. [2] As an example, the semantics of $\mathbb{B} \to \mathbb{B}$ is a set of four functions: $\{const\ False, id, \neg, const\ True\}$ where $\neg : \mathbb{B} \to \mathbb{B}$ is boolean negation.[3] The function type construction is very powerful, and can be used to model a wide range of concepts in mathematics (and the real world). Because function types are really important, we immediately introduce a few basic building blocks which are as useful for functions as zero and one are for numbers.

TODO: JP: In fact we don't write this for other types. Is this really necessary?

Identity function For each type A there is an *identity function* $id_A : A \to A$. In Haskell all of these functions are defined once and for all as follows:

$$id :: a \to a$$
$$id\ x = x$$

When a type variable (here a) is used in a type signature it is implicitly quantified (bound) as if preceded by "for all types a". This use of type variables is called "parametric polymorphism" and the compiler gives more help when implementing functions with such types. Another "function building block" is *const* which has two type variables and two arguments:

$$const :: a \to b \to a$$
$$const\ x\ \_ = x$$

The term "arity" is used to describe how many arguments a function has. An n-argument function has arity n. For small n special names are often used: binary means arity 2 (like $(+)$), unary means arity 1 (like *negate*) and nullary means arity 0 (like `"hi!"`).

We can also construct functions which manipulate functions. They are called *higher-order* functions and as a first example we present *flip* which "flips" the two arguments of a binary operator.

$$flip :: (a \to b \to c) \to (b \to a \to c)$$
$$flip\ op\ x\ y = op\ y\ x$$

As an example $flip\ (-)\ 5\ 10 == 10 - 5$ and $flip\ const\ x\ y == const\ y\ x == y$.

Function composition The composition of two functions f and g, written $f \circ g$ and sometimes pronounced "f after g" can be defined as follows:

$$f \circ g = \lambda x \to f\ (g\ x)$$

[2] The only way to be precise about semantics is to use a formal language, and so semantics sometimes does not seem to be that much different from syntax, as in this example. Fortunately, in the rest of this book, we will be describing domains where the semantics is gives more insight than here.

[3] Pedantic remark: Additionally in Haskell one has to deal with diverging computations. These induce very subtle difficulties which we mention later. Fortunately, in the rest of this book, we can largely ignore those subtleties.

As an exercise it is good to experiment a bit with these building blocks to see how they fit together and what types their combinations have.

The type is perhaps best illustrated by a diagram with types as nodes and functions (arrows) as directed edges:

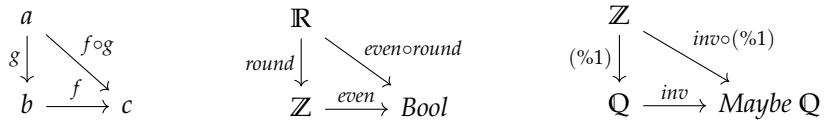


Figure 1.2: Function composition diagrams: in general, and two examples

TODO: Explain operator section somewhere: it is used as $(\%1)$ in Fig. 1.2.

In Haskell we get the following type:

$$(\circ) :: (b \to c) \to (a \to b) \to (a \to c)$$

which may take a while to get used to.

Partial and total functions There are some differences between "mathematical" functions and Haskell functions. Some Haskell "functions" are not defined for all inputs — they are *partial* functions. Simple examples include $head :: [a] \to a$ which is not defined for the empty list and $(1/) :: \mathbb{R} \to \mathbb{R}$ which is not defined for zero. A proper mathematical function is called *total*: it is defined for all its inputs, that is, it terminates and returns a value.

There are basically two ways of "fixing" a partial function: limit the type of the inputs (the domain) to avoid the "bad" inputs, or extend the type of the output (the range) to include "default" or "error" values[4]. As an example, $\sqrt{\cdot}$, the square root function, is partial if considered as a function from $\mathbb{R}$ to $\mathbb{R}$ but total if the domain is restricted to $\mathbb{R}_{\geq 0}$. In most programming languages the range is extended instead; $\sqrt{\cdot} :: Double \to Double$ where $\sqrt{-1}$ returns the "error value" *NaN* (Not a Number). Similarly, $(1/) :: Double \to Double$ returns $Infinity :: Double$ when given zero as an input. Thus *Double* is a mix of "normal" numbers and "special quantities" like *NaN* and *Infinity*.

There are also mathematical functions which cannot be implemented at all (uncomputable functions), but we will not deal with that in this course.

Pure and impure functions Many programming languages provide so called "functions" which are actually not functions at all, but rather procedures: computations depending on some hidden state or exhibiting some other effect. A typical example is $rand(N)$ which returns a random number in the range $1\,..\,N$. Treating such an "impure function" as a mathematical "pure" function quickly

[4] or better yet, meaningful values, as we shall see later

leads to confusing results. For example, we know that any pure function f will satisfy '$x == y$ implies $f(x) == f(y)$'. As a special case we certainly want $f(x) == f(x)$ for every x. But with $rand(\cdot)$ this does not hold: $rand(6) == rand(6)$ will only be true occasionally. Fortunately, in mathematics and in Haskell all functions are pure.

Variable names as type hints In mathematical texts there are often conventions about the names used for variables of certain types. Typical examples include f, g for functions, i, j, k for natural numbers or integers, x, y for real numbers and z, w for complex numbers.

The absence of explicit types in mathematical texts can sometimes lead to confusing formulations. For example, a standard text on differential equations by Edwards, Penney, and Calvis [2008] contains at page 266 the following remark:

> The differentiation operator D can be viewed as a transformation which, when applied to the function $f(t)$, yields the new function $D\{f(t)\} = f'(t)$. The Laplace transformation $\mathcal{L}$ involves the operation of integration and yields the new function $\mathcal{L}\{f(t)\} = F(s)$ of a new independent variable s.

This is meant to introduce a distinction between "operators", such as differentiation, which take functions to functions of the same type, and "transforms", such as the Laplace transform, which take functions to functions of a new type. To the logician or the computer scientist, the way of phrasing this difference in the quoted text sounds strange: surely the *name* of the independent variable does not matter; the Laplace transformation could very well return a function of the "old" variable t. We can understand that the name of the variable is used to carry semantic meaning about its type (this is also common in functional programming, for example with the conventional use of a plural "s" suffix, as in the name xs, to denote a list of values.). Moreover, by using this (implicit!) convention, it is easier to deal with cases such as that of the Hartley transform (a close relative of the Fourier transform), which does not change the type of the input function, but rather the *interpretation* of that type. We prefer to always give explicit typings rather than relying on syntactical conventions, and to use type synonyms for the case in which we have different interpretations of the same type. In the example of the Laplace transformation, this leads to

$$\mathcal{L} : (T \to \mathbb{C}) \to (S \to \mathbb{C})$$

where $T = \mathbb{R}$ and $S = \mathbb{C}$. Note that the function type constructor $(\to)$ is used three times here: once in $T \to \mathbb{C}$, once in $S \to \mathbb{C}$ and finally at the top level to indicate that the transform maps functions to functions. This means that $\mathcal{L}$ is an example of a higher-order function, and we will see many uses of this idea in this book.

TODO: JP: But we use the variable name ('s','t') as a type name. This is perhaps also confusing?

Now we move to introducing some of the ways types are defined in Haskell, our language of choice for the implementation (and often also specification) of mathematical concepts.

1.1.3 Types in Haskell: type, newtype, and data

There are three keywords in Haskell involved in naming types: **type**, **newtype**, and **data**.

type – abbreviating type expressions The **type** keyword is used to create a type synonym – just another name for a type expression. The semantics is unchanged: the set of values of type *Heltal* is exactly the same as the set of values of type *Integer*, etc.

TODO: JP: What is Heltal? Shall we say "WholeNumbers" for a less parochial approach?

$$\begin{aligned}
&\textbf{type } \mathit{Heltal} &&= \mathit{Integer}\\
&\textbf{type } \mathit{Foo} &&= (\mathit{Maybe}\ [\mathit{String}], [[\mathit{Heltal}]])\\
&\textbf{type } \mathit{BinOp} &&= \mathit{Heltal} \to \mathit{Heltal} \to \mathit{Heltal}\\
&\textbf{type } \mathit{Env}\ v\ s &&= [(v, s)]
\end{aligned}$$

The new name for the type on the right hand side (RHS) does not add type safety, just readability (if used wisely). The *Env* example shows that a type synonym can have type parameters. Note that *Env v s* is a type (for any types v and s), but *Env* itself is not a type but a *type constructor*.

newtype – more protection A simple example of the use of **newtype** in Haskell is to distinguish values which should be kept apart. A fun example of *not* keeping values apart is shown in Figure 1.1. To avoid this class of problems Haskell provides the **newtype** construct as a stronger version of **type**.

$$\begin{aligned}
&\textbf{newtype } \mathit{Population} &&= \mathit{Pop}\ \mathit{Int} && \text{-- Population count}\\
&\textbf{newtype } \mathit{Ftabovesealevel} &&= \mathit{Hei}\ \mathit{Int} && \text{-- Elevation in feet above sea level}\\
&\textbf{newtype } \mathit{Established} &&= \mathit{Est}\ \mathit{Int} && \text{-- Year of establishment}\\
&\text{-- Example values of the new types}\\
&\mathit{pop} :: \mathit{Population}; && \mathit{pop} = \mathit{Pop}\ 562;\\
&\mathit{hei} :: \mathit{Ftabovesealevel}; && \mathit{hei} = \mathit{Hei}\ 2150;\\
&\mathit{est} :: \mathit{Established}; && \mathit{est} = \mathit{Est}\ 1951;
\end{aligned}$$

This example introduces three new types, *Population*, *Ftabovesealevel*, and *Established*, which all are internally represented by an *Int* but which are good to keep apart. The syntax also introduces *constructor functions* $\mathit{Pop} :: \mathit{Int} \to \mathit{Population}$, *Hei* and *Est* which can be used to translate from plain integers to the new types, and for pattern matching. The semantics of *Population* is the set of values of the form

Pop i for every value $i :: Int$. It is not the same as the semantics of *Int* but it is isomorphic (there is a one-to-one correspondence between the sets). [5]

Later in this chapter we use a newtype for the semantics of complex numbers as a pair of numbers in the Cartesian representation but it may also be useful to have another newtype for complex as a pair of numbers in the polar representation.

The keyword data – for syntax trees The simplest form of a recursive datatype is the unary notation for natural numbers:

> **data** $N = Z \mid S\ N$

This declaration introduces

- a new type N for unary natural numbers,
- a constructor $Z :: N$ to represent zero, and
- a constructor $S :: N \to N$ to represent the successor.

The semantics of N is the infinite set $\{Z, S\ Z, S\ (S\ Z), ...\}$ which is isomorphic to $\mathbb{N}$. Examples values: $zero = Z$, $one = S\ Z$, $three = S\ (S\ one)$.

The **data** keyword will be used throughout the book to define (inductive) datatypes of syntax trees for different kinds of expressions: simple arithmetic expressions, complex number expressions, etc. But it can also be used for non-inductive datatypes, like **data** $Bool = False \mid True$, or **data** $TownData = Town\ String\ Population\ Established$. The *Bool* type is the simplest example of a *sum type*, where each value uses either of the two variants *False* and *True* as the constructor. The *TownData* type is an example of a *product type*, where each value uses the same constructor *Town* and records values for the name, population, and year of establishment of the town modelled. (See Exercise 1.5 for the intuition behind the terms "sum" and "product" used here.)

***Maybe* and parameterised types** It is very often possible to describe a family of types using a type parameter. One simple example is the type constructor *Maybe*:

> **data** $Maybe\ a = Nothing \mid Just\ a$

This declaration introduces

[5] Pedantic remark: Usually an isomorphism cares not only about the elements of the sets, but some structure involving operations defined on this set (otherwise the term "bijection" is more typical). Here we have not talked about these operations (yet). If we were to, we'd be able to establish a more useful isomorphism. (And we only introduce algebraic structures in Chapter 4.)

- a new type *Maybe a* for every type a,
- a constructor $Nothing :: Maybe\ a$ to represent "no value", and
- a constructor $Just :: a \to Maybe\ a$ to represent "just a value".

A maybe type is often used when an operation may, or may not, return a value:

$$
\begin{aligned}
&inv :: \mathbb{Q} \to Maybe\ \mathbb{Q} \\
&inv\ 0 = Nothing \\
&inv\ r = Just\ (1\ /\ r)
\end{aligned}
$$

Two other examples of, often used, parameterised types are (a, b) for the type of pairs (a product type) and *Either a b* for either an a or a b (a sum type).

TODO: JP: Why do we give the semantics of functions but not the semantics of other types?

$$\textbf{data}\ Either\ p\ q = Left\ p \mid Right\ q$$

1.2 Notation and abstract syntax for sequences

TODO: JP: BLOAT: Seems like an odd place to talk about this. Why not put it together with the limits section?

As preparation for the language of sequences and limits later (Sections 2.5.1 and 6.4), we spend a few lines on the notation and abstract syntax of sequences.

In math textbooks, the following notation is commonly in use: $\{a_i\}_{i=0}^{\infty}$ or just $\{a_i\}$ and (not always) an indication of the type X of the a_i. Note that the a at the center of this notation actually carries all of the information: an infinite family of values a_i each of type X. If we interpret the subscript notation a_i as function application ($a(i)$) we can see that $a : \mathbb{N} \to X$ is a useful typing of an infinite sequence. Some examples:

$$
\begin{aligned}
&\textbf{type}\ \mathbb{N} = Natural && \text{-- imported from } Numeric.Natural \\
&\textbf{type}\ \mathbb{Q}^+ = Ratio\ \mathbb{N} && \text{-- imported from } Data.Ratio \\
&\textbf{type}\ Seq\ a = \mathbb{N} \to a \\
&idSeq :: Seq\ \mathbb{N} \\
&idSeq\ i = i && \text{-- } \{0, 1, 2, 3, \ldots\} \\
&invSeq :: Seq\ \mathbb{Q}^+ \\
&invSeq\ i = 1\ \%\ (1 + i) && \text{-- } \{\tfrac{1}{1}, \tfrac{1}{2}, \tfrac{1}{3}, \tfrac{1}{4}, \ldots\} \\
&pow2 :: Num\ r \Rightarrow Seq\ r \\
&pow2 = (2\hat{}) && \text{-- } \{1, 2, 4, 8, \ldots\} \\
&conSeq :: a \to Seq\ a \\
&conSeq\ c\ i = c && \text{-- } \{c, c, c, c, \ldots\}
\end{aligned}
$$

What operations can be performed on sequences? We have seen the first one: given a value c we can generate a constant sequence with *conSeq c*. We can also add sequences componentwise (also called "pointwise"):

$$addSeq :: Num\ a \Rightarrow Seq\ a \to Seq\ a \to Seq\ a$$
$$addSeq\ f\ g\ i = f\ i + g\ i$$

and in general we can lift any binary operation $op :: a \to b \to c$ to the corresponding, pointwise, operation of sequences:

$$liftSeq_2 :: (a \to b \to c) \to Seq\ a \to Seq\ b \to Seq\ c$$
$$liftSeq_2\ op\ f\ g\ i = op\ (f\ i)\ (g\ i) \quad \text{-- } \{op\ (f\ 0)\ (g\ 0), op\ (f\ 1)\ (g\ 1), ...\}$$

Similarly we can lift unary operations, and "nullary" operations:

$$liftSeq_1 :: (a \to b) \to Seq\ a \to Seq\ b$$
$$liftSeq_1\ h\ f\ i = h\ (f\ i) \quad \text{-- } \{h\ (f\ 0), h\ (f\ 1), h\ (f\ 2), ...\}$$
$$liftSeq_0 :: a \to Seq\ a$$
$$liftSeq_0\ c\ i = c$$

Exercise 1.10: what does function composition do to a sequence? For a sequence a what is $a \circ (1+)$? What is $(1+) \circ a$?

Another common mathematical operator on sequences is the limit. We will get back to limits later (Sections 2.5 and 2.5.2), but for now we just analyse the notation and typing. This definition is slightly adapted from Wikipedia (2017-11-08):

> We call L the limit of the sequence $\{x_n\}$ if the following condition holds: For each real number $\epsilon > 0$, there exists a natural number N such that, for every natural number $n \geqslant N$, we have $|x_n - L| < \epsilon$.
>
> If so, we say that the sequence converges to L and write
>
> $$L = \lim_{i \to \infty} x_i$$

There are (at least) two things to note here. First, with this syntax, the $\lim_{i\to\infty} x_i$ expression form binds i in the expression x_i. We could just as well say that *lim* takes a function $x :: \mathbb{N} \to X$ as its only argument. Second, an arbitrary sequence x, may or may not have a limit. Thus the customary use of $L =$ is a bit of an abuse of notation, because the right hand side may not be well defined. One way to capture that idea is to let *lim* return the type *Maybe X*, with *Nothing* corresponding to divergence. Then its complete type is $(\mathbb{N} \to X) \to Maybe\ X$ and $L = \lim_{i\to\infty} x_i$ means $Just\ L = lim\ x$ We will return to limits and their proofs in Section 2.5.1 after we have reviewed some logic.

TODO: JP: This is explained in Section 3.1

TODO: JP: Again, awkward back-and-forth

Here we just define one more common operation: the sum of a sequence (like $\sigma = \sum_{i=0}^{\infty} 1/i!$[6]). Just as not all sequences have a limit, not all have a sum either. But for every sequence we can define a new sequence of partial sums:

[6] Here $n! = 1 * 2 * ... * n$ is the factorial .

$$
\begin{aligned}
&\mathit{sums} :: \mathit{Num}\ a \Rightarrow \mathit{Seq}\ a \to \mathit{Seq}\ a \\
&\mathit{sums} = \mathit{scan}\ (+)\ 0
\end{aligned}
$$

The function *sums* is perhaps best illustrated by examples:

$$
\begin{aligned}
&\mathit{sums}\ (\mathit{conSeq}\ c) &&== \{0, c, 2 * c, 3 * c, \ldots\} \\
&\mathit{sums}\ (\mathit{idSeq}) &&== \{0, 0, 1, 3, 6, 10, \ldots\}
\end{aligned}
$$

The general pattern is to start at zero and accumulate the sum of initial prefixes of the input sequence (we leave the defintion of *scan* as an exercise to the reader).

By combining *sums* with limits we can state formally that the sum of an infinite sequence a exists and is S iff the limit of *sums a* exists and is S. We can write the above as a formula: $\mathit{Just}\ S = \mathit{lim}\ (\mathit{sums}\ a)$. For our example it turns out that the sum converges and that $\sigma = \sum_{i=0}^{\infty} 1/i! = e$ but we will not get to that until Section 8.1.

We will also return to (another type of) limits in Section 3.4 about derivatives where we explore variants of the classical definition

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

To sum up this subsection, we have defined a small Domain-Specific Language (DSL) for infinite sequences by defining a type (*Seq a*), some operations (*conSeq*, *addSeq*, $\mathit{liftSeq}_1$, *sums*, ...) and some evaluation functions or predicates (like *lim* and *sum*).

1.3 A DSL of complex numbers[7]

We now turn to our first study of mathematics as found "in the wild": we will do an analytic reading of a piece of the introduction of complex numbers by Adams and Essex [2010]. We choose a simple domain to allow the reader to concentrate on the essential elements of our approach without the distraction of potentially unfamiliar mathematical concepts. In fact, for this section, we temporarily pretend to forget any previous knowledge of complex numbers, and study the textbook of Adams and Essex as we would approach a completely new domain, even if that leads to a somewhat exaggerated attention to detail.

Adams and Essex introduce complex numbers in Appendix A of their book. The section *Definition of Complex Numbers* starts with:

[7] This part is partly based on material by Ionescu and Jansson [2016] which appeared at the International Workshop on Trends in Functional Programming in Education, 2015.

> We begin by defining the symbol i, called **the imaginary unit**, to have the property
>
> $$i^2 = -1$$
>
> Thus, we could also call i the square root of -1 and denote it $\sqrt{-1}$. Of course, i is not a real number; no real number has a negative square.

At this stage, it is not clear what the type of i is meant to be, we only know that i is not a real number. Moreover, we do not know what operations are possible on i, only that i^2 is another name for -1 (but it is not obvious that, say $i * i$ is related in any way with i^2, since the operations of multiplication and squaring have only been introduced so far for numerical types such as $\mathbb{N}$ or $\mathbb{R}$, and not for "symbols").

For the moment, we introduce a type for the symbol i, and, since we know nothing about other symbols, we make i the only member of this type. We use a capital I in the **data** declaration because a lowercase constructor name would cause a syntax error in Haskell. For convenience we add a synonym $i = I$.

```
data ImagUnits = I
i :: ImagUnits
i = I
```

We can give the translation from the abstract syntax to the concrete syntax as a function *showIU*:

```
showIU :: ImagUnits → String
showIU  I           = "i"
```

Next, in the book, we find the following definition:

> **Definition:** A **complex number** is an expression of the form
>
> $$a + bi \qquad \text{or} \qquad a + ib,$$
>
> where a and b are real numbers, and i is the imaginary unit.

This definition clearly points to the introduction of a syntax (notice the keyword "form"). This is underlined by the presentation of *two* forms, which can suggest that the operation of juxtaposing i (multiplication?) is not commutative.

A profitable way of dealing with such concrete syntax in functional programming is to introduce an abstract representation of it in the form of a datatype:

$$\begin{array}{l}\textbf{data}\ \mathit{ComplexA} = \mathit{CPlus}_1\ \mathbb{R}\ \mathbb{R}\ \mathit{ImagUnits} \quad \text{-- the form } a + bi \\ \qquad\qquad\quad \mid\ \mathit{CPlus}_2\ \mathbb{R}\ \mathit{ImagUnits}\ \mathbb{R} \quad \text{-- the form } a + ib\end{array}$$

We can give the translation from the (abstract) syntax to its concrete representation as a string of characters, as the function *showCA*:

$$\begin{array}{l}\mathit{showCA} :: \mathit{ComplexA} \to \quad \mathit{String} \\ \mathit{showCA}\quad (\mathit{CPlus}_1\ x\ y\ i) = \mathit{show}\ x +\!\!+ \text{" + "} +\!\!+ \mathit{show}\ y +\!\!+ \mathit{showIU}\ i \\ \mathit{showCA}\quad (\mathit{CPlus}_2\ x\ i\ y) = \mathit{show}\ x +\!\!+ \text{" + "} +\!\!+ \mathit{showIU}\ i +\!\!+ \mathit{show}\ y\end{array}$$

Notice that the type $\mathbb{R}$ is not implemented yet (and it is not even clear how to implement it with fidelity to mathematical convention at this stage) but we want to focus on complex numbers so we will simply approximate $\mathbb{R}$ by double precision floating point numbers for now.

$$\textbf{type}\ \mathbb{R} = \mathit{Double}$$

Adams and Essex continue with examples:

> For example, $3 + 2i$, $\frac{7}{2} - \frac{2}{3}i$, $i\pi = 0 + i\pi$ and $-3 = -3 + 0i$ are all complex numbers. The last of these examples shows that every real number can be regarded as a complex number.

The second example is somewhat problematic: it does not seem to be of the form $a + bi$. Given that the last two examples seem to introduce shorthand for various complex numbers, let us assume that this one does as well, and that $a - bi$ can be understood as an abbreviation of $a + (-b)\ i$. With this provision, in our notation the examples are written as in Table 1.1. We interpret the sentence

Mathematics		Haskell
$3 + 2i$		$\mathit{CPlus}_1\ 3\ 2\ I$
$\frac{7}{2} - \frac{2}{3}i$	$= \frac{7}{2} + \frac{-2}{3}i$	$\mathit{CPlus}_1\ (7\ /\ 2)\ (-2\ /\ 3)\ I$
$i\pi$	$= 0 + i\pi$	$\mathit{CPlus}_2\ 0\ I\ \pi$
-3	$= -3 + 0i$	$\mathit{CPlus}_1\ (-3)\ 0\ I$

Table 1.1: Examples of notation and abstract syntax for some complex numbers.

"The last of these examples ..." to mean that there is an embedding of the real numbers in *ComplexA*, which we introduce explicitly:

$$\begin{array}{l}\mathit{toComplex} :: \mathbb{R} \to \mathit{ComplexA} \\ \mathit{toComplex}\ x = \mathit{CPlus}_1\ x\ 0\ I\end{array}$$

Again, at this stage there are many open questions. For example, we can assume that the mathematical expression $i1$ stands for the complex number $\mathit{CPlus}_2\ 0\ I\ 1$, but what about the expression i by itself? If juxtaposition is

meant to denote some sort of multiplication, then perhaps 1 can be considered as a unit, in which case we would have that i abbreviates $i1$ and therefore $CPlus_2\ 0\ I\ 1$. But what about, say, $2i$? Abbreviations with i have only been introduced for the ib form, and not for the bi one!

The text then continues with a parenthetical remark which helps us dispel these doubts:

> (We will normally use $a + bi$ unless b is a complicated expression, in which case we will write $a + ib$ instead. Either form is acceptable.)

This remark suggests strongly that the two syntactic forms are meant to denote the same elements, since otherwise it would be strange to say "either form is acceptable". After all, they are acceptable according to the definition provided earlier.

Given that $a + ib$ is only "syntactic sugar" for $a + bi$, we can simplify our representation for the abstract syntax, eliminating one of the constructors:

> **data** $ComplexB = CPlusB\ \mathbb{R}\ \mathbb{R}\ ImagUnits$

In fact, since it doesn't look as though the type *ImagUnits* will receive more elements, we can dispense with it altogether:

> **data** $ComplexC = CPlusC\ \mathbb{R}\ \mathbb{R}$

(The renaming of the constructor to *CPlusC* serves as a guard against the case that we have suppressed potentially semantically relevant syntax.)

We read further:

> It is often convenient to represent a complex number by a single letter; w and z are frequently used for this purpose. If a, b, x, and y are real numbers, and $w = a + bi$ and $z = x + yi$, then we can refer to the complex numbers w and z. Note that $w = z$ if and only if $a = x$ and $b = y$.

First, let us notice that we are given an important semantic information: to check equality for complex numbers, it is enough to check equality of the components (the arguments to the constructor *CPlusC*). Another way of saying this is that *CPlusC* is injective. In Haskell we could define this equality as:

```
instance Eq ComplexC where
  CPlusC a b == CPlusC x y = a == x ∧ b == y
```

The line **instance** *Eq ComplexC* is there because the specify that *ComplexC* supports the $(==)$ operator. (The cognoscenti would prefer to obtain an equivalent definition using the shorter **deriving** *Eq* clause upon defining the type.)

This shows that the set of complex numbers is, in fact, isomorphic with the set of pairs of real numbers, a point which we can make explicit by re-formulating the definition in terms of a **newtype**:

TODO: JP: have we ever defined isomorphism?

$$\textbf{newtype}\ \mathit{ComplexD} = \mathit{CD}\ (\mathbb{R}, \mathbb{R})\ \textbf{deriving}\ \mathit{Eq}$$

As we see it, the somewhat confusing discussion of using "letters" to stand for complex numbers serves several purposes. First, it hints at the implicit typing rule that the symbols z and w should be complex numbers. Second, it shows that, in mathematical arguments, one needs not abstract over two real variables: one can instead abstract over a single complex variable. We already know that we have an isomorphism between pair of reals and complex numbers. But additionally, we have a notion of *pattern matching*, as in the following definition:

> **Definition:** If $z = x + yi$ is a complex number (where x and y are real), we call x the **real part** of z and denote it $\mathit{Re}\ (z)$. We call y the **imaginary part** of z and denote it $\mathit{Im}\ (z)$:
>
> $$\begin{aligned} \mathit{Re}\ (z) &= \mathit{Re}\ (x + yi) = x \\ \mathit{Im}\ (z) &= \mathit{Im}\ (x + yi) = y \end{aligned}$$

This is rather similar to Haskell's *as-patterns*:

$$\begin{aligned} &\mathit{re} :: \mathit{ComplexD} \quad \to \mathbb{R} \\ &\mathit{re}\ z@(\mathit{CD}\ (x, y)) \ = x \\ &\mathit{im} :: \mathit{ComplexD} \quad \to \mathbb{R} \\ &\mathit{im}\ z@(\mathit{CD}\ (x, y)) \ = y \end{aligned}$$

a potential source of confusion being that the symbol z introduced by the as-pattern is not actually used on the right-hand side of the equations (although it could be).

The use of as-patterns such as "$z = x + yi$" is repeated throughout the text, for example in the definition of the algebraic operations on complex numbers:

> **The sum and difference of complex numbers**
>
> If $w = a + bi$ and $z = x + yi$, where a, b, x, and y are real numbers, then
>
> $$\begin{aligned} w + z &= (a + x) + (b + y)\ i \\ w - z &= (a - x) + (b - y)\ i \end{aligned}$$

With the introduction of algebraic operations, the language of complex numbers becomes much richer. We can describe these operations in a *shallow embedding* in terms of the concrete datatype *ComplexD*, for example:

$$plusD :: ComplexD \to ComplexD \to ComplexD$$
$$plusD\ (CD\ (a,b))\ (CD\ (x,y)) = CD\ ((a+x),(b+y))$$

or we can build a datatype of "syntactic" complex numbers from the algebraic operations to arrive at a *deep embedding* as seen in the next section. Both shallow and deep embeddings will be further explained in Sections 3.1.1 and 4.6.[8]

At this point we can sum up the "evolution" of the datatypes introduced so far. Starting from *ComplexA*, the type has evolved by successive refinements through *ComplexB*, *ComplexC*, ending up in *ComplexD* (see Fig. 1.3). We can also make a parameterised version of *ComplexD*, by noting that the definitions for complex number operations work fine for a range of underlying numeric types. The operations for *ComplexSem* are defined in module *CSem*, available in Appendix A.

$$\begin{array}{lll}
\textbf{data} & ImagUnits & = I \\
\textbf{data} & ComplexA & = CPlus_1\ \mathbb{R}\ \mathbb{R}\ ImagUnits \\
 & & \mid CPlus_2\ \mathbb{R}\ ImagUnits\ \mathbb{R} \\
\textbf{data} & ComplexB & = CPlusB\ \mathbb{R}\ \mathbb{R}\ ImagUnits \\
\textbf{data} & ComplexC & = CPlusC\ \mathbb{R}\ \mathbb{R} \\
\textbf{newtype} & ComplexD & = CD\ (\mathbb{R},\mathbb{R})\ \textbf{deriving}\ Eq \\
\textbf{newtype} & ComplexSem\ r & = CS\ \ (r,r)\quad \textbf{deriving}\ Eq
\end{array}$$

Figure 1.3: Complex number datatype refinement (semantics).

1.3.1 A syntax for (complex) arithmetical expressions

By following Adams and Essex [2010], we have arrived at representation which captures the *semantics* of complex numbers. This kind of representation is often called a "shallow embedding". Now we turn to the study of the *syntax* instead ("deep embedding").

We want a datatype *ComplexE* for the abstract syntax tree of expressions. The syntactic expressions can later be evaluated to semantic values: The concept of "an evaluator", a function from the syntax to the semantics, is something we will return to many times in this book.

$$evalE :: ComplexE \to ComplexD$$

The datatype *ComplexE* should collect ways of building syntactic expressions representing complex numbers and we have so far seen the symbol i, an embedding from $\mathbb{R}$, plus and times. We make these four *constructors* in one recursive datatype as follows:

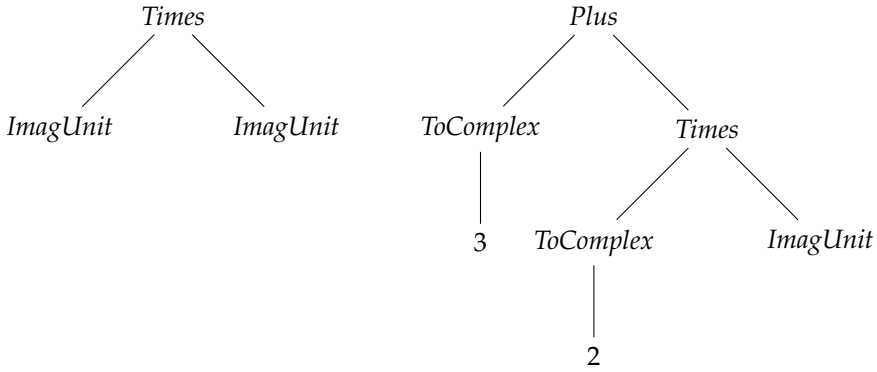
[8] And several other places: this is a recurrent idea of the book

```
data ComplexE = ImagUnit    -- syntax for i, not to be confused with the type ImagUnits
              | ToComplex ℝ
              | Plus    ComplexE ComplexE
              | Times ComplexE ComplexE
deriving (Eq, Show)
```

Note that, in *ComplexA* above, we also had a constructor for "plus" ($CPlus_1$), but it was playing a different role. They are distinguished by type: $CPlus_1$ took two real numbers as arguments, while *Plus* here takes two complex expressions as arguments.

Here are two examples of type *ComplexE* as Haskell code and as abstract syntax trees:

```
testE1 = Times ImagUnit ImagUnit
testE2 = Plus (ToComplex 3) (Times (ToComplex 2) ImagUnit)
```



We can implement the evaluator *evalE* by pattern matching on the constructors of the syntax tree and by recursion. To write a recursive function requires a small leap of faith. It can be difficult to get started implementing a function (like *evalE*) that should handle all the cases and all the levels of a recursive datatype (like *ComplexE*). One way to overcome this difficulty is through what may seem at first glance "wishful thinking": assume that all but one case have been implemented already. All you need is to focus on that one remaining case, and you can freely call the function (that you are implementing) recursively, as long as you do it for subexpressions (subtrees of the abstract syntax tree datatype). This pattern is called *structural induction*.

For example, when implementing the *evalE* $(Plus\ \ c_1\ \ c_2)$ case, you can assume that you already know the values $s_1, s_2 :: ComplexD$ corresponding to the subtrees c_1 and c_2 of type *ComplexE*. The only thing left is to add them up componentwise and we can assume there is a function $plusD :: ComplexD \to ComplexD \to ComplexD$ taking care of this step (in fact, we implemented it earlier in Section 1.3). Continuing in this direction (by structural induction or "wishful thinking") we arrive at the following implementation.

$$
\begin{aligned}
&\mathit{evalE}\ \mathit{ImagUnit} &&= \mathit{imagUnitD}\\
&\mathit{evalE}\ (\mathit{ToComplex}\ r) &&= \mathit{toComplexD}\ r\\
&\mathit{evalE}\ (\mathit{Plus}\ c_1\ c_2) &&= \mathit{plusD}\ \ (\mathit{evalE}\ c_1)\ (\mathit{evalE}\ c_2)\\
&\mathit{evalE}\ (\mathit{Times}\ c_1\ c_2) &&= \mathit{timesD}\ (\mathit{evalE}\ c_1)\ (\mathit{evalE}\ c_2)
\end{aligned}
$$

Note the pattern here: for each constructor of the syntax datatype we assume that there exists a corresponding semantic function. The next step is to implement these functions, but let us first list their types and compare them with the types of the syntactic constructors:

$$
\begin{aligned}
&\mathit{imagUnitD} :: \mathit{ComplexD} && \text{-- } \mathit{ComplexE}\\
&\mathit{toComplexD} :: \mathbb{R} \to \mathit{ComplexD} && \text{-- } \mathbb{R} \to \mathit{ComplexE}\\
&\mathit{timesD} :: \mathit{ComplexD} \to \mathit{ComplexD} \to \mathit{ComplexD} && \text{-- } \mathit{ComplexE} \to \mathit{ComplexE} \to \mathit{ComplexE}
\end{aligned}
$$

As we can see, each use of *ComplexE* has been replaced be a use of *ComplexD*. Finally, we can start filling in the implementations:

$$
\begin{aligned}
&\mathit{imagUnitD} &&= \mathit{CD}\ (0,1)\\
&\mathit{toComplexD}\ r &&= \mathit{CD}\ (r,0)
\end{aligned}
$$

The function *plusD* was defined earlier and *timesD* is left as an exercise for the reader. To sum up we have now implemented a recursive datatype for mathematical expressions describing complex numbers, and an evaluator that computes the underlying number. Note that many different syntactic expressions will evaluate to the same number (*evalE* is not injective).

Generalising from the example of *testE2* we also define a function to embed a semantic complex number in the syntax:

$$
\begin{aligned}
&\mathit{fromCD} :: \mathit{ComplexD} \to \mathit{ComplexE}\\
&\mathit{fromCD}\ (\mathit{CD}\ (x,y)) = \mathit{Plus}\ (\mathit{ToComplex}\ x)\ (\mathit{Times}\ (\mathit{ToComplex}\ y)\ \mathit{ImagUnit})
\end{aligned}
$$

This function is injective: different complex numbers map to different syntactic expressions.

1.4 Laws, properties and testing

There are certain laws that we would like to hold for operations on complex numbers. To specify these laws, in a way which can be easily testable in Haskell, we use functions to *Bool* (also called *predicates* or *properties*). The intended meaning of such a boolean function (representing a law) is "forall inputs, this should return *True*". This idea is at the core of *property based testing* (pioneered by Claessen and Hughes [2000]) and conveniently available in the library QuickCheck.

The simplest law is perhaps $i^2 = -1$ from the start of Section 1.3,

$$
\begin{aligned}
&\mathit{propImagUnit} :: \mathit{Bool}\\
&\mathit{propImagUnit} = \mathit{Times}\ \mathit{ImagUnit}\ \mathit{ImagUnit} \mathrel{=\!\!=\!\!=} \mathit{ToComplex}\ (-1)
\end{aligned}
$$

Note the we use a new operator here, $(===)$. Indeed, we reserve the usual equality $(==)$ for syntactic equality (and here the left hand side (LHS) is clearly not syntactically equal to the right hand side). The new operator $(===)$ corresponds to semantic equality, that is, equality *after evaluation*:

$$
\begin{aligned}
&(===) :: \mathit{ComplexE} \to \mathit{ComplexE} \to \mathit{Bool} \\
&z === w \quad = \quad \mathit{evalE}\ z == \mathit{evalE}\ w
\end{aligned}
$$

Another law is that *fromCD* is an embedding: if we start from a semantic value, embed it back into syntax, and evaluate that syntax we get back to the value we started from.

$$
\begin{aligned}
&\mathit{propFromCD} :: \mathit{ComplexD} \to \mathit{Bool} \\
&\mathit{propFromCD}\ s = \mathit{evalE}\ (\mathit{fromCD}\ s) == s
\end{aligned}
$$

Other desirable laws are that $+$ and $*$ should be associative and commutative and $*$ should distribute over $+$:

$$
\begin{aligned}
&\mathit{propCommPlus}\ \ x\ y && = && x + y && === y + x \\
&\mathit{propCommTimes}\ x\ y && = && x * y && === y * x \\
&\mathit{propAssocPlus}\ \ x\ y\ z && = && (x + y) + z && === x + (y + z) \\
&\mathit{propAssocTimes}\ x\ y\ z && = && (x * y) * z && === x * (y * z) \\
&\mathit{propDistTimesPlus}\ x\ y\ z && = && x * (y + z) && === (x * y) + (x * z)
\end{aligned}
$$

These laws actually fail, but not due to any mistake in the implementation of *evalE* in itself. To see this, let us consider associativity at different types:

$$
\begin{aligned}
&\mathit{propAssocInt} \quad = \mathit{propAssocPlus} :: \mathit{Int} \quad \to \mathit{Int} \quad \to \mathit{Int} \quad \to \mathit{Bool} \\
&\mathit{propAssocDouble} = \mathit{propAssocPlus} :: \mathit{Double} \to \mathit{Double} \to \mathit{Double} \to \mathit{Bool}
\end{aligned}
$$

The first property is fine, but the second fails. Why? QuickCheck can be used to find small examples — this one is perhaps the best one:

$$
\begin{aligned}
&\mathit{notAssocEvidence} :: (\mathit{Double}, \mathit{Double}, \mathit{Double}, \mathit{Bool}) \\
&\mathit{notAssocEvidence} = (\mathit{lhs}, \mathit{rhs}, \mathit{lhs} - \mathit{rhs}, \mathit{lhs} == \mathit{rhs}) \\
&\quad \textbf{where}\ \mathit{lhs} = (1 + 1) + 1\ /\ 3 \\
&\qquad\quad\ \ \mathit{rhs} = 1 + (1 + 1\ /\ 3)
\end{aligned}
$$

For completeness: these are the values:

```
(2.3333333333333335      -- Notice the five at the end
,2.333333333333333,      -- which is not present here.
,4.440892098500626e−16   -- The (very small) difference
,False)
```

We can now see the underlying reason why some of the laws failed for complex numbers: the approximative nature of *Double*. Therefore, to ascertain that there

is no other bug hiding, we need to move away from the implementation of $\mathbb{R}$ as *Double*. We do this by abstraction: we make one more version of the complex number type, which is parameterised on the underlying representation type for $\mathbb{R}$. At the same time we combine *ImagUnit* and *ToComplex* to *ToComplexCart*, which corresponds to the primitive from $a + bi$ discussed above: TODO: JP: why though?

$$
\begin{aligned}
&\textbf{data}\ \mathit{ComplexSyn}\ r = \mathit{ToComplexCart}\ r\ r \\
&\qquad\qquad\qquad\quad \mid \mathit{ComplexSyn}\ r :+: \mathit{ComplexSyn}\ r \\
&\qquad\qquad\qquad\quad \mid \mathit{ComplexSyn}\ r :*: \mathit{ComplexSyn}\ r \\
&\mathit{toComplexSyn} :: \mathit{Num}\ a \Rightarrow a \to \mathit{ComplexSyn}\ a \\
&\mathit{toComplexSyn}\ x = \mathit{ToComplexCart}\ x\ 0
\end{aligned}
$$

From Appendix A we import **newtype** $\mathit{ComplexSem}\ r = \mathit{CS}\ (r, r)$ **deriving** *Eq* and the semantic operations $(.+.)$ and $(.*.)$ corresponding to *plusD* and *timesD*.

$$
\begin{aligned}
&\mathit{evalCSyn} :: \mathit{Num}\ r \Rightarrow \mathit{ComplexSyn}\ r \to \mathit{ComplexSem}\ r \\
&\mathit{evalCSyn}\ (\mathit{ToComplexCart}\ x\ y) = \mathit{CS}\ (x, y) \\
&\mathit{evalCSyn}\ (l :+: r) \qquad\quad = \mathit{evalCSyn}\ l\ .+.\ \mathit{evalCSyn}\ r \\
&\mathit{evalCSyn}\ (l :*: r) \qquad\quad = \mathit{evalCSyn}\ l\ .*.\ \mathit{evalCSyn}\ r
\end{aligned}
$$

Exercise 1.1. Add a few more operations (hint: extend *ComplexSyn* as well) and extend *eval* appropriately.

With this parameterised type we can test the code for "complex rationals"[9] to avoid rounding errors.

1.4.1 Generalising laws

Some laws appear over and over again in different mathematical contexts. For example, binary operators are often associative or commutative, and sometimes one operator distributes over another. We will work more formally with logic in Chapter 2 but we introduce a few definitions already here:

$\mathit{Associative}\ (\circledast) = \forall\, a, b, c.\ \ (a \circledast b) \circledast c = a \circledast (b \circledast c)$

$\mathit{Commutative}\ (\circledast) = \forall\, a, b.\ \ a \circledast b = b \circledast a$

$\mathit{Distributive}\ (\otimes)\ (\oplus) = \forall\, a, b, c.\ \ (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$

The above laws are *parameterised* over some operators ($(\circledast), (\otimes), (\oplus)$). These laws will hold for some operators, but not for others. For example, division is not commutative; taking the average of two quantities is commutative but not associative.[10]

[9] The reason why math textbooks never talk about this version of complex numbers is because complex numbers are used to handle roots of all numbers uniformly, and roots are in general irrational.

[10] See also Item 4, on page 104 for further analysis of distributivity.

Such generalisation can be reflected in QuickCheck properties as well.

$$\begin{aligned}&\mathit{propAssoc} :: \mathit{SemEq}\ a \Rightarrow (a \to a \to a) \to a \to a \to a \to \mathit{Bool}\\&\mathit{propAssoc}\ (\circledast)\ x\ y\ z = (x \circledast y) \circledast z \mathrel{===} x \circledast (y \circledast z)\end{aligned}$$

Note that *propAssocA* is a higher order function: it takes a function $(\circledast)$ (written as a binary operator) as its first parameter, and tests if it is associative. The property is also polymorphic: it works for many different types a (all types which have an $===$ operator).

Thus we can specialise it to *Plus*, *Times* and any other binary operator, and obtain some of the earlier laws (*propAssocPlus*, *propAssocTimes*). The same can be done with distributivity. Doing so we learnt that the underlying set matters: $(+)$ for $\mathbb{R}$ has some properties, but $(+)$ for *Double* has others. When formalising math as DSLs, approximation is sometimes convenient, but makes many laws false. Thus, we should attempt to do it late, and if possible, leave a parameter to make the degree of approximation tunable (*Int*, *Integer*, *Float*, *Double*, $\mathbb{Q}$, syntax trees, etc.).

Exercise 1.2. Find other pairs of operators satisfying a distributive law.

1.5 Exercises: Haskell, DSLs and complex numbers

Exercise 1.3. Consider the following data type for simple arithmetic expressions:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Exp} = \mathit{Con}\ \mathit{Integer} \\
&\qquad\qquad \mid \mathit{Exp}\ \text{`}\mathit{Plus}\text{`}\ \ \mathit{Exp} \\
&\qquad\qquad \mid \mathit{Exp}\ \text{`}\mathit{Minus}\text{`}\ \mathit{Exp} \\
&\qquad\qquad \mid \mathit{Exp}\ \text{`}\mathit{Times}\text{`}\ \ \mathit{Exp} \\
&\quad \textbf{deriving}\ (\mathit{Eq}, \mathit{Show})
\end{aligned}
$$

Note the use of "backticks" around *Plus* etc. which makes it possible to use a name as an infix operator.

1. Write the following expressions in Haskell, using the *Exp* data type:
   (a) $a_1 = 2 + 2$
   (b) $a_2 = a_1 + 7 * 9$
   (c) $a_3 = 8 * (2 + 11) - (3 + 7) * (a_1 + a_2)$
2. Create a function $\mathit{eval} :: \mathit{Exp} \rightarrow \mathit{Integer}$ that takes a value of the *Exp* data type and returns the corresponding number (for instance, $\mathit{eval}\ ((\mathit{Con}\ 3)\ \text{`}\mathit{Plus}\text{`}\ (\mathit{Con}\ 3)) == 6$). Try it on the expressions from the first part, and verify that it works as expected.
3. Consider the following expression:

   $$
   \begin{aligned}
   &c_1 = (x - 15) * (y + 12) * z \\
   &\quad \textbf{where}\ x = 5; y = 8; z = 13
   \end{aligned}
   $$

   In order to represent this with our *Exp* data type, we are going to have to make some modifications:
   (a) Update the *Exp* data type with a new constructor $\mathit{Var}\ \ \mathit{String}$ that allows variables with strings as names to be represented. Use the updated *Exp* to write an expression for c_1 in Haskell.
   (b) Create a function $\mathit{varVal} :: \mathit{String} \rightarrow \mathit{Integer}$ that takes a variable name, and returns the value of that variable. For now, the function just needs to be defined for the variables in the expression above, i.e. varVal `"x"` should return 5, varVal `"y"` should return 8, and varVal `"z"` should return 13.
   (c) Update the *eval* function so that it supports the new *Var* constructor, and use it get a numeric value of the expression $c1$.

Exercise 1.4. We will now look at a slightly more generalized version of the *Exp* type from the previous exercise:

```
data E2 a = Con a
          | Var String
          | E2 a `Plus`  E2 a
          | E2 a `Minus` E2 a
          | E2 a `Times` E2 a
  deriving (Eq, Show)
```

The type has now been parametrized, so that it is no longer limited to representing expressions with integers, but can instead represent expressions with any type. For instance, we could have an *E2 Double* to represent expressions with doubles, or an *E2 ComplexD* to represent expressions with complex numbers.

1. Write the following expressions in Haskell, using the new *E2* data type.
   (a) $a_1 = 2.0 + a$
   (b) $a_2 = 5.3 + b * c$
   (c) $a_3 = a * (b + c) - (d + e) * (f + a)$
2. In order to evaluate these expressions, we will need a way of translating a variable name into the value. The following table shows the value of each variable in the expressions above:

Name	a	b	c	d	e	f
Value	1.5	4.8	2.4	7.4	5.8	1.7

   In Haskell, we can represent this table using a value of type *Table a* = *Env String a* = $[(String, a)]$, which is a list of pairs of variable names and values, where each entry in the list corresponds to a column in the table.
   (a) Express the table above in Haskell by creating *vars* :: *Table Double*.
   (b) Create a function *varVal* :: *Table a* $\rightarrow$ *String* $\rightarrow$ *a* that returns the value of a variable, given a table and a variable name. For instance, *varVal vars* `"d"` should return 7.4
   (c) Create a function *eval* :: *Num a* $\Rightarrow$ *Table a* $\rightarrow$ *E2 a* $\rightarrow$ *a* that takes a value of the new *E2* data type and returns the corresponding number. For instance, *eval vars* $((Con\ 2)$ `Plus` $(Var$ `"a"`$))$ == 3.5. Try it on the expressions from the first part, and verify that it works as expected.

Exercise 1.5. Counting values. Assume we have three finite types *a*, *b*, *c* with cardinalites *A*, *B*, and *C*. (For example, the cardinality of *Bool* is 2, the cardinality of *Weekday* is 7, etc.) Then what is the cardinality of the types *Either a b*? (a, b)? $a \rightarrow b$? etc. These rules for computing the cardinality suggests that *Either* is similar to sum, $(,)$ is similar to product and $(\rightarrow)$ to (flipped) power. These rules show that we can use many intuitions from high-school algebra when working with types.

Exercise 1.6. Counting *Maybe*s. For each of the following types, enumerate and count the values:

1. $Bool \to Maybe\ Bool$
2. $Maybe\ Bool \to Bool$
3. $Maybe\ (Bool, Maybe\ (Bool, Maybe\ Bool))$

This is an opportunity to practice the learning outcome "develop adequate notation for mathematical concepts": what is a suitable notation for values of type $Bool$, $Maybe\ a$, $a \to b$, (a, b), etc.?

Exercise 1.7. Functions as tuples. For any type t the type $Bool \to t$ is basically "the same" as the type (t, t). Implement the two functions *isoR* and *isoL* forming an isomorphism:

$$isoR :: (Bool \to t) \to (t, t)$$
$$isoL :: (t, t) \to (Bool \to t)$$

and show that $isoL \circ isoR = id$ and $isoR \circ isoL = id$.

Exercise 1.8. Functions and pairs (the "tupling transform"). From one function $f :: a \to (b, c)$ returning a pair, you can always make a pair of two functions $pf :: (a \to b, a \to c)$. Implement this transform:

$$f2p :: (a \to (b, c)) \to (a \to b, a \to c)$$

Also implement the opposite transform:

$$p2f :: (a \to b, a \to c) \to (a \to (b, c))$$

This kind of transformation if often useful, and it works also for n-tuples.

Exercise 1.9. There is also a "dual" to the tupling transform: to show this, implement these functions:

$$s2p :: (Either\ b\ c \to a) \to (b \to a, c \to a)$$
$$p2s :: (b \to a, c \to a) \to (Either\ b\ c \to a)$$

Exercise 1.10. From Section 1.2:

- What does function composition do to a sequence? More concretely: for a sequence a what is $a \circ (1+)$? What is $(1+) \circ a$?
- How is $liftSeq_1$ related to *fmap*? $liftSeq_0$ to *conSeq*?

Exercise 1.11. Operator sections. Please fill out the remaining parts of this table with simplified expressions:

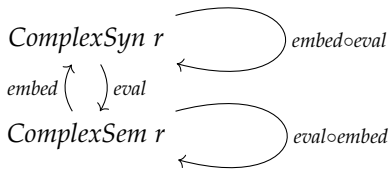
$$
\begin{aligned}
(1+) &= \lambda x \to 1 + x \\
(*2) &= \lambda x \to x * 2 \\
(1+) \circ (*2) &= \\
(*2) \circ (1+) &= \\
&= \lambda x \to x\hat{}2 + 1 \\
&= \lambda x \to (x + 1)\hat{}2 \\
(a+) \circ (b+) &=
\end{aligned}
$$

Exercise 1.12. Read the full chapter and complete the definition of the instance for *Num* for the datatype *ComplexSyn*. Also add a constructor for variables to enable writing expressions like $(Var\ \texttt{"z"}) :*: toComplex\ 1$.

Exercise 1.13. We can embed semantic complex numbers in the syntax:

$$
\begin{aligned}
&embed :: ComplexSem\ r \to ComplexSyn\ r \\
&embed\ (CS\ (x, y)) = ToComplexCart\ x\ y
\end{aligned}
$$

The embedding should satisfy a round-trip property: $eval\ (embed\ s) == s$ for all semantic complex numbers s. Here is a diagram showing how the types and the functions fit together



What about the opposite direction: when is $embed\ (eval\ e) == e$?

Step 0: type the quantification: what is the type of e?

Step 1: what equality is suitable for this type?

Step 2: if you use "equality up to eval" — how is the resulting property related to the first round-trip property?

Exercise 1.14. Read the next few pages of Appendix I (in [Adams and Essex, 2010]) defining the polar view of Complex Numbers and try to implement complex numbers again, this time based on magnitude and phase for the semantics.

Exercise 1.15. Implement a simplifier $simp :: ComplexSyn\ r \to ComplexSyn\ r$ that handles a few cases like $0 * x = 0$, $1 * x = x$, $(a + b) * c = a * c + b * c$, ... What class context do you need to add to the type of *simp*?

Exercise 1.16. A semiring is a set R equipped with two binary operations $+$ and $\cdot$, called addition and multiplication, such that:

- $(R, +, 0)$ is a commutative monoid with identity element 0:

  $$(a + b) + c = a + (b + c)$$
  $$0 + a = a + 0 = a$$
  $$a + b = b + a$$

- $(R, \cdot, 1)$ is a monoid with identity element 1:

  $$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$
  $$1 \cdot a = a \cdot 1 = a$$

- Multiplication left and right distributes over $(R, +, 0)$:

  $$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$$
  $$(a + b) \cdot c = (a \cdot c) + (b \cdot c)$$
  $$0 \cdot a = a \cdot 0 = 0$$

1. Define a datatype $SR\ v$ for the language of semiring expressions (with variables of type v). These are expressions formed from applying the semiring operations to the appropriate number of arguments, e.g., all the left hand sides and right hand sides of the above equations.
2. Implement the expressions from the laws.
3. Give a type signature for, and define, a general evaluator for $SR\ v$ expressions on the basis of an assignment function. An "assignment function" is a mapping from variable names to values.

Exercise 1.17. A *lattice* is a set L together with two operations $\vee$ and $\wedge$ (usually pronounced "sup" and "inf") such that

- $\vee$ and $\wedge$ are associative:

  $$\forall\, x, y, z \in L.\ \ (x \vee y) \vee z = x \vee (y \vee z)$$
  $$\forall\, x, y, z \in L.\ \ (x \wedge y) \wedge z = x \wedge (y \wedge z)$$

- $\vee$ and $\wedge$ are commutative:

  $$\forall\, x, y \in L.\ \ x \vee y = y \vee x$$
  $$\forall\, x, y \in L.\ \ x \wedge y = y \wedge x$$

- $\vee$ and $\wedge$ satisfy the *absorption laws*:

$$\forall\, x, y \in L.\ x \vee (x \wedge y) = x$$
$$\forall\, x, y \in L.\ x \wedge (x \vee y) = x$$

1. Define a datatype for the language of lattice expressions.
2. Define a general evaluator for *Lattice* expressions on the basis of an assignment function.

Exercise 1.18. An *abelian monoid* is a set M together with a constant (nullary operation) $0 \in M$ and a binary operation $\oplus : M \to M \to M$ such that:

- 0 is a unit of $\oplus$

$$\forall\, x \in M.\ 0 \oplus x = x \oplus 0 = x$$

- $\oplus$ is associative

$$\forall\, x, y, z \in M.\ x \oplus (y \oplus z) = (x \oplus y) \oplus z$$

- $\oplus$ is commutative

$$\forall\, x, y \in M.\ x \oplus y = y \oplus x$$

1. Define a datatype *AbMonoidExp* for the language of abelian monoid expressions. (These are expressions formed from applying the monoid operations to the appropriate number of arguments, e.g., all the left hand sides and right hand sides of the above equations.)
2. Define a general evaluator for *AbMonoidExp* expressions on the basis of an assignment function.

Chapter 2

Logic and calculational proofs

In this chapter, we continue to exercise our skill of organize areas of mathematics in DSL terms. We apply our methodology to the languages of logic: propositions and proofs. Additionally, at the same time, we will develop adequate notions and notations for mathematical foundations and learn to perform calculational proofs. There will be a fair bit of theory: introducing propositional and first order logic, but also applications to mathematics: prime numbers, (ir)rationals, limit points, limits, etc. and some Haskell concepts.

2.1 An aside: Pure set theory

One way to build mathematics from the ground up is to start from pure set theory and define all concepts by translation to sets. We will only work with this as a mathematical domain to study, not as "the right way" of doing mathematics (there are other ways). The core of the language of pure set theory is captured by four function symbols. We have a nullary function symbol $\{\,\}$ for the empty set (sometimes written $\emptyset$) and a unary function symbol S for the function that builds a singleton set from an "element". All non-variable terms so far are $\{\,\}$, $S\ \{\,\}$, $S\ (S\ \{\,\})$, ... The first set is empty but all the others are (different) one-element sets.

TODO: JP: Really? Where is the DSL?

Next we add two binary function symbols for union and intersection of sets (denoted by terms). Using union we can build sets of more than one element, for example $\mathit{Union}\ (S\ \{\,\})\ (S\ (S\ \{\,\}))$ which has two "elements": $\{\,\}$ and $S\ \{\,\}$.

In pure set theory we don't actually have any distinguished "elements" to start from (other than sets), but it turns out that quite a large part of mathematics can still be expressed. Every term in pure set theory denotes a set, and the elements of each set are again sets.

At this point it can be a good exercise to enumerate a few sets of cardinality[1] 0, 1, 2, and 3. There is really just one set of cardinality 0: the empty set $s0 = \{\}$. Using S we can then construct $s_1 = S\ s0$ of cardinality 1. Continuing in this manner we can build $s_2 = S\ s_1$, also of cardinality 1, and so on. Now we can combine different sets (like s_1 and s_2) with *Union* to build sets of cardinality 2: $s_3 = Union\ s_1\ s_2$, $s_4 = Union\ s_2\ s_3$, etc.. And we can at any point apply S to get back a new set of cardinality 1, like $s_5 = S\ s_3$.

Natural numbers To talk about things like natural numbers in pure set theory they need to be encoded. FOL does not have function definitions or recursion, but in a suitable meta-language (like Haskell) we can write a function that creates a set with n elements (for any natural number n) as a term in FOL. Here is some pseudo-code defining the "von Neumann" encoding:

$$\begin{aligned} vN\ 0 &= \{\} \\ vN\ (n+1) &= step\ (vN\ n) \\ step\ x &= Union\ x\ (S\ x) \end{aligned}$$

If we use conventional set notation we get $vN\ 0 = \{\}$, $vN\ 1 = \{\{\}\}$, $vN\ 2 = \{\{\}, \{\{\}\}\}$, $vN\ 3 = \{\{\}, \{\{\}\}, \{\{\}, \{\{\}\}\}\}$, etc. If we use the shorthand $\overline{n}$ for $vN\ n$ we see that $\overline{0} = \{\}$, $\overline{1} = \{\overline{0}\}$, $\overline{2} = \{\overline{0}, \overline{1}\}$, $\overline{3} = \{\overline{0}, \overline{1}, \overline{2}\}$ and, in general, that $\overline{n}$ has cardinality n (meaning it has n elements).

Pairs The constructions presented so far show that, even starting from no elements, we can embed all natural numbers in pure set theory. We can also embed unordered pairs: $\{a, b\} \stackrel{\text{def}}{=} Union\ (S\ a)\ (S\ b)$ and normal, ordered pairs: $(a, b) \stackrel{\text{def}}{=} \{S\ a, \{a, b\}\}$. With a bit more machinery it is possible to step by step encode $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$. A good read in this direction is "The Haskell Road to Logic, Maths and Programming" [Doets and van Eijck, 2004].

2.2 Propositional Calculus

Our first DSL for this chapter is the language of *propositional calculus* (or propositional logic), modelling simple propositions with the usual combinators for and, or, implies, etc. When reading a logic book, one will encounter several concrete syntactic constructs related to propositional logic, which are collected in Table 2.1. Each row lists common synonyms and their arity (number of arguments).

[1] The *cardinality* of a set is the number of elements in it.

False	$\bot$	F	nullary
True	$\top$	T	nullary
Not	$\neg$	$\sim$	unary
And	$\wedge$	&	binary
Or	$\vee$	$\mid$	binary
Implies	$\supset$	$\Rightarrow$	binary

Table 2.1: Syntax for propositions. In addition, a, b, c, ... are used as names of propositions

Some example propositions will include $p_1 = a \wedge (\neg a)$, $p_2 = a \Rightarrow b$, $p_3 = a \vee (\neg a)$, $p_4 = (a \wedge b) \Rightarrow (b \wedge a)$. The names a, b, c, ... are "propositional variables": they can be substituted for any proposition. We could call them "variables", but in upcoming sections we will add another kind of variables (and quantification over them) to the calculus — so we keep calling them "names" to avoid mixing them up.

Just as we did with simple arithmetic, and with complex number expressions in Chapter 1, we can model the abstract syntax of propositions as a datatype:

```
data Prop = Con     Bool
          | Not     Prop
          | And     Prop Prop
          | Or      Prop Prop
          | Implies Prop Prop
          | Name    Name
type Name = String
```

The example expressions can then be expressed as

```
p1 = And (Name "a") (Not (Name "a"))
p2 = Implies (Name "a") (Name "b")
p3 = Or (Name "a") (Not (Name "a"))
p4 = Implies (And a b) (And b a)
  where a = Name "a"; b = Name "b"
```

Because "names" stand for propositions, if we assign truth values for the names, we can compute a truth value of the whole proposition for the assignment in question.

2.2.1 An Evaluator for *Prop*

Let us formalise this in general, by writing an evaluator which takes a *Prop* term to its truth value.

(The evaluation function for a DSL describing a logic is often called *check* instead of *eval* but for consistency we stick to *eval*.)

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Prop} \to (\mathit{Name} \to \mathit{Bool}) \to \mathit{Bool}\\
&\mathit{eval}\ (\mathit{Implies}\ p\ q)\ \mathit{env} = \mathit{eval}\ p\ \mathit{env}\ `\mathit{implies}`\ \mathit{eval}\ q\ \mathit{env}\\
&\mathit{eval}\ (\mathit{And}\ p\ q)\ \ \ \mathit{env} = \mathit{eval}\ p\ \mathit{env} \wedge \mathit{eval}\ q\ \mathit{env}\\
&\mathit{eval}\ (\mathit{Or}\ p\ q)\ \ \ \ \mathit{env} = \mathit{eval}\ p\ \mathit{env} \vee \mathit{eval}\ q\ \mathit{env}\\
&\mathit{eval}\ (\mathit{Not}\ p)\ \ \ \ \ \mathit{env} = \neg\ (\mathit{eval}\ p\ \mathit{env})\\
&\mathit{eval}\ (\mathit{Name}\ n)\ \ \ \mathit{env} = \mathit{env}\ n\\
&\mathit{eval}\ (\mathit{Con}\ t)\ \ \ \ \ \mathit{env} = t
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{implies} :: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool}\\
&\mathit{implies}\ \mathit{False}\ \_ = \mathit{True}\\
&\mathit{implies}\ \_\ \ \ \ p = p
\end{aligned}
$$

The function *eval* translates from the syntactic domain to the semantic domain, given an environment (an assignment of names to truth values), which we represent as a function from each *Name* to *Bool*. . Here *Prop* is the (abstract) *syntax* of the language of propositional calculus and *Bool* is the *semantic domain*, and $\mathit{Name} \to \mathit{Bool}$ is a necessary extra parameter to write the function. Alternatively, and perhaps more elegantly, we can view $(\mathit{Name} \to \mathit{Bool}) \to \mathit{Bool}$ as the semantic domain.

2.2.2 Truth tables and tautologies

As a first example of a truth table, consider the proposition $F \Rightarrow a$ which we call t here. The truth table semantics of t is usually drawn as in Fig. 2.1: one column for the name a listing all combinations of T and F, and one column for the result of evaluating the expression. This table shows that no matter what value assignment we try for the only variable a, the semantic value is $T = \mathit{True}$. Thus the whole expression could be simplified to just T without changing its semantics.

a	t
F	T
T	T

Figure 2.1: $F \Rightarrow a$

TODO: PJ: make different columns use different background colours

If we continue with the example p_4 from above we have two names a and b which together can have any of four combinations of true and false. After the name-columns are filled, we fill in the rest of the table one operation (column) at a time (see Fig. 2.2). The & columns become $F\ F\ F\ T$ and finally the $\Rightarrow$ column (the output) becomes true everywhere. For our other examples, p_1 is always false, p_2 is mixed and p_3 is always true.

a	&	b	$\Rightarrow$	b	&	a
F	F	F	T	F	F	F
F	F	T	T	T	F	F
T	F	F	T	F	F	T
T	T	T	T	T	T	T

Figure 2.2: $p_4 = (a \wedge b) \Rightarrow (b \wedge a)$.

A proposition whose truth table output is constantly true is called a *tautology*. Thus t, p_3 and p_4 are tautologies. We can formalise this idea as the following tautology-tester:

$$
\begin{aligned}
&\mathit{isTautology} :: \mathit{Prop} \to \mathit{Bool}\\
&\mathit{isTautology}\ p = \mathit{and}\ [\mathit{eval}\ p\ e \mid e \leftarrow \mathit{envs}\ (\mathit{freeNames}\ p)]
\end{aligned}
$$

which uses the following intermediate function to generate all possible environments for a given list of names

TODO: PJ: Haskell notation for list comprehensions not introduced

$$
\begin{aligned}
&\mathit{envs} :: [\mathit{Name}] \to [\mathit{Name} \to \mathit{Bool}] \\
&\mathit{envs}\ (n : ns) = [\ \lambda n' \to \textbf{if}\ n == n'\ \textbf{then}\ b\ \textbf{else}\ e\ n' \\
&\qquad\qquad\qquad\quad \mid b \leftarrow [\mathit{True}, \mathit{False}] \\
&\qquad\qquad\qquad\quad , e \leftarrow \mathit{envs}\ ns \\
&\qquad\qquad\qquad\quad]
\end{aligned}
$$

and a function to find all names in a proposition:

$$
\begin{aligned}
&\mathit{freeNames} :: \mathit{Prop} \to [\mathit{Name}] \\
&\mathit{freeNames} = \mathit{error}\ \texttt{"exercise"}
\end{aligned}
$$

Truth table verification is only viable for propositions with few names because of the exponential growth in the number of cases to check: we get 2^n different *truthTables* for n names.

Exercise 2.1. Define the function *freeNames*.

There are much better algorithms to evaluate truth values than the naive one we just showed, but we will not go this route. Rather, we can introduce the notion of *proof*. (And in fact, the best (known) algorithms remain exponential in the number of variables.)

2.2.3 Proofs for Propositional Logic

Given a *Prop*osition p and a proof t (represented as an element of another type *Proof*), we can write a function that checks that t is a valid proof of p:

$$\mathit{checkProof} :: \mathit{Proof} \to \mathit{Prop} \to \mathit{Bool}$$

But we still have to figure out what consitutes proofs.

To prove *And P Q*, one needs simultaneously of P and a proof of Q. (In logic texts, one will often find

$$\frac{P \quad Q}{P \wedge Q}$$

to represent this fact, which is called the *introduction rule for* $(\wedge)$) (For the proof to be complete, one still needs to provide a full proof of P and another for Q — it is not enough to just invoke this rule.)

Therefore, in Haskell, can represent this rule by a proof-term constructor *AndIntro* with two *Proof* arguments:

$$\mathit{AndIntro} :: \mathit{Proof} \to \mathit{Proof} \to \mathit{Proof}$$

and, the corresponding case of the *checkProof* function will look like this:

$$checkProof\ (AndIntro\ t\ u)\ (And\ p\ q) = checkProof\ t\ p \wedge checkProof\ u\ q$$

To prove $Or\ P\ Q$, we need either a proof of P or proof of Q — but we need to know which side (*Left* for p or *Right* for q) we refer to. Therefore, we need a proof-term constructor:

$$OrIntro :: Either\ Proof\ Proof \rightarrow Proof$$

For reference, the either type is defined as follows in the Haskell prelude:

```
data Either p q where
  Left  :: p → Either p q
  Right :: q → Either p q
```

There are a couple of possible approaches to deal with negation. One approach is to define it as de Morgan dualisation:

$$Not\ (a\ `Or`\ b) = Not\ a\ `And`\ Not\ b$$
$$Not\ (a\ `And`\ b) = Not\ a\ `Or`\ Not\ b$$
$$...$$

Negation can then only apply to names, which can recieve a special treatment in proof-checking.

However, we will instead apply the same treatment to negation as to other constructions, and define a suitable introduction rule:

$$\frac{P \rightarrow Q \quad P \rightarrow \neg Q}{\neg P}$$

. We can represent it by the $NegIntro :: Prop \rightarrow Proof \rightarrow Proof \rightarrow Proof$ constructor.

Because we have inductive proofs (described from the bottom up), we have the additional difficulty that this rule conjures-up a new proposition, Q. This is why we need an additional *Prop* argument, which gives the Q formula.

There is no way to introduce Falsity ($\bot$), otherwise we'd have an inconsistent logic! Finally we can introduce "Truth", with no premiss: $\top$. The proof has no information either $TruthIntro :: Proof$.

To complete the system, in addition to introduction rules (where the connective appears as conclusion), we also need elimination rules (where the connective appears as premiss). For conjuction, we have two eliminations rules: $\frac{P \wedge Q}{P}$ and $\frac{P \wedge Q}{Q}$. So we represent them by $AndElim1 :: Proof \rightarrow Prop \rightarrow Proof$ (and *AndElim2* symmetrically), where the extra *Prop* argument corresponds to Q.

Our elimination rule for disjunction is $\frac{P \vee Q \quad P \to R \quad Q \to R}{R}$. The idea here is that if we know that $P \vee Q$ holds, then we have two cases: either P holds and Q holds. If only we can find a proposition R which is a consequence of both P and Q, then, regardless of which case we are facing, we know that R will hold.

Our elimination for negation is $\frac{\neg\neg P}{P}$. It simply says that two negations cancel out.

Finally we can eliminate Falsity as follows: $\frac{\bot}{P}$. This rule goes some times by its descriptive latin name *ex falso quodlibet* — from falsehood, anything (follows).

We can then write our proof checker as follows:

```
checkProof TruthIntro          (Con True)  = True
checkProof (AndIntro t u)      (And p q)   = checkProof t p
                                           ∧ checkProof u q
checkProof (OrIntro (Left t))  (Or p q)    = checkProof t p
checkProof (OrIntro (Right t)) (Or p q)    = checkProof t q
checkProof (NotIntro t u q)    (Not p)     = checkProof t (p `Implies` q)
                                           ∧ checkProof u (p `Implies` Not q)
checkProof (AndElimL t q)      p = checkProof t (p `And` q)
checkProof (AndElimR t p)      q = checkProof t (p `And` q)
checkProof (OrElim t u v p q)  r = checkProof t (p `Implies` r)
                                 ∧ checkProof u (q `Implies` r)
                                 ∧ checkProof v (Or p q)
checkProof (NotElim t)         p = checkProof t (Not (Not p))
checkProof (FalseElim t)       p = checkProof t (Con False)
```

Any other combination of proof/prop else is an incorrect combination: the proof is not valid for the proposition.

```
checkProof _ _ = False   -- incorrect proof
```

At this point it can be interesting to, see *checkProof* as an evaluator again, by, and flipping its arguments: $flip\ checkProof :: Prop \to (Proof \to Bool)$. This way, one can understand $Proof \to Bool$, a subset of proofs, as the semantic domain of *Prop*. In other words, a proposition can be interpreted at the subset of proofs which prove it.

2.2.4 Implication, hypothetical derivations, contexts

We have so far omitted to deal with *Implies*. One reason is that we can use the so-called material implication definition which we invoked earlier in truth tables. It means to define $Implies\ a\ b = (Not\ a)\ `Or`\ b$ — and this equality means that there is no need to deal specially with *Implies*. However this approach does not bring any new insight. In particular, this view is hard to transpose to more complicated logics (such as second-order logic).

Thus we take our usual approach and give rules for it. The introduction rule is sometimes written like so in logic texts:

$$\frac{\begin{array}{c} P \\ \vdots \\ Q \end{array}}{P \to Q}$$

Such a notation can however be terribly confusing. We were used to the fact that proofs above the line had to be continued, so what can the dots possibly mean? The intended meaning of this notation is that, to prove that $P \to Q$, it suffices to prove Q, but one is also allowed to use P as an assumption in this (local) proof of Q.

We can use our DSL to formalise this rule as Haskell data, by adding a constructor corresponding to implication introduction: $\mathit{ImplyIntro} :: (\mathit{Proof} \to \mathit{Proof}) \to \mathit{Proof}$. The fact that the premiss can depend on the assumption Q is represented by a function whose parameter is the proof of Q in question. In other words, to prove the formula $P \to Q$ we assume a proof t of P and derive a proof u of Q. So, a proof of an implication is a function from proofs to proofs.

The eliminator for implication (also known as the hard-to-translate phrase *modus ponens*) is $\frac{P \to Q \quad P}{Q}$. We formalise it as $\mathit{ImplyElim} :: \mathit{Proof} \to \mathit{Proof} \to \mathit{Prop} \to \mathit{Proof}$ (The proposition P is a not given by the conclusion and thus has to be provided as part of the proof.). And we can finally complete our proof checker as follows:

```
checkProof (Assumption p') p = p == p'
checkProof (ImplyIntro f) (p `Implies` q) = checkProof (f (Assumption p)) q
checkProof (ImplyElim t u p) q = checkProof t (q `Implies` p) ∧ checkProof u q
checkProof _ _ = False   -- incorrect proof
```

And, for reference, the complete DSL for proofs is given by the following datatype:

```
data Proof = Assumption Prop
           | TruthIntro
           | FalseElim Proof
           | AndIntro Proof Proof
           | AndElimL Proof Prop
           | AndElimR Proof Prop
           | OrIntro (Either Proof Proof)
           | OrElim Proof Proof Proof Prop Prop
           | NotIntro Proof Proof Prop
           | NotElim Proof
           | ImplyIntro (Proof → Proof)
           | ImplyElim Proof Proof Prop
```

Aside The *Assumption* constructor may make the reader somewhat uneasy: how come that we can simply assume anything? The intent is that this constructor is *private* to the *checkProof* function (or module). No user-defined proof can use it. The most worried readers can also define the following version of *checkProof*, which uses an extra context to check that assumption have been rightfully introduced earlier. [2]

```
checkProof :: Context → Proof → Prop → Bool
checkProof ctx (ImplyIntro t) (p `Implies` q) = checkProof' (p : ctx) t q
checkProof ctx Assumption p = p ∈ ctx
```

Example proof We can put our proof-checker to the test, by writing a number of proofs and verifying them.

```
conjunctionCommutative :: Prop
conjunctionCommutative = (p `And` q) `Implies` (q `And` p)
  where p = Name "p"; q = Name "q"
conjunctionCommutativeProof :: Proof
conjunctionCommutativeProof =
  ImplyIntro (λevPQ →
  AndIntro (AndElimR evPQ (Name "p"))
           (AndElimL evPQ (Name "q")))
```

(where *evPQ* stands for "evidence for P and Q.")

We can then run the checker and verify: *checkProof conjunctionCommutativeProof conjunctionCommutative* == *True*

Exercise 2.2. Try to swap *AndElimL* and *AndElimR* in the above proof. What will happen and why?

2.2.5 The Haskell type-checker as a proof checker

Perhaps surprisingly, the proof-checker that we just wrote is already built-in in the Haskell compiler. Let us clarify what we mean, using the same example, but adapt it to let the type-checker do the work:

```
conjunctionCommutativeProof' :: (p `And` q) `Implies` (q `And` p)
conjunctionCommutativeProof' =
  implyIntro (λevPQ →
  andIntro (andElimR evPQ)
           (andElimL evPQ))
```

[2] This kind of presentation of the checker matches well the sequent calculus presentation of the proof system.

That is, instead of writing propositions, we write types (*And*, *Or*, *Implies*). Instead of using *Proof* constructors, we use functions whose types capture rules:

$$
\begin{aligned}
&\mathit{truthIntro} &&:: \mathit{Truth}\\
&\mathit{falseElim} &&:: \mathit{False} \to p\\
&\mathit{andIntro} &&:: p \to q \to \mathit{And}\ p\ q\\
&\mathit{andElimL} &&:: a\ \text{`}\mathit{And}\text{`}\ b \to a\\
&\mathit{andElimR} &&:: a\ \text{`}\mathit{And}\text{`}\ b \to b\\
&\mathit{orIntro} &&:: \mathit{Either}\ a\ b \to \mathit{Or}\ a\ b\\
&\mathit{orElim} &&:: \mathit{Or}\ p\ q \to (p\ \text{`}\mathit{Implies}\text{`}\ r) \to (q\ \text{`}\mathit{Implies}\text{`}\ r) \to r\\
&\mathit{notIntro} &&:: (p\ \text{`}\mathit{Implies}\text{`}\ q)\ \text{`}\mathit{And}\text{`}\ (p\ \text{`}\mathit{Implies}\text{`}\ \mathit{Not}\ q) \to \mathit{Not}\ p\\
&\mathit{notElim} &&:: \mathit{Not}\ (\mathit{Not}\ p) \to p\\
&\mathit{implyIntro} &&:: (p \to q) \to (p\ \text{`}\mathit{Implies}\text{`}\ q)\\
&\mathit{implyElim} &&:: (p\ \text{`}\mathit{Implies}\text{`}\ q) \to p \to q
\end{aligned}
$$

Instead of running *checkProof*, we type-check the above program. Because the proof is correct, we get no type-error.

Exercise 2.3. What would happen if you swap *andElimR* and *andElimL*? Why?

This style of propositional logic proof is very advantageous, because we not only the checker comes for free, but we additionally get all the engineering tools of the Haskell tool-chain.

One should be careful however that Haskell is not design with theorem-proving in mind. For this reason it is easily possible make the compiler accept invalid proofs. The main two sources of invalid proofs are 1. non-terminating programs and 2. exception-raising programs.

2.2.6 Intuitionistic Propositional Logic and Simply Typed Lambda-Calculus, Curry-Howard isomorphism.

We can make the link beween Haskell and logic more tight if we restrict ourselves to *intuitionistic* logic.

One way to characterize intuitionistic logic is that it lacks native support for negation. Instead, *Not p* is represented as *p* ‘*Implies*‘ *False*:

$$\textbf{type}\ \mathit{Not}\ p = p\ \text{`}\mathit{Implies}\text{`}\ \mathit{False}$$

The intuition behind this definition is the principle of proof by contradiction: if assuming *p* leads to a contradition (*False*), then *p* must be false; so *Not p* should hold.

When doing this kind of definition, one gives up on *notElim*: there is no way to eliminate (double) negation.

$$\mathit{notElim} = \mathit{error}\ \texttt{"not possible as such in intuitionistic logic"}$$

On the other hand the introduction for negation becomes a theorem of the logic. The formulation of the theorem is:

$$notIntro :: (p \text{ `Implies` } q) \text{ `And` } (p \text{ `Implies` } (q \text{ `Implies` } False)) \to p \text{ `Implies` } False$$

And its proof is:

$$notIntro\ (evPimpliesQ, evPimpliesNotQ)\ evP = \\ (evPimpliesNotQ \text{ `implyElim` } evP) \text{ `implyElim` } (evPimpliesQ \text{ `implyElim` } evP)$$

By focusing on intuitionistic logic, we can give a *typed* representation for each of the formula constructors. Let us consider implication first. *impIntro* and *impElim* seem to be conversion from and to functions, and so it should be obvious that the representation of the implication formula is a function:

$$\textbf{type } Implies\ p\ q = p \to q \\ implyElim\ f\ x = f\ x \\ implyIntro\ f\ x = f\ x$$

Conjunction is represented as pairs; that is, if $p : P$ and $q : Q$ then the proof of $And\ P\ Q$ should be a pair of p and q. The elimination rules are projections. In code:

$$\textbf{type } And\ p\ q = (p, q) \\ andIntro\ t\ u = (t, u) \\ andElimL = fst \\ andElimR = snd$$

Similarly, disjuction is represented as *Either*: if $p : P$ then $Left\ p : Or\ P\ Q$ and if $q : Q$ then $Right\ q : Or\ P\ Q$.

$$\textbf{type } Or\ a\ b = Either\ a\ b \\ orIntro\ x = x \\ orElim\ (Left\ \ t)\ u\ \_ = u\ t \\ orElim\ (Right\ t)\ \_\ v = v\ t$$

We already had characterize or-elimination as case analysis, and, indeed, this is how we implement it.

Truth is represented as the unit type:

$$\textbf{type } Truth = () \\ truthIntro = ()$$

And falsehood is represented as the *empty* type (with no constructor):

$$\textbf{data } False \\ falseElim\ x = \textbf{case } x \textbf{ of } \{\ \}$$

Note that the case-analysis has nothing to take care of here.

In this way we can build proofs ("proof terms") for all of intuitionistic propositional logic (IPL). As we have seen, each such proof term is a program in Haskell. Conversely, every program written in this fragment of Haskell (functions, pairs, *Either*, no recursion and full coverage of cases) can be turned into a proof in IPL. This fragment is called the simply-typed lambda calculus (STLC) with sum and products.

The law of the excluded middle As an example of how intuitionism twists usual law, consider the law of the excluded middle, which states that, for any proposition P, either P or $Not\ P$ holds. For example, either it rains or it does not rain. There is no "middle compromise". If we attempt to prove $P\ `Or`\ (Not\ P)$ in intuitionitic logic, we quickly find ourselves in a dead end. Clearly, we cannot prove P for any P. Likewise $Not\ P$, or equivalently $P \to False$ cannot be deduced.

What we have to do account for the fact that we cannot use negation elimination, and so we have to make-do with proving $Not\ (Not\ Q)$ instead of Q. This is exactly what we have to do to prove the law of excluded middle. We can then provide this Haskell-encoded proof:

```
excludedMiddle :: Not (Not (p `Or` Not p))
  -- to prove this, we can ...
excludedMiddle k =    -- ... assume Not (p `Or` (Not p)) and prove falsity.
  k  -- So, we can prove falsity if we can prove p `Or` (Not p).
  (Right    -- We can prove in particular the right case, Not p
    (λevP →    -- ... by assuming that p holds, and prove falsity.
      k  -- But again, we can prove falsity if we can prove p `Or` (Not p).
      (Left    -- This time, we can prove in particular the left case, p ...
        evP)))   -- because we assumed it earlier!
```

Revisiting the tupling transform In Exercise 1.8, the "tupling transform" was introduced, relating a pair of functions to a function returning a pair. (Revisit that exercise if you skipped it before.) There is a logic formula corresponding to the type of the tupling transform:

$$(a\ `Implies`\ (b\ `And`\ c))\ `Iff`\ (a `Implies` b)\ `And`\ (a\ `Implies`\ c)$$

(*Iff* refers to implication in both directions). The proof of this formula closely follows the implementation of the transform. Therefore we start with the two directions of the transform as functions:

```
test1' :: (a → (b, c)) → (a → b, a → c)
test1' = λa2bc → (λa → fst (a2bc a)
```

```
                    , λa → snd (a2bc a))
test2' :: (a → b, a → c) → (a → (b, c))
test2' = λfg → λa → (fst fg a, snd fg a)
```

Then we move on to the corresponding logic statements with proofs. Note how the functions are "hidden inside" the proof.

```
test1 :: Implies (Implies a (And b c)) (And (Implies a b) (Implies a c))
test1 = implyIntro (λa2bc →
          andIntro (implyIntro (λa → andElimL (implyElim a2bc a)))
                   (implyIntro (λa → andElimR (implyElim a2bc a))))
test2 :: Implies (And (Implies a b) (Implies a c)) (Implies a (And b c))
test2 = implyIntro (λfg →
          implyIntro (λa →
            andIntro (implyElim (andElimL fg) a)
                     (implyElim (andElimR fg) a)))
```

Logic as impoverished typing rules Another view of the same isomorphism is that the logical rules for IPL can be obtained by erasing programs from the typing rules for STLC. We will show here only the application rule, leaving the rest as exercise. This typing rule for function application can be written as follows:

$$\frac{f : A \to B \quad x : A}{f(x) : B}$$

After erasing the colon (:) sign and what comes before it, we obtain *modus ponens* — implication elimination.

The *Curry–Howard correspondence* is a general principle that says that we can think of propositions as types, and proofs as programs. This principle goes beyond propositional logic (and first order logic, etc.): it applies to all sorts of logics and programming languages, with various levels of expressivity and features.

2.2.7 *Or* is the dual of *And*.

Before moving on to our next topic, we make a final remark on *And* and *Or*. Most of the properties of *And* have corresponding properties for *Or*. This can be explained one way by observing that they are de Morgan duals. Another explanatino is that one can swap the direction of the arrows in the types of the the role between introduction and elimination. (Using our presentation, doing so requires applying isomorphisms.)

```
module FOL where
```

2.3 First Order Logic

Our second DSL is that of *First Order Logic*, or FOL for short, and also known as Predicate Logic.

Compared to propositional logic, the main addition is *quantification over individuals*. Additionally, one adds a language of terms — its semantic domain being the individuals which we quantify over.

Let us study terms first. A *term* is either a (term) *variable* (like x, y, z), or the application of a *function symbol* (like f, g) to a suitable number of terms. If we have the function symbols f of arity 2 and g of arity 3 we can form terms like $f\ (x,x)$, $g\ (y,z,z)$, $g\ (x,y,f\ (x,y))$, etc. The individuals are often limited to a single domain. For example here we will take individuals to be rationals. Consequently, the actual function symbols are also domain-specific — for rationals we will have addition, division, etc. In this case we can model the terms as a datatype: TODO: JP: why?

$$\begin{array}{l}\textbf{type}\ \mathit{VarT} = \mathit{String}\\ \textbf{data}\ \mathit{RatT} = \mathit{RV}\ \mathit{VarT} \mid \mathit{FromI}\ \mathit{Integer} \mid \mathit{RPlus}\ \mathit{RatT}\ \mathit{RatT} \mid \mathit{RDiv}\ \mathit{RatT}\ \mathit{RatT}\\ \quad \textbf{deriving}\ \mathit{Show}\end{array}$$

The above introduces variables (with the constructor *RV*) and three function symbols: *FromI* of arity 1, *RPlus*, *RDiv* of arity 2.

Exercise 2.4. Following the usual pattern, write the evaluator for *RatT*:

$$\begin{array}{l}\mathit{evalRat} :: \mathit{RatT} \to (\mathit{VarT} \to \mathit{RatSem}) \to \mathit{RatSem}\\ \textbf{type}\ \mathit{RatSem} = \mathit{Rational}\end{array}$$

As mentioned above, the propositions (often referred to as *formulas* in the context of FOL) are extended so that they can refer to terms. That is, the names from the propositional calculus are generalised to *predicate symbols* of different arity. The predicate symbols can only be applied to terms, not to other predicate symbols or formulas. If we have the predicate symbols *New* of arity 0, $\mathbb{N}_{>0}$ of arity 1 and *Less* of arity 2 we can form *formulas* like *New*, $\mathbb{N}_{>0}\ (x)$, *Less* $(f\ (x,x),y)$, etc. Note that we have two separate layers, with terms at the bottom: formulas normally refer to terms, but terms cannot refer to formulas.

The formulas introduced so far are all *atomic formulas*: generalisations of the *names* from *PropCalc*. Now we will add two more concepts: first the logical connectives from the propositional calculus: *And*, *Or*, *Implies*, *Not*, and then two quantifiers: "forall" ($\forall$) and "exists" ($\exists$).

Thus the following is an example FOL formula:

$$\forall\, x.\ \mathbb{N}_{>0}\ (x) \Rightarrow (\exists\, y.\ \mathit{Less}\ (f\ (x,x),y))$$

The fact that quantification is over individuals is a defining characteristic of FOL. If one were to, say, quantify over predicates, we'd have a higher-order logic, with completely different properties.

As another example, we can write a formula stating that the function symbol *plus* is commutative:

$$\forall\, x.\ \forall\, y.\ \mathit{Eq}\ (\mathit{plus}\ (x, y), \mathit{plus}\ (y, x))$$

Here is the same formula with infix operators:

$$\forall\, x.\ \forall\, y.\ (x + y) == (y + x)$$

Note that $==$ is a binary predicate symbol (written *Eq* above), while $+$ is a binary function symbol (written *plus* above).

As before we can model the expression syntax (for FOL, in this case) as a datatype. We keep the logical connectives *And*, *Or*, *Implies*, *Not* from the type *PropCalc*, add predicates over terms, and quantification. The constructor *Equal* could be eliminated in favour of *PName* `"Eq"` but it is often included as a separate constructor.

```
type PSym = String
data FOL = PName PSym [RatT]
         | Equal RatT RatT
         | And     FOL FOL
         | Or      FOL FOL
         | Implies FOL FOL
         | Not     FOL
         | FORALL VarT FOL
         | EXISTS  VarT FOL
  deriving Show
```

2.3.1 Evaluator for Formulas and ∗Undecidability

Setting us up for failure, let us attempt to write an *eval* function for FOL, as we did for propositional logic.

In propositional logic, we allowed the interpretation of propositional variables to change depending on the environment. Here, we will let the interpretation of term variables be dependent on an environment, which will therefore map (term) variables to individuals ($\mathit{VarT} \to \mathit{RatSem}$). If we so wished, we could have an environment for the interpretation of predicate names, with an environment of type $\mathit{PSym} \to [\mathit{RatSem}] \to \mathit{Bool}$. Rather, with little loss of generality, we will fix this interpretation, via a constant function *eval0*, which may look like this:

$$
\begin{aligned}
&\mathit{eval0} :: \mathit{PSym} \to [\mathit{RatSem}] \to \mathit{Bool} \\
&\mathit{eval0}\ \texttt{"Equal"}\ [t_1, t_2] = t_1 == t_2 \\
&\mathit{eval0}\ \texttt{"LessThan"}\ [t_1, t_2] = t_1 < t_2 \\
&\mathit{eval0}\ \texttt{"Positive"}\ [t_1] = t_1 > 0
\end{aligned}
$$

etc.

the environment which was mapping names to truth values is replaced by an environment mapping each predicate/argument combinations to a truth value: $(\mathit{PSym} \to [\mathit{RatSem}] \to \mathit{Bool})$. Additionally we need en evironment mapping So we would use the following type:

$$\mathit{eval} :: \mathit{FOL} \to (\mathit{VarT} \to \mathit{RatSem}) \to \mathit{Bool}$$

And go our merry way for most cases:

$$
\begin{aligned}
&\mathit{eval\ formula\ ratEnv} = \textbf{case}\ \mathit{formula}\ \textbf{of} \\
&\quad (\mathit{PName\ n\ args}) \to \mathit{eval0\ n\ (map\ (flip\ evalRat\ ratEnv)\ args)} \\
&\quad \mathit{Equal\ a\ b} \to \mathit{evalRat\ a\ ratEnv} == \mathit{evalRat\ b\ ratEnv} \\
&\quad \mathit{And\ p\ q} \to \mathit{eval\ p\ ratEnv} \wedge \mathit{eval\ p\ ratEnv} \\
&\quad \mathit{Or\ p\ q} \to \mathit{eval\ p\ ratEnv} \vee \mathit{eval\ p\ ratEnv}
\end{aligned}
$$

etc.

However, as soon as we encounter quantifiers, we have a problem. To evaluate *EXISTS x p* (at least in certain contexts) we may need to evaluate p for each possible value of x. But, unfortunately, there are infinitely many such possible values, and so we can never know if the formula is a tautology.[3] So, if we were to try to run the evaluator, it would not terminate. Hence, the best that we can ever do is, given a hand-written proof of the formula, check if the proof is valid. Fortunately, we have already studied the notion of proof in the section on propositional logic, and it only needs to be extended to support quantifiers.

2.3.2 Universal quantification

Universal quantification (Forall or $\forall$) can be seen as a generalisation of *And*. To see this, we can begin by generalising the binary operator *And* to an n-ary version: And_n. To prove $\mathit{And}_n\ (A_1, A_2, ..., A_n)$ we need a proof of each A_i. Thus we could define $\mathit{And}_n\ (A_1, A_2, ..., A_n) = A_1\ \&\ A_2\ \&\ ...\ \&\ A_n$ where & is the infix version of binary *And*. The next step is to require the formulas A_i to be of the same form, ie. the result of applying a constant function A to the individual i.

[3] FOL experts will scoff at this view, because they routinely use much more sophisticated methods of evaluation, which handle quantifiers in completely different ways. Their methods are even able to identify tautologies as such. However, even such methods are not guaranteed to terminate on formulas which are not tautologies. Therefore, as long as an even-very-advanced FOL tautology-checker is running, there is no way to know how close it is to confirming if the formula at hand is a tautology or not. This is not a technical limitation, but rather a fundamental one, which boils down to the presence of quantifier over an infinite domain.

And, we can think of the variable i ranging over the full set of individuals i_1, i_2, ... Then the final step is to introduce the notation $\forall\ i.\ A\ (i)$ for $A\ (i_1)$ & $A\ (i_2)$ & ....

Now, a proof of $\forall\ x.\ A\ (x)$ should in some way contain a proof of $A\ (x)$ for every possible x. For the binary *And* we simply provide the two proofs, but in the infinite case, we need an infinite collection of proofs. To do so, a possible procedure is to introduce a fresh (meaning that we know nothing about this new term) constant term a and prove $A\ (a)$. Intuitively, if we can show $A\ (a)$ without knowing anything about a, we have proved $\forall\ x.\ A\ (x)$. Another way to view this is to say that a proof of $\forall\ x.\ P\ x$ is a function f from individuals to proofs such that $f\ t$ is a proof of $P\ t$ for each term t.

So we can now extend our type for proofs: the introduction rule for universal quantification is $\frac{A(x) \quad x \text{ fresh}}{\forall x.A(x)}$. The corresponding constructor can be $\mathit{UniversalIntro} :: (\mathit{RatSem} \to \mathit{Proof}) \to \mathit{Proof}$.

Note that the scoping rule for $\forall\ x.\ b$ is similar to the rule for a function definition, $f\ x = b$, and for anonymous functions, $\lambda x \to b$. Just as in those cases we say that the variable x is *bound* in b and that the *scope* of the variable binding extends until the end of b (but not further).

One common source of confusion in mathematical (and other semi-formal) texts is that variable binding sometimes is implicit. A typical example is the notation for equations: for instance $x\hat{}2 + 2 * x + 1 == 0$ usually means roughly $\exists\ x.\ \ x\hat{}2 + 2 * x + 1 == 0$. We write "roughly" here because in math texts the scope of x very often extends to some text after the equation where something more is said about the solution x. [4]

Let us now consider the elimination rule for universal quantification. The idea here is that if $A\ (x)$ holds for every abstract individual x, then it also holds for any concrete individual a: $\frac{\forall x.A(x)}{A(a)}$. As for *And* we had to provide the other argument to recover p '*And*' q, here we have to be able reconstruct the general form $A\ (x)$ — indeed, it is not simply a matter of substituting x for a, because there can be several occurences of a in the formula to prove. So, in fact, the proof constructor must contain the general form $A\ (x)$, for example as a function from individuals: $\mathit{UniversalElim} :: (\mathit{RatSem} \to \mathit{Prop}) \to \mathit{Proof} \to \mathit{Proof}$.

Let us sketch the proof-checker cases corresponding to universal quantification:

$$\begin{aligned}
&\mathit{proofChecker}\ (\mathit{UniversalIntro}\ f\ a)\ (\forall\ x.\ p) = \mathit{proofChecker}\ (f\ a)\ (\mathit{subst}\ x\ a\ p)\\
&\quad \textbf{where}\ a = \mathit{freshFor}\ [a, \forall\ x.\ p]\\
&\mathit{proofChecker}\ (\mathit{UniversalElim}\ f\ t)\ p = \mathit{checkUnify}\ (f\ x)\ p \wedge \mathit{proofChecker}\ t\ (\forall\ x.\ f\ x)\\
&\quad \textbf{where}\ x = \mathit{freshFor}\ [f\ x, p]
\end{aligned}$$

[4] This phenomena seems to be borrowed the behaviour of quantifiers in natural language. See for example [Bernardy and Chatzikyriakidis, 2020] for a discussion.

The introduction rule uses a new concept: *subst x a p*, which replaces the variable x by a in p, but otherwise follows closely our informal explanation. The eliminator uses *checkUnify* which verifies that $f\ x$ is indeed a generalisation of the formula to prove, p. Finally we need a way to introduce fresh variables *freshFor*, which conjures up a variable occuring nowhere (in its argument).

2.3.3 Existential quantification

We have already seen how the universal quantifier can be seen as a generalisation of *And* and in the same way we can see the existential ($\exists$) quantifier as a generalisation of *Or*.

First we generalise the binary *Or* to an n-ary Or_n. To prove $Or_n\ A_1\ A_2\ ...\ A_n$ it is enough (and necessary) to find one i for which we can prove A_i. As before we then take the step from a family of formulas A_i to a single unary predicate A expressing the formulas $A\ (i)$ for the (term) variable i. Then the final step is to take the disjunction of this infinite set of formulas to obtain $\exists\ i.\ A\ i$.

The elimination and introduction rules for existential quantification are:

$$\frac{P(a)}{\exists i.P(i)} \qquad\qquad \frac{\text{for every } a,\ P(a) \to R \qquad \exists i.A(i)}{R}$$

The introduction rule says that to prove the existential quantification, we only need exhibit one witness (a) and one proof for that member of set of individuals. For binary *Or* the "family" only had two members, one labelled *Left* and *Right*, and we effectively had one introduction rule for each. Here, for the generalisation of *Or*, we have unified the two rules into one with an added parameter a corresponding to the label which indicates the family member.

In the other direction, if we look at the binary elimination rule, we see the need for two arguments to be sure of how to prove the implication for any family member of the binary *Or*.

$$orElim :: Or\ p\ q \to (p\ `Implies`\ r) \to (q\ `Implies`\ r) \to r$$

The generalisation unifies these two to one family of arguments. If we can prove R for each member of the family, we can be sure to prove R when we encounter some family member:

The constructors for proofs can be $\exists Intro :: RatSem \to Proof \to Proof$ and $\exists Elim :: (RatSem \to Prop) \to Proof \to Proof$. In this case we'd have i as the first argument of *ExistsProof* and a proof of $A\ (i)$ as its second argument.

Exercise 2.5. Sketch the *proofChecker* cases for universal quantification.

2.3.4 Typed quantification

So far, we have considered quantification always as over the full set of individuals, but it is often convenient to quantify over a subset with a certain property (like all even numbers, or all non-empty sets).

Even though it is not usually considered as strictly part of FOL, it does not fundamentally change its character if we extended it with several types (or sorts) of individuals (one speaks of "multi-sorted" FOL).

TODO: Fix formatting using $\forall \cdot . \cdot$.

In such a variant, the quantifiers look like $\forall(x : S) \circ P\ (x)$ and $\exists(x : S) \circ P\ (x)$.

Indeed, if a type (a set) S of terms can be described as those that satisfy the unary predicate T we can understand $\forall(x : T) \circ P\ (x)$ as a shorthand for $\forall x \circ T\ (x)\ `Implies`\ P\ (x)$. Likewise we can understand $\exists(x : T) \circ P\ (x)$ as a shorthand for $\exists x \circ T\ (x)\ `And`\ P\ (x)$.

As hinted at in the previous chapters, we find that writing types explicitly can greatly help understanding, and we won't refrain from writing down types in quantifiers in FOL formulas.

Exercise 2.6. Prove that the de Morgan dual of typed universal quantification is the typed existential quantification, using the above translation to untyped quantification.

2.3.5 Curry-Howard for quantification over individuals

We can try and draw parrellels with an hypothetical programming lanaguage corresponding to FOL.

In such a programming language, we expect to be able to encode proof rules as follows:

$$
\begin{aligned}
&\mathit{allIntro} :: ((a : \mathit{Individual}) \to P\ a) \to (\forall\ x.\ P\ x)\\
&\mathit{allElim} :: (\forall\ x.\ P\ x) \to ((a : \mathit{Individual}) \to P\ a)\\
&\mathit{existIntro} :: (a : \mathit{Individual}) \to P\ a \to \exists\ x.\ P\ x\\
&\exists\text{-}\mathit{Elim} :: ((a : \mathit{Individual}) \to P\ a\ `\mathit{Implies}`\ R) \to (\exists\ x.\ P\ x)\ `\mathit{Implies}`\ R
\end{aligned}
$$

(We must write the above using a *dependent* function type $(a : A) \to B$, see below.).

Taking the intuitionistic version of FOL (with the same treatment of negation as for propositional logic), we additionally expect to be able to represent proofs of quantifiers, directly. That is:

(t, b_t) is a program of type $\exists\ x.\ P\ (x)$ if b_t is has type $P\ (t)$.

f is a program of type $\forall\ x.\ P\ (x)$ if $f\ t$ is has type $P\ (t)$ for all t.

Unfortunately, in its 2010 standard, Haskell does not provide the equivalent of quantification over individuals. Therefore, one would have to use a different tool than Haskell as a proof assistant for (intuitionistic) FOL. the quantification that Haskell provides *forall a ∘ ...* is *over types* rather than individuals. [5] What we'd need is: 1. a type corresponding to universal quantification, the dependent function type $(a : A) \to B$, and 2. a type corresponding to $\exists\, x : A.\ P\ x$, the dependent pair $(x : A, P\ (x))$.

We can recommend the language Agda (which provides even more forms of quantification), however, in order to avoid a multiplicity of tools and potentially an excessive emphasis on proof formalism, we will refrain to formalise any proof as Agda programs in the Remainder. Rather, in the rest of the chapter, we will illustrate the logical principles seen so far by examples.

2.4 Examples

2.4.1 Proof by contradiction

Let us express and prove the irrationality of the square root of 2. We have two main concepts involved: the predicate "irrational" and the function "square root of". The square root function can be specified by the relation between two positive real numbers r and s as $r = \sqrt{s}$ iff $r\hat{}2 == s$. The formula "x is irrational" is just $\neg\ (R\ x)$ where R is the predicate "is rational".[6]

$$R\ x = \exists\, a : \mathbb{Z}.\ \exists\, b : \mathbb{N}_{>0}.\ b * x == a\ \&\ GCD\ (a, b) == 1$$

The pattern of proof by contradiction says that to prove a negation $\neg\ P$ is to assume P and derive something absurd. This pattern was formalised in Exercise 2.18 as *Not a = a 'Implies' False*. In turn, to obtain *False* we could prove simultaneously some Q and $\neg\ Q$, for example.

Let us take $P = R\ r$ and $Q = GCD\ (a, b) == 1$. Assuming P we immediately get Q so what we need is to prove $\neg\ Q$, that is $GCD\ (a, b) \neq 1$. We can use the equations $b * r == a$ and $r\hat{}2 == 2$. Squaring the first equation and using the second we get $b\hat{}2 * 2 == a\hat{}2$. Thus $a\hat{}2$ is even, which means that a is even, thus $a == 2 * c$ for some c. But then $b\hat{}2 * 2 == a\hat{}2 == 4 * c\hat{}2$ which means that $b\hat{}2 == 2 * c\hat{}2$. By the same reasoning again we have that also b is even. But then 2 is a factor of both a and b, which means that $GCD\ (a, b) \geqslant 2$, which in turn implies $\neg\ Q$.

To sum up: by assuming P we can prove both Q and $\neg\ Q$. Thus, by contradiction $\neg\ P$ must hold.

[5] There is ongoing progress in this direction, but we find the current state too clunky to be worthy of basing our development on it.

[6] In fact we additionally require the rational to be normalised (no common factor between the denominator and numerator) to simplify the proof.

2.4.2 Proof by cases

As another example, let's prove that there are two irrational numbers p and q such that $p\hat{}q$ is rational.

$$S = \exists\, p.\ \exists\, q.\ \neg\,(R\ p)\ \&\ \neg\,(R\ q)\ \&\ R\ (p\hat{}q)$$

We know from above that $r = \sqrt{2}$ is irrational, so as a first attempt we could set $p = q = r$. Then we have satisfied two of the three clauses ($\neg\ (R\ p)$ and $\neg\ (R\ q)$). What about the third clause: is $x = p\hat{}q == r\hat{}r$ rational? By the principle of the excluded middle (Section 2.2.6), we know that either $R\ x$ or $\neg\ (R\ x)$ must hold. Then, we apply $\vee$-elimination, and thus we have to deal with the two possible cases separately.

TODO: JP: how to parse this?

Case 1: $R\ x$ holds. Then we have a proof of S with $p = q = r = \sqrt{2}$.

Case 2: $\neg\ (R\ x)$ holds. Then we have another irrational number x to play with. Let's try $p = x$ and $q = r$. Then $p\hat{}q == x\hat{}r == (r\hat{}r)\hat{}r == r\hat{}(r * r) == r\hat{}2 == 2$ which is clearly rational. Thus, also in this case we have a proof of S, but now with $p = r\hat{}r$ and $q = r$.

To sum up: yes, there are irrational numbers such that their power is rational. We can prove the existence without knowing what numbers p and q actually are: this is because negation-elimination is a *non-constructive principle*. The best we could do in an intuitionistic logic, which is constructive, is to show that, if they were not to exist, then we come to a contradiction.

2.4.3 There is always another prime

As an example of combining quantification (forall, exists) and implication let us turn to one statement of the fact that there are infinitely many primes. If we assume that we have a unary predicate expressing that a number is prime and a binary (infix) predicate ordering the natural numbers we can define a formula *IP* for "Infinitely many Primes" as follows:

$$IP = \forall\, n.\ \mathit{Prime}\ n \Rightarrow \exists\, m.\ \mathit{Prime}\ m\ \&\ m > n$$

Combined with the fact that there is at least one prime (like 2) we can repeatedly refer to this statement to produce a never-ending stream of primes.

To prove this formula we are going to translate from logic to programs as described in Section 2.2.6. We can translate step by step, starting from the top level. The forall-quantifier translates to a (dependent) function type $(n : \mathit{Term}) \rightarrow$ and the implication to a normal function type $\mathit{Prime}\ n \rightarrow$. The exists-quantifier translates to a (dependent) pair type $((m : \mathit{Term}), ...)$ and finally the & translates into a pair type. Putting all this together we get a type signature for any *proof* of the theorem:

$$proof : (n : Term) \to Prime\ n \to ((m : Term), (Prime\ m, m > n))$$

This time the proof is going to be constructive: we have to find a concrete bigger prime, m. We can start filling in the definition of *proof* as a 2-argument function returning a triple. The key idea is to consider $1 + factorial\ n$ as a candidate new prime:

$$\begin{array}{l} proof\ n\ np = (m, (pm, gt)) \\ \quad \textbf{where}\ m' = 1 + factorial\ n \\ \qquad\qquad m = \text{\{- some non-trivial prime factor of } m' \text{ -\}} \\ \qquad\qquad pm = \text{\{- a proof that } m \text{ is prime -\}} \\ \qquad\qquad gt = \text{\{- a proof that } m > n \text{ -\}} \end{array}$$

The proof pm is the core of the theorem. First, we note that for any $2 \leqslant p \leqslant n$ we have

$$\begin{array}{ll} m'\%p & == \text{\{- Def. of } m' \text{ -\}} \\ (1 + n!)\%p & == \text{\{- modulo distributes over } + \text{ -\}} \\ (1\%p + (n!)\%p)\%p & == \text{\{- modulo comp.: } n! \text{ has } p \text{ as a factor -\}} \\ (1 \quad + 0)\%p & == \\ 1 & \end{array}$$

where $x\%y$ is the remainder after integer division of x by y. Thus m' is not divisible by any number from 2 to n. But is it a prime? Here we could, has before, use the law of excluded middle to progress. But we don't have to, because primality is a *decidable property*: we can write a terminating function which checks if m' is prime. We can then proceed by case analysis again: If m' is prime then $m = m'$ and the proof is done (because $1 + n! \geqslant 1 + n > n$). Otherwise, let m be a prime factor of m' (thus $m' = m * q$, $q > 1$). Then $1 == m'\%p == (m\%p) * (q\%p)$ which means that neither m nor q are divisible by p (otherwise the product would be zero). Thus they must both be $>n$, including m. QED.

The constructive character of this proofs means that it can be used to define a (mildly useful) function which takes any prime number to some larger prime number. We can compute a few example values:

$2 \mapsto 3$ (1+2!)
$3 \mapsto 7$ (1+3!)
$5 \mapsto 11$ (1+5! = 121 = 11*11)
$7 \mapsto 71$ ...

2.5 Basic concepts of calculus

Now we have built up quite a bit of machinery to express logic formulas and proofs. It is time time to apply it to some concepts in calculus. We start with the concept of "limit point" which is used in the formulation of different properties of limits of functions.

Limit point *Definition* (adapted from Rudin [1964], page 28): Let X be a subset of $\mathbb{R}$. A point $p \in \mathbb{R}$ is a limit point of X iff for every $\epsilon > 0$, there exists $q \in X$ such that $q \neq p$ and $|q - p| < \epsilon$.

To express "Let X be a subset of $\mathbb{R}$" we write $X : \mathcal{P}\ \mathbb{R}$. In general, the operator $\mathcal{P}$ takes a set (here $\mathbb{R}$) to the set of all its subsets.

$$\begin{aligned} &Limp : \mathbb{R} \to \mathcal{P}\ \mathbb{R} \to Prop \\ &Limp\ p\ X = \forall \epsilon > 0.\ \exists\, q \in X - \{p\}.\ |q - p| < \epsilon \end{aligned}$$

Notice that q depends on ϵ. Thus by introducing a function *getq* we can move the $\exists$ out.

$$\begin{aligned} &\textbf{type}\ Q = \mathbb{R}_{>0} \to (X - \{p\}) \\ &Limp\ p\ X = \exists getq : Q.\ \forall\, \epsilon > 0.\ |getq\ \epsilon - p| < \epsilon \end{aligned}$$

Next: introduce the "open ball" function B.

$$\begin{aligned} &B : \mathbb{R} \to \mathbb{R}_{>0} \to \mathcal{P}\ \mathbb{R} \\ &B\ c\ r = \{x \mid |x - c| < r\} \end{aligned}$$

$B\ c\ r$ is often called an "open ball" around c of radius r. On the real line this "open ball" is just an open interval, but with complex c or in more dimensions the term feels more natural. In every case $B\ c\ r$ is an open set of values (points) of distance less than r from c. The open balls around c are special cases of *neighbourhoods of c* which can have other shapes but must contain some open ball.

Using B we get

$$Limp\ p\ X = \exists getq : Q.\ \forall\, \epsilon > 0.\ getq\ \epsilon \in B\ p\ \epsilon$$

Example 1: Is $p = 1$ a limit point of $X = \{1\}$? No! $X - \{p\} = \{\}$ (there is no $q \neq p$ in X), thus there cannot exist a function *getq* because it would have to return elements in the empty set!

Example 2: Is $p = 1$ a limit point of the open interval $X = (0, 1)$? First note that $p \notin X$, but it is "very close" to X. A proof needs a function *getq* which from any ϵ computes a point $q = getq\ \epsilon$ which is in both X and $B\ 1\ \epsilon$. We need a point q which is in X and *closer* than ϵ from 1. We can try with $q = 1 - \epsilon\ /\ 2$ because $|1 - (1 - \epsilon\ /\ 2)| = |\epsilon\ /\ 2| = \epsilon\ /\ 2 < \epsilon$ which means $q \in B\ 1\ \epsilon$. We also see that $q \neq 1$ because $\epsilon > 0$. The only remaining thing to check is that $q \in X$. This is true for sufficiently small ϵ but the function *getq* must work for all positive reals. We can use any value in X (for example 17 / 38) for ϵ which are "too big" ($\epsilon \geqslant 2$). Thus our function can be

$$\begin{aligned} getq\ \epsilon \mid \epsilon < 2 \quad &= 1 - \epsilon\ /\ 2 \\ \mid otherwise &= 17\ /\ 38 \end{aligned}$$

A slight variation which is often useful would be to use *max* to define $getq\ \epsilon = max\ (17\ /\ 38, 1 - \epsilon\ /\ 2)$. Similarly, we can show that any internal point (like $1\ /\ 2$) is a limit point.

Example 3: limit of an infinite discrete set X

$$X = \{1\ /\ n \mid n \in \mathbb{N}_{>0}\}$$

Show that 0 is a limit point of X. Note (as above) that $0 \notin X$.

We want to prove $Limp\ 0\ X$ which is the same as $\exists getq : Q.\ \forall \epsilon > 0.\ getq\ \epsilon \in B\ 0\ \epsilon$. Thus, we need a function $getq$ which takes any $\epsilon > 0$ to an element of $X - \{0\} = X$ which is less than ϵ away from 0. Or, equivalently, we need a function $getn : \mathbb{R}_{>0} \to \mathbb{N}_{>0}$ such that $1\ /\ n < \epsilon$. Thus, we need to find an n such that $1\ /\ \epsilon < n$. If $1\ /\ \epsilon$ would be an integer we could use the next integer $(1 + 1\ /\ \epsilon)$, so the only step remaining is to round up:

$$getq\ \epsilon = 1\ /\ getn\ \epsilon$$
$$getn\ \epsilon = 1 + ceiling\ (1\ /\ \epsilon)$$

Exercise 2.7. prove that 0 is the *only* limit point of X.

Proposition: If X is finite, then it has no limit points.

$$\forall p \in \mathbb{R}.\ \neg\ (Limp\ p\ X)$$

This is a good exercise in quantifier negation!

$$\begin{array}{l}
\neg\ (Limp\ p\ X) = \{\text{- Def. of } Limp \text{ -}\} \\
\neg\ (\exists getq : Q.\ \forall \epsilon > 0.\ getq\ \epsilon \in B\ p\ \epsilon) \ = \{\text{- Negation of existential -}\} \\
\forall getq : Q.\ \neg\ (\forall \epsilon > 0.\ getq\ \epsilon \in B\ p\ \epsilon) \ = \{\text{- Negation of universal -}\} \\
\forall getq : Q.\ \exists \epsilon > 0.\ \neg\ (getq\ \epsilon \in B\ p\ \epsilon) = \{\text{- Simplification -}\} \\
\forall getq : Q.\ \exists \epsilon > 0.\ |getq\ \epsilon - p| \geqslant \epsilon
\end{array}$$

Thus, using the "functional interpretation" of this type we see that a proof needs a function *noLim*

$$noLim : (getq : Q) \to \mathbb{R}_{>0}$$

such that $\textbf{let}\ \epsilon = noLim\ getq\ \textbf{in}\ |getq\ \epsilon - p| \geqslant \epsilon$.

Note that *noLim* is a *higher-order* function: it takes a function *getq* as an argument. How can we analyse this function to find a suitable ϵ? The key here is that the range of *getq* is $X - \{p\}$ which is a finite set (not containing p). Thus we can enumerate all the possible results in a list $xs = [x_1, x_2, \ldots x_n]$, and measure their distances to p: $ds = map\ (\lambda x \to |x - p|)\ xs$. Now, if we let $\epsilon = minimum\ ds$ we can be certain that $|getq\ \epsilon - p| \geqslant \epsilon$ just as required (and $\epsilon \neq 0$ because $p \notin xs$).

TODO: JP: We explain this in Section 3.1

Exercise 2.8. If $Limp\ p\ X$ we now know that X is infinite.

Show how to construct an infinite sequence $a : \mathbb{N} \to \mathbb{R}$ of points in $X - \{p\}$ which gets arbitrarily close to p. Note that this construction can be seen as a proof of $Limp\ p\ X \Rightarrow Infinite\ X$.

2.5.1 The limit of a sequence

Now we can move from limit points to the more familiar limit of a sequence. At the core of this book is the ability to analyse definitions from mathematical texts, and here we will use the definition of the limit of a sequence of Adams and Essex [2010, page 498]:

> We say that sequence a_n converges to the limit L, and we write $\lim_{n\to\infty} a_n = L$, if for every positive real number ϵ there exists an integer N (which may depend on ϵ) such that if $n > N$, then $|a_n - L| < \epsilon$.

The first step is to type the variables introduced. A sequence a is a function from $\mathbb{N}$ to $\mathbb{R}$, thus $a : \mathbb{N} \to \mathbb{R}$ where a_n is special syntax for normal function application of a to $n : \mathbb{N}$. Then we have $L : \mathbb{R}$, $\epsilon : \mathbb{R}_{>0}$, and $N : \mathbb{N}$ (or $N : \mathbb{R}_{>0} \to \mathbb{N}$).

In the next step we analyse the new concept introduced: the syntactic form $\lim_{n\to\infty} a_n = L$ which we could express as an infix binary predicate *haslim* where a *haslim* L is well-typed if $a : \mathbb{N} \to \mathbb{R}$ and $L : \mathbb{R}$.

The third step is to formalise the definition using logic: we define *haslim* using a ternary helper predicate P:

$$
\begin{aligned}
&a \mathit{\ haslim\ } L = \forall\, \epsilon > 0.\ P\ a\ L\ \epsilon \quad \text{-- ``for every positive real number } \epsilon \ldots\text{''} \\
&P\ a\ L\ \epsilon = \exists N : \mathbb{N}.\ \forall\, n \geq N.\ |a_n - L| < \epsilon \\
&\qquad\quad = \exists N : \mathbb{N}.\ \forall\, n \geq N.\ a_n \in B\ L\ \epsilon \\
&\qquad\quad = \exists N : \mathbb{N}.\ I\ a\ N \subseteq B\ L\ \epsilon
\end{aligned}
$$

where we have introduced an "image function" for sequences "from N onward":

$$
\begin{aligned}
&I : (\mathbb{N} \to X) \to \mathbb{N} \to \mathcal{P}\ X \\
&I\ a\ N = \{a\ n \mid n \geq N\}
\end{aligned}
$$

The "forall-exists"-pattern is very common and it is often useful to transform such formulas into another form. In general $\forall\, x : X.\ \exists\, y : Y.\ Q\ x\ y$ is equivalent to $\exists\, gety : X \to Y.\ \forall\, x : X.\ Q\ x\ (gety\ x)$. In the new form we more clearly see the function *gety* which shows how the choice of y depends on x. For our case with *haslim* we can thus write

$$
a \mathit{\ haslim\ } L = \exists\, getN : \mathbb{R}_{>0} \to \mathbb{N}.\ \forall\, \epsilon > 0.\ I\ a\ (getN\ \epsilon) \subseteq B\ L\ \epsilon
$$

where we have made the function *getN* more visible. The core evidence of a *haslim* L is the existence of such a function (with suitable properties).

Exercise 2.9. Prove that the limit of a sequence is unique.

Exercise 2.10. prove that $(a_1 \; \mathit{haslim} \; L_1) \,\&\, (a_2 \; \mathit{haslim} \; L_2)$ implies $(a_1 + a_2) \; \mathit{haslim} \; (L_1 + L_2)$.

When we are not interested in the exact limit, just that it exists, we say that a sequence a is *convergent* when $\exists L.\ a \; \mathit{haslim} \; L$.

2.5.2 Case study: The limit of a function

As our next mathematical text book quote we take the definition of the limit of a function of type $\mathbb{R} \to \mathbb{R}$ from Adams and Essex [2010]:

> **A formal definition of limit**
>
> We say that $f(x)$ **approaches the limit** L as x **approaches** a, and we write
>
> $$\lim_{x \to a} f(x) = L,$$
>
> if the following condition is satisfied:
> for every number $\epsilon > 0$ there exists a number $\delta > 0$, possibly depending on ϵ, such that if $0 < |x - a| < \delta$, then x belongs to the domain of f and
>
> $$|f\ (x) - L| < \epsilon.$$

The *lim* notation has four components: a variable name x, a point a, an expression $f(x)$ and the limit L. The variable name and the expression can be combined into just the function f and this leaves us with three essential components: f, a, and L. Thus, *lim* can be seen as a ternary (3-argument) predicate which is satisfied if the limit of f exists at a and equals L. If we apply our logic toolbox we can define *lim* starting something like this:

TODO: JP: This is explained in Section 3.1

$$\mathit{lim}\ f\ a\ L = \forall\, \epsilon > 0.\ \exists\, \delta > 0.\ P\ \epsilon\ \delta$$

when P is a predicate yet to define. Indeed, it is often useful to introduce a local name (like P here) to help break the definition down into more manageable parts. If we now naively translate the last part we get this "definition" for P:

$$\textbf{where}\ P\ \epsilon\ \delta = (0 < |x - a| < \delta) \Rightarrow (x \in \mathit{Dom}\, f \wedge |f\ x - L| < \epsilon))$$

Note that there is a scoping problem: we have f, a, and L from the "call" to *lim* and we have ϵ and δ from the two quantifiers, but where did x come from? It turns out that the formulation "if ... then ..." hides a quantifier that binds x. Thus we get this definition:

$$lim\ a\ f\ L = \forall\, \epsilon > 0.\ \exists\, \delta > 0.\ \forall\, x.\ P\ \epsilon\ \delta\ x$$
$$\textbf{where}\ P\ \epsilon\ \delta\ x = (0 < |x - a| < \delta) \Rightarrow (x \in Dom\, f \wedge |f\ x - L| < \epsilon))$$

The predicate *lim* can be shown to be a partial function of two arguments, f and a. This means that each function f can have *at most* one limit L at a point a. (This is not evident from the definition and proving it is a good exercise.)

TODO: JP: Exercise: what does Adams means by "delta possibly depends on epsilon"? How did we express that in our formal defintion? Hint: how whould you express that delta cannot depend on epsilon?

2.6 Exercises

2.6.1 Representations of propositions

Exercise 2.11. Define a function for de Morgan dualisation.

Exercise 2.12. Define a function to rewrite propositions into conjunctive normal form.

Exercise 2.13. Define a function to rewrite propositions into disjunctive normal form.

Exercise 2.14. Propositions as polynomials. (This is a difficult exercise: it is a good idea to come back to it after Chapter 4 (where one learns about abstract structures) and after Chapter 5).

One way to connect logic to calculus is to view propositions as polynomials (in several variables). The key idea is to represent the truth values by zero (False) and one (True) and each named proposition P by a fresh variable p.

To represent logical operations one just has to check that the usual notion of expression evaluation gives the right answer for zero and one. (Can you express this as a homomorphism — seen in Chapter 4?)

The simplest operation to represent is *And* which becomes multiplication: P *And* Q translates to pq as can be easily checked. (Note that $p + q$ does not represent any proposition, because its value would be 2 for $p = q = 1$, but 2 does not represent any boolean.)

How should *Not*, *Or*, and *Implies* be represented?

2.6.2 Proofs

```
{-# LANGUAGE EmptyCase #-}
import PropositionalLogic
```

Short technical note For the exercises on the abstract representation of proofs for the propositional calculus using Haskell, (see Section 2.2.5), you might find it useful to take a look at typed holes, a feature which is enabled by default in GHC and available (the same way as the language extension `EmptyCase` above) from version 7.8.1 onwards: `https://wiki.haskell.org/GHC/Typed_holes`.

If you are familiar with Agda, these will be familiar to use. In summary, when trying to code up the definition of some expression (which you have already typed) you can get GHC's type checker to help you out a little in seeing how

far you might be from forming the expression you want. That is, how far you are from constructing something of the appropriate type.

Take *example0* below, and say you are writing:

$$example0\ e = andIntro\ (\_\ e)\ \_$$

When loading the module, GHC will tell you which types your holes (marked by "_") should have for the expression to be type correct.

On to the exercises.

Exercise 2.15. Prove these theorems (for arbitrary p, q and r):

$Impl\ (And\ p\ q)\ q$
$Or\ p\ q \to Or\ q\ p$
$(p \to q) \to (Not\ q \to Not\ p)$ -- call it *notMap*
$Or\ p\ (Not\ p)$ -- recall the law of excluded middle

For the hardest examples it can be good to use "theory exploration": try to combine the earlier theorems and rules to build up suitable term for which *notMap* or *notElim* could be used.

Exercise 2.16. Translate to Haskell and prove the De Morgan laws:

$$\neg\,(p \vee q) \leftrightarrow \neg\,p \wedge \neg q$$
$$\neg\,(p \wedge q) \leftrightarrow \neg\,p \vee \neg q$$

(translate equivalence to conjunction of two implications).

Exercise 2.17. So far, the implementation of the datatypes has played no role: we treated them as abstract. To make this clearer: define the types for connectives in *AbstractFol* in any way you wish, e.g.:

newtype $And\ p\ q = A\ p\ q$
newtype $Not\ p\ \ = B\ p$

etc. as long as you still export only the data types, and not the constructors. Convince yourself that the proofs given above still work and that the type checker can indeed be used as a poor man's proof checker.

Exercise 2.18. From now on you can assume the representation of proofs defined in .

1. Check your understanding by re-defining all the introduction and elimination rules as functions.
2. Compare proving the distributivity laws

$$(p \wedge q) \vee r \leftrightarrow (p \vee r) \wedge (q \vee r)$$
$$(p \vee q) \wedge r \leftrightarrow (p \wedge r) \vee (q \wedge r)$$

using only the introduction and elimination rules (no pairs, functions, etc.), with writing the corresponding functions with the given implementations of the datatypes. The first law, for example, requires a pair of functions:

$$(\mathit{Either}\ (p, q)\ r \rightarrow (\mathit{Either}\ p\ r, \mathit{Either}\ q\ r)$$
$$, (\mathit{Either}\ p\ r, \mathit{Either}\ q\ r) \rightarrow \mathit{Either}\ (p, q)\ r$$
$$)$$

Exercise 2.19. Assume

$$\textbf{type}\ \mathit{Not}\ p = p \rightarrow \mathit{False}$$

Implement *notIntro2* using the definition of *Not* above, i.e., find a function

$$\mathit{notIntro2} :: (p \rightarrow (q, q \rightarrow \mathit{False})) \rightarrow (p \rightarrow \mathit{False})$$

Using

$$\mathit{contraHey} :: \mathit{False} \rightarrow p$$
$$\mathit{contraHey}\ \mathit{evE} = \textbf{case}\ \mathit{evE}\ \textbf{of}\ \{\ \}$$

prove

$$(q \wedge \neg q) \rightarrow p$$

Can you prove $p \vee \neg p$?

Prove

$$\neg\, p \vee \neg q \rightarrow \neg(p \wedge q)$$

Can you prove the converse?

Exercise 2.20. Recall that every sentence provable in constructive logic is provable in classical logic. But the converse, as we have seen in the previous exercise, does not hold. On the other hand, there is no sentence in classical logic which would be contradicted in constructive logic. In particular, while we cannot prove $p \vee \neg p$, we *can* prove (constructively!) that there is no p for which $\neg(p \vee \neg p)$, i.e., that the sentence $\neg\, \neg\, (p \vee \neg p)$ is always true.

Show this by implementing the following function:

$$\mathit{noContra} :: (\mathit{Either}\ p\ (p \rightarrow \mathit{False}) \rightarrow \mathit{False}) \rightarrow \mathit{False}$$

Hint: The key is to use the function argument to *noContra* twice.

2.6.3 Continuity and limits

- when asked to "sketch an implementation" of a function, you must explain how the various results might be obtained from the arguments, in particular, why the evidence required as output may result from the evidence given as input. You may use all the facts you know (for instance, that addition is monotonic) without formalisation.
- to keep things short, let us abbreviate a significant chunk of the definition of *a haslim L* (see Section 2.5.1) by

$$P : \mathit{Seq}\ X \to X \to \mathbb{R}_{>0} \to \mathit{Prop}$$
$$P\ a\ L\ \epsilon = \exists\, N : \mathbb{N}.\ \forall\, n : \mathbb{N}.\ (n \geq N) \to (|a_n - L| < \epsilon)$$

Exercise 2.21. Consider the classical definition of continuity:

> *Definition:* Let $X \subseteq \mathbb{R}$, and $c \in X$. A function $f : X \to \mathbb{R}$ is *continuous at c* if for every $\epsilon > 0$, there exists $\delta > 0$ such that, for every x in the domain of f, if $|x - c| < \delta$, then $|fx - fc| < \epsilon$.

1. Write the definition formally, using logical connectives and quantifiers.
2. Introduce functions and types to simplify the definition.
3. Prove the following proposition: If `f` and `g` are continuous at `c`, `f + g` is continuous at `c`.

Exercise 2.22. Adequate notation for mathematical concepts and proofs.

A formal definition of "$f : X \to \mathbb{R}$ is continuous" and "f is continuous at c" can be written as follows (using the helper predicate Q):

$$\begin{aligned} C\ (f) &= \forall\, c : X.\ \mathit{Cat}\ (f, c) \\ \mathit{Cat}\ (f, c) &= \forall\, \epsilon > 0.\ \exists\, \delta > 0.\ Q\ (f, c, \epsilon, \delta) \\ Q\ (f, c, \epsilon, \delta) &= \forall\, x : X.\ |x - c| < \delta \Rightarrow |f\ x - f\ c| < \epsilon \end{aligned}$$

By moving the existential quantifier outwards we can introduce the function $get\delta$ which computes the required δ from c and ϵ:

$$C'\ (f) = \exists\, get\delta : X \to \mathbb{R}_{>0} \to \mathbb{R}_{>0}.\ \forall\, c : X.\ \forall\, \epsilon > 0.\ Q\ (f, c, \epsilon, get\delta\ c\ \epsilon)$$

Now, consider this definition of *uniform continuity*:

> **Definition:** Let $X \subseteq \mathbb{R}$. A function $f : X \to \mathbb{R}$ is *uniformly continuous* if for every $\epsilon > 0$, there exists $\delta > 0$ such that, for every x and y in the domain of f, if $|x - y| < \delta$, then $|f\ x - f\ y| < \epsilon$.

1. Write the definition of $UC\ (f)$ = "f is uniformly continuous" formally, using logical connectives and quantifiers. Try to use Q.
2. Transform $UC\ (f)$ into a new definition $UC'\ (f)$ by a transformation similar to the one from $C\ (f)$ to $C'\ (f)$. Explain the new function $new\delta$ introduced.
3. Prove that $\forall f : X \to \mathbb{R}.\ UC'\ (f) \Rightarrow C'\ (f)$. Explain your reasoning in terms of $get\delta$ and $new\delta$.

Exercise 2.23. Consider the statement:

The sequence $\{a_n\} = (0, 1, 0, 1, ...)$ does not converge.

1. Define the sequence $\{a_n\}$ as a function $a : \mathbb{N} \to \mathbb{R}$.
2. The statement "the sequence $\{a_n\}$ is convergent" is formalised as

   $$\exists\, L : \mathbb{R}.\ \forall\, \epsilon > 0.\ P\ a\ L\ \epsilon$$

   The formalisation of "the sequence $\{a_n\}$ is not convergent" is therefore

   $$\neg\, \exists\, L : \mathbb{R}.\ \forall\, \epsilon > 0.\ P\ a\ L\ \epsilon$$

   Simplify this expression using the rules

   $$\neg\, (\exists\, x.\ P\ x) \leftrightarrow (\forall\, x.\ \neg\, (P\ x))$$
   $$\neg\, (\forall\, x.\ P\ x) \leftrightarrow (\exists\, x.\ \neg\, (P\ x))$$
   $$\neg\, (P \to Q) \leftrightarrow P \wedge \neg Q$$

   The resulting formula should have no $\neg$ in it (that's possible because the negation of $<$ is $\geq$).
3. Give a functional interpretation of the resulting formula.
4. Sketch an implementation of the function, considering two cases: $L \neq 0$ and $L = 0$.

Exercise 2.24. Same as Exercise 2.23 but for $a = id$.

Exercise 2.25. Consider the statement:

The limit of a convergent sequence is unique.

1. There are many ways of formalising this in FOL. For example:

   $$\textbf{let}\ Q\ a\ L = \forall\, \epsilon > 0.\ P\ a\ L\ \epsilon$$
   $$\textbf{in}\ \forall\, L_1 : \mathbb{R}.\ \forall\, L_2 : \mathbb{R}.\ (Q\ a\ L_1 \wedge Q\ a\ L_2) \to L_1 = L_2$$

   i.e., if the sequence converges to two limits, then they must be equal, or

$$\forall\, L_1 : \mathbb{R}.\ \forall\, L_2 : \mathbb{R}.\ Q\ a\ L_1 \wedge L_1 \neq L_2 \rightarrow \neg\, Q\ a\ L_2$$

i.e., if a sequence converges to a limit, then it doesn't converge to anything that isn't the limit.

Simplify the latter alternative to eliminate the negation and give functional representations of both.

2. Choose one of the functions and sketch an implementation of it.

Chapter 3

Types in Mathematics

```
{-# LANGUAGE FlexibleInstances #-}
module DSLsofMath.W03 where
import Prelude hiding (Num (..), Fractional (..), Floating (..))
import DSLsofMath.Algebra (Algebraic (..), Transcendental (..))
type ℝ = Double
type ℝ = ℝ
type ℤ = Int
```

3.1 Types of functions, expressions and big operators

Examples of types in mathematics Simple types are sometimes mentioned explicitly in mathematical texts:

- $x \in \mathbb{R}$
- $\sqrt{\_} : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$
- $(\_)^2 : \mathbb{R} \to \mathbb{R}$ or, alternatively but *not* equivalently
- $(\_)^2 : \mathbb{R} \to \mathbb{R}_{\geq 0}$

However the types of big operators (sums, limits, integrals, etc.) are usually not given explicitly. In fact, it may not be clear at first sight that the summing operator ($\sum$) should be assigned a type at all! Yet this is exactly what we will set out to do, dealing with a dangerous pitfall of mathematical notation (Section 0.5.2). However, to be able to do so convincingly we shall clarify the relationship between functions and expressions first.

3.1.1 Expressions and functions of one variable

- $f(x) = x - 1$
- $g(x) = 2 * x^2 + 3$
- $h(y) = 2 * y^2 + 3$

As the reader may guess by now, we can assign to f, g, h the type $\mathbb{R} \to \mathbb{R}$. But other choices could work, such as $\mathbb{Z} \to \mathbb{Z}$, etc. For sure, they are functions. Additionally, the name of the variable appears to play no role in the meaning of the functions, and we can say, for example, $g = h$.

Consider now:

- $x - 1$
- $2 * x^2 + 3$
- $2 * y^2 + 3$

These are all expressions of one (free) variable. We could say that they type is $\mathbb{R}$ — but this is assuming that the free variable also has type $\mathbb{R}$. Furthermore, it is less clear now if $2 * x + 3 = 2 * y + 3$. In general one cannot simply change a variable name by another without making sure that 1. the renaming is applied everywhere uniformly and 2. the new variable name is not used for another purpose in the same scope (otherwise one informally says that there is a "clash").

To clarify this situation, we will now formalise expressions of one variables as a DSL. For simplicity we will focus on arithmetic expressions only. Therefore we have constructors for addition, multiplication and constants, as in Section 1.3.1. Additionally, we have the all-important constructor for variables, which we will call X here. We can implement all this in a datatype as follows:

TODO: JP: Rename FunExp -> Exp1V or similar?

Deep embedding

$$
\begin{aligned}
\textbf{data}\ \mathit{FunExp} = {} & \mathit{Const}\ \mathbb{R} \\
\mid {} & X \\
\mid {} & \mathit{FunExp}\ \mathbin{:+:}\ \mathit{FunExp} \\
\mid {} & \mathit{FunExp}\ \mathbin{:*:}\ \mathit{FunExp} \\
\mid {} & \mathit{Exp}\ \mathit{FunExp}
\end{aligned}
$$

We could encode our examples as follows:

- $X \mathbin{:+:} \mathit{Const}\ (-1)$

- $Const\ 2 :*: (X :*: X) :+: Const\ 3$

We no longer have a third example: we can only ever represent one variable, as X, and thus we skip the last example, equal to the second.

We can now evaluate the value of these expressions. The meaning of operators and constants is as in Section 1.3.1. But, to be able to evaluate X, the variable, we need its value — and we simply take it as a parameter.

$$\begin{array}{lll} eval :: FunExp \to \mathbb{R} \to \mathbb{R} & & \\ eval & (Const\ \alpha) & x = \alpha \\ eval & X & x = x \\ eval & (e_1 :+: e_2) & x = eval\ e_1\ x + eval\ e_2\ x \\ eval & (e_1 :*: e_2) & x = eval\ e_1\ x * eval\ e_2\ x \end{array}$$

However, we can make an equivalent interpretation of the above type as

$$eval :: FunExp \to (\mathbb{R} \to \mathbb{R})$$

That is, *FunExp* can be interpreted as a function! This is perhaps surprising, but the reason is that we used a fixed Haskell symbol (constructor) for the variable. There is ever a single variable in *FunExp*, and thus they are really equivalent to functions of a single variable.

Shallow embedding

Thus the above was a deep embedding for functions of a single variable. A shallow embedding would be a using functions as the representation, say:

$$\textbf{type}\ FunExpS = \mathbb{R} \to \mathbb{R}$$

Then we can define the operators directly on functions, as follows:

$$\begin{array}{l} funConst\ \alpha = \lambda x \to \alpha \\ funX = \lambda x \to x \\ funPlus\ f\ g = \lambda x \to f\ x + g\ x \\ funTimes\ f\ g = \lambda x \to f\ x * g\ x \end{array}$$

Again, we have two possible intuitive readings of the above equations. The first reading, as expressions of a single variable, is that the variable (x) is interpreted as the identity function; a constant α is interpreted as a constant function; the sum of two expressions is interpreted as the sum of the evaluation of the operands, etc.

The second reading is that we can define an arithmetic structure ($*$,$+$, etc.) on functions, by lifting the operators to work pointwise.

TODO: JP: This vocabulary is probably unknown at this point.

To wrap it up, if we're so inclined, we can re-define the evaluator of the deep-embedding using the operators of the shallow embedding:

$$
\begin{array}{lll}
\mathit{eval} :: \mathit{FunExp} & \to \mathbb{R} \to \mathbb{R} \\
\mathit{eval} \quad (\mathit{Const}\ \alpha) & = \mathit{funConst}\ \alpha \\
\mathit{eval} \quad X & = \mathit{funX} \\
\mathit{eval} \quad (e_1 :+: e_2) & = \mathit{funPlus} \quad (\mathit{eval}\ e_1)\ (\mathit{eval}\ e_2) \\
\mathit{eval} \quad (e_1 :*: e_2) & = \mathit{funTimes}\ (\mathit{eval}\ e_1)\ (\mathit{eval}\ e_2)
\end{array}
$$

Representing expressions of one variable as functions (of one argument) is a recurring technique in this book (). To start, we can use it to assign types to big operators.

TODO: JP: Link back references to FunExp and function instances.

3.1.2 Scoping and Typing big operators

Consider the mathematical expression

$$\sum_{i=1}^{n} i^2$$

To be able to see which type is appropriate for $\sum$, we have to consider the type of the summand (i^2 in the example) first. As you may have guessed, it is an expression of one variable (i). You may ask: but surely the body of the summation operator can use other variables? You'd be entirely correct. However, *from the point of view of the summation*, it is as if such other variables were constant. Accepting this assertion as a fact until we can show a more complicated example, we can now assign a type to the summation operator. For simplicity, we will be using the shallow embedding; thus the operand can be typed as, say, $\mathbb{Z} \to \mathbb{R}$. The other arguments will be the limit points (1 and n in our example). The variable, i shall not be represented as an argument: indeed, the variable name is *fixed by the representation of functions*. There is no choice to make at the point of summation. Thus, we write:

$$\mathit{summation} :: \mathbb{Z} \to \mathbb{Z} \to (\mathbb{Z} \to \mathbb{R}) \to \mathbb{R}$$

Conveniently, we can even provide a simple implementation:

$$\mathit{summation}\ \mathit{low}\ \mathit{high}\ f = \mathit{sum}\ [f\ i \mid i \leftarrow [\mathit{low}\,..\,\mathit{high}]]$$

As another example, let us represent the following nested sum

$$\sum_{i=1}^{n} \sum_{j=1}^{m} i^2 + j^2$$

using the shallow embedding of summation. This representation can be written simply as follows:

$$exampleSum\ m\ n = summation\ 1\ m\ (\lambda i \to summation\ 1\ n\ (\lambda j \to i\hat{}2 + j\hat{}2))$$

Aren't we cheating though? Surely we said that only one variable could occur in the summand, but we see both i and j? Well, we are not cheating as long as we use the *shallow embedding* for functions of one variables. Doing so allows us to 1. use lambda notation to bind (and name) the variable name of the summation however we wish (in this case i and j) and 2. we can freely use any haskell function of type $\mathbb{Z} \to \mathbb{R}$ as the summand. In particular, the this function can be any lambda-expression returing $\mathbb{R}$, and this expression can include summation itself. This freedom is an advantage of shallow embeddings: if we were to use the deep embedding, then we'd need a whole lot more work to ensure that we can represent summation within the deep embedding. In particular we need a way to embed variable binding itself. And we shall not be opening this can of worms just yet, even though we take a glimpse in Section 3.2.

Sticking conveniently to the shallow embedding, we can apply the same kind of reasoning to other big operators, and obtain the following typings:

- $lim : (\mathbb{N} \to \mathbb{R}) \to \mathbb{R}$ for $lim_{n\to\infty}\{a_n\}$
- $d/dt : (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$
  - sometimes, instead of df/dt one sees f' or $\dot{f}$ or Df

In sum, the chief difficulty to overcome when assigning types for mathematical operators is that they often introduce (bind) variable names. To take another example from the above, $\lim_{n\to\infty}$ binds n in a_n. In this book our stance is to make this binding obvious by letting the body of the limit (a_n in the example) be a function. Thus we assign it the type $\mathbb{N} \to \mathbb{R}$. Therefore the limit operator has a higher order type. A similar line of reasoning justifies the types of derivatives. We study in detail how these play out first.

3.2 Detour: expressions of several variables

In first reading this section can be skipped, however it is natural to to extend the study of expressions from single variables to multiple variables.

3.2.1 Partial functions

As an warmup, and for reasons which will become obvious soon (in Section 3.2.2), we begin by presenting a DSL for partial functions with a finite domain. The type *Env v s* will be the *syntax* for the type of partial functions from v to s, and defined as follows:

$$\textbf{type}\ Env\ v\ s = [(v, s)]$$

As an example value of this type we can take:

$$env1 :: Env\ String\ Int$$
$$env1 = [(\texttt{"hey"}, 17), (\texttt{"you"}, 38)]$$

The intended meaning is that `"hey"` is mapped to 17, etc. The semantic domain is the set of partial functions, and, as discussed above, we represent those as the Haskell type $v \to Maybe\ s$.

Our evaluation function, *evalEnv*, maps the syntax to the semantics, and as such has the following type:

$$evalEnv :: Eq\ v \Rightarrow Env\ v\ s \to (v \to Maybe\ s)$$

This type signature deserves some more explanation. The first part ($Eq\ v \Rightarrow$) is a constraint which says that the function works, not for *all* types v, but only for those who support a boolean equality check ($(==) :: v \to v \to Bool$). The rest of the type signature ($Env\ v\ s \to (v \to Maybe\ s)$) can be interpreted in two ways: either as the type of a one-argument function taking an $Env\ v\ s$ and returning a function, or as the type of a two-argument function taking an $Env\ v\ s$ and a v and maybe returning an s.

The implementation proceeds by searching for the first occurence of x In the list of pairs (v, s) such that $x == v$, and return *Just s* if one is found, and *Nothing* otherwise.

$$\begin{aligned} &evalEnv\ vss\ x = findFst\ vss \\ &\quad \textbf{where}\ findFst\ ((v, s) : vss) \\ &\qquad\qquad \mid x == v \quad\ \ = Just\ s \\ &\qquad\qquad \mid otherwise = findFst\ vss \\ &\qquad\ \ findFst\ [\,] \quad\ \ = Nothing \end{aligned}$$

Another equivalent definition is $evalEnv = flip\ lookup$, where *lookup* is defined in the Haskell Prelude:

$$lookup :: Eq\ a \Rightarrow a \to [(a, b)] \to Maybe\ b$$

3.2.2 The data type of multiple variables expressions

Let us define the following type, describing a deep embedding for simple arithmetic expressions. Compared to single variable expressions, we add one argument for variables, giving the *name* of the variable. Here we use a string, so we have an infinite supply of variables.

TODO: JP: rename type/constructors to match single variable expressions

$$\textbf{data}\ AE = V\ String \mid P\ AE\ AE \mid T\ AE\ AE$$

TODO: JP: There does not seem to be sense or rhyme in the name of data types and constructors.

The above declaration introduces:

TODO: JP: move this kind of consideration much earlier. Haskell primer?

- a new type AE for simple arithmetic expressions,
- a constructor $V :: String \to AE$ to represent variables,
- a constructor $P :: AE \to AE \to AE$ to represent plus, and
- a constructor $T :: AE \to AE \to AE$ to represent times.

Example values include $x = V$ `"x"`, $e_1 = P\ x\ x$, and $e_2 = T\ e_1\ e_1$.

If you want a constructor to be used as an infix operator you need to use symbol characters and start with a colon:

$$\textbf{data}\ AE' = V'\ String \mid AE' :\dot{+} AE' \mid AE' :* AE'$$

Example values are then $y = V'$ `"y"`, $e_1 = y :\dot{+} y$ and $e_2 = x :* e_1$.

Finally, you can add one or more type parameters to make a whole family of datatypes in one go:

TODO: JP: move this kind of consideration much earlier. Haskell primer?

$$\textbf{data}\ AE'\ v = V'\ v \mid AE'\ v :\dot{+} AE'\ v \mid AE'\ v :* AE'\ v$$

The purpose of the parameter v here is to enable a free choice of type for the variables (be it *String* or *Int* or something else).

The careful reader will note that the same Haskell module cannot contain both these definitions of AE'. This is because the name of the type and the names of the constructors are clashing. The typical ways around this are either to define the types in different modules, or rename one of them (often by adding primes as in AE'). In this book we often take the liberty of presenting more than one version of a datatype without changing the names, to avoid multiple modules or too many primed names.

TODO: JP: move this kind of consideration much earlier. Haskell primer?

Together with a datatype for the syntax of arithmetic expressions we want to define an evaluator of the expressions.

In the evaluator for AE we take this idea one step further: given an environment *env* and the syntax of an arithmetic expression e we compute the value of that expression. Hence, the semantics of AE is a function of type $Env\ String\ Integer \to Maybe\ Integer$.

$$\begin{aligned}
&evalAE :: AE \to (Env\ String\ Integer \to Maybe\ Integer)\\
&evalAE\ (V\ x)\ env\ \ \ \ \ \ = evalEnv\ env\ x\\
&evalAE\ (P\ e_1\ e_2)\ env = mayP\ (evalAE\ e_1\ env)\ (evalAE\ e_2\ env)\\
&evalAE\ (T\ e_1\ e_2)\ env = mayT\ (evalAE\ e_1\ env)\ (evalAE\ e_2\ env)\\
&mayP :: Maybe\ Integer \to Maybe\ Integer \to Maybe\ Integer\\
&mayP\ (Just\ a)\ (Just\ b) = Just\ (a + b)\\
&mayP\ _\ _\ \ \ \ \ \ \ \ \ \ \ \ \ = Nothing\\
&mayT :: Maybe\ Integer \to Maybe\ Integer \to Maybe\ Integer
\end{aligned}$$

$$\begin{aligned} &mayT\ (Just\ a)\ (Just\ b) = Just\ (a * b)\\ &mayT\ \_\ \_ \qquad\qquad\quad = Nothing \end{aligned}$$

The corresponding code for AE' is more general and you don't need to understand it at this stage, but it is left here as an example for those with a stronger Haskell background.

TODO: JP: Actually the AE/AE′ generalisation has nothing to do with the change in code.

$$\begin{aligned} &evalAE' :: (Eq\ v, \_) \Rightarrow (Env\ v\ sem) \to (AE'\ v \to Maybe\ sem)\\ &evalAE'\ env\ (V'\ x) \quad = evalEnv\ env\ x\\ &evalAE'\ env\ (e_1 :+ e_2) = liftM\ (+)\ (evalAE'\ env\ e_1)\ (evalAE'\ env\ e_2)\\ &evalAE'\ env\ (e_1 :* e_2) = liftM\ (*)\ \ (evalAE'\ env\ e_1)\ (evalAE'\ env\ e_2)\\ &liftM :: (a \to b \to c) \to (Maybe\ a \to Maybe\ b \to Maybe\ c)\\ &liftM\ op\ \ (Just\ a)\ (Just\ b) = Just\ (op\ a\ b)\\ &liftM\ \_op\ \_ \qquad \_ \qquad = Nothing \end{aligned}$$

The approach taken above is to use a *String* to name each variable: indeed, $Env\ String\ \mathbb{R}$ is like a tuple of several variables values. However, other situations, it is better to refer to variables by position.

For example, we can pick out any variable and make it a function of said variable like so:

$$\begin{aligned} &fun1 :: (Env\ String\ \mathbb{R} \to \mathbb{R}) \to Env\ String\ \mathbb{R} \to String \to (\mathbb{R} \to \mathbb{R})\\ &fun1\ funMultiple\ env\ variable\ value = funMultiple\ ((variable, value) : env) \end{aligned}$$

Exercise 3.1. Assume a function f of 3 variables, named `"x"`,`"y"` and `"y"`, and given the type $Env\ String\ \mathbb{R} \to \mathbb{R}$. Turn it into a function g of type $\mathbb{R} \to \mathbb{R} \to \mathbb{R} \to \mathbb{R}$ with the same intended meaning.

TODO: JP: Talk some about variable capture?

3.3 Typing Mathematics: derivative of a function

Let's start with the classical definition of the derivative of Adams and Essex [2010]:

> The **derivative** of a function f is another function f' defined by
>
> $$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$
>
> at all points x for which the limit exists (i.e., is a finite real number).
> If $f'(x)$ exists, we say that f is **differentiable** at x.

We can start by assigning types to the expressions in the definition. Let's write X for the domain of f so that we have $f : X \to \mathbb{R}$ and $X \subseteq \mathbb{R}$ (or, equivalently, $X : \mathcal{P}\ \mathbb{R}$). If we denote with Y the subset of X for which f is differentiable we get $f' : Y \to \mathbb{R}$. Thus, the operation which maps f to f' has type $(X \to \mathbb{R}) \to (Y \to \mathbb{R})$. Unfortunately, the only notation for this operation given (implicitly) in the definition is a postfix prime. To make it easier to see we use a prefix D instead and we can thus write $D : (X \to \mathbb{R}) \to (Y \to \mathbb{R})$. We will often assume that $X = Y$ (f is differentiable everywhere) so that we can can see D as preserving the type of its argument.

Now, with the type of D sorted out, we can turn to the actual definition of the function $D\ f$. The definition is given for a fixed (but arbitrary) x. (At this point the reader may want to check the definition of "limit of a function" in Section 2.5.2.) The *lim* expression is using the (anonymous) function $g\ h = \frac{f\ (x+h) - f\ x}{h}$ and that the limit of g is taken at 0. Note that g is defined in the scope of x and that its definition uses x so it can be seen as having x as an implicit, first argument. To be more explicit we write $\varphi\ x\ h = \frac{f\ (x+h) - f\ x}{h}$ and take the limit of $\varphi\ x$ at 0. So, to sum up, $D\ f\ x = lim\ (\varphi\ x)\ 0$. [1] The key here is that we name, type, and specify the operation of computing the derivative (of a one-argument function). We will use this operation quite a bit in the rest of the book, but here are just a few examples to get used to the notation. With the following definitions:

TODO: JP: But in chapter 2 lim was a predicate? Check.

$$\begin{aligned} sq\ x &= x\hat{}2 \\ double\ x &= 2 * x \\ c_2\ x &= 2 \end{aligned}$$

we have the following equalities:

$$\begin{aligned} sq' &== D\ sq == D\ (\lambda x \to x\hat{}2) == D\ (\hat{}2) == (2*) == double \\ sq'' &== D\ sq' == D\ double == c_2 == const\ 2 \end{aligned}$$

What we cannot do at this stage is to actually *implement* D in Haskell. If we only have a function $f : \mathbb{R} \to \mathbb{R}$ as a "black box" we cannot really compute the actual derivative $f' : \mathbb{R} \to \mathbb{R}$, but only numerical approximations. However if we also have access to the "source code" of f, then we can apply the usual rules we have learnt in calculus. We will get get back to this question in Section 3.8.

3.4 Typing Mathematics: partial derivative

Armed with our knowledge of functions of more than one variable, we can continue on our quest to type the elements of mathematical textbook definitions.

[1] We could go one step further by noting that f is in the scope of φ and used in its definition. Thus the function $\psi\ f\ x\ h = \varphi\ x\ h$, or $\psi\ f = \varphi$, is used. With this notation, and $limAt\ x\ f = lim\ f\ x$, we obtain a point-free definition that can come in handy: $D\ f = limAt\ 0 \circ \psi\ f$.

Our example here is by Mac Lane [1986, page 169], where we read

> [...] a function $z = f\ (x, y)$ for all points (x, y) in some open set U of the cartesian (x, y)-plane. [...] If one holds y fixed, the quantity z remains just a function of x; its derivative, when it exists, is called the *partial derivative* with respect to x. Thus at a point (x, y) in U this derivative for $h \neq 0$ is
>
> $$\partial z/\partial x = f'_x(x, y) = \lim_{h\to 0}(f(x + h, y) - f(x, y))/h$$

What are the types of the elements involved? We have

$$\begin{array}{lll} U \subseteq \mathbb{R} \times \mathbb{R} & & \text{-- cartesian plane} \\ f & : U \to \mathbb{R} & \\ z & : U \to \mathbb{R} & \text{-- but see below} \\ f'_x & : U \to \mathbb{R} & \end{array}$$

The x in the subscript of f' is *not* a real number, but a symbol (we used *String* for similar purposes in Section 3.2).

The expression (x, y) has six occurrences. The first two (on line 1) denote variables of type U, the third (on line 2) is just a name ((x, y)-plane). The fourth (at line 4) denotes a variable of type U bound by a universal quantifier: "a point (x, y) in U" as text which would translate to $\forall (x, y) \in U$ as a formula fragment.

The variable h is a non-zero real number. The use of the word "for" might lead one to believe that it is bound by a universal quantifier ("for $h \neq 0$" on line 4), but that is incorrect. In fact, h is used as a local variable introduced in the subscript of lim. This variable h is a parameter of an anonymous function, whose limit is then taken at 0.

That function, which we can name φ, has the type $\varphi : U \to (\mathbb{R} - \{0\}) \to \mathbb{R}$ and is defined by

$$\varphi\ (x, y)\ h = (f\ (x + h, y) - f\ (x, y))\ /\ h$$

The limit is then written $lim\ (\varphi\ (x, y))\ 0$. Note that 0 is a limit point of $\mathbb{R} - \{0\}$, so the type of *lim* is the one we have discussed:

$$lim : (X \to \mathbb{R}) \to \{p \mid p \in \mathbb{R}, Limp\ p\ X\} \to \mathbb{R}$$

On line 1, $z = f\ (x, y)$ probably does not mean that we let z be a fixed value in $\mathbb{R}$, although the phrase "the quantity z" (on line 2) suggests this. Rather, a possible interpretation is that z is used to abbreviate the expression $f\ (x, y)$. That is, z stands for an expression which depends on x and y; thus, it can be enlightening to replace z with $f\ (x, y)$ everywhere. In particular, $\partial z\ /\ \partial x$ becomes $\partial f\ (x, y)\ /$

∂x, which we can interpret as the operator ∂ / ∂x applied to $f\ (x, y)$ (remember that (x, y) is bound in the context by a universal quantifier on line 4). There is the added difficulty that, just like the subscript in f'_x, the x in ∂x is not the x bound by the universal quantifier, but just a symbol.

To sum up, partial derivative operators which mention symbols (such as ∂ / ∂x or "prime subscript x") do act on an representation of functions which uses symbols for the variables (not positions), such as presented in Section 3.2.2. This is why we mostly see $\partial f/\partial x$, $\partial f/\partial y$, $\partial f/\partial z$ etc. when, in the context, the function f has been given a definition of the form $f(x, y, z) = \ldots$. This kind of approach presents several difficulties:

1. it makes it hard to rename variables (which can be a problem if one is renaming variables, for example for the purpose of integration)
2. Further confusion can be created when a variable (such as z above) depends on other variables. Tracing dependencies can become daunting and it is easy to make errors of name when doing calculations.
3. it makes it difficult to assign a higher order type to the partial derivatives. Indeed, as we have seen in Section 3.1.2, doing this means that the operator binds the name of the variable. But it is often awkward to make partial differentiation bind a variable.

One possibility would be to use the following type: $\partial/\partial x_i : (\mathbb{R}^n \to \mathbb{R}) \to (\mathbb{R}^n \to \mathbb{R})$ But it still assume as input a vector of variables x. Hence we prefer a notation which doesn't rely on the names given to the arguments whatsoever. It was popularised by Landau [1934] (English edition Landau [2001]): D_1 for the partial derivative with respect to the the first argument, D_2 for the partial derivative with respect to the the second argument, etc.

Exercise 3.6: for $f : \mathbb{R}^2 \to \mathbb{R}$ define D_1 and D_2 using only D.

3.5 Type inference and understanding: Lagrangian case study

From (Sussman and Wisdom 2013): TODO: JP: fix citation

> A mechanical system is described by a Lagrangian function of the system state (time, coordinates, and velocities). A motion of the system is described by a path that gives the coordinates for each moment of time. A path is allowed if and only if it satisfies the Lagrange equations. Traditionally, the Lagrange equations are written

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{q}} - \frac{\partial L}{\partial q} = 0$$

What could this expression possibly mean?

To start answering the question of Sussman and Wisdom, we start typing the elements involved:

1. The use of notation for "partial derivative", $\partial L/\partial q$, suggests that L is a function of at least a pair of arguments:

   $$L : \mathbb{R}^i \to \mathbb{R}, i \geq 2$$

   This is consistent with the description: "Lagrangian function of the system state (time, coordinates, and velocities)". So, if we let "coordinates" be just one coordinate, then there is also a single velocity[2] and so we can take $i = 3$:

   $$L : \mathbb{R}^3 \to \mathbb{R}$$

   The "system state" here is a triple (of type $S = (T, Q, V) = \mathbb{R}^3$) and we can call the three components $t : T$ for time, $q : Q$ for coordinate, and $v : V$ for velocity. (We use $T = Q = V = \mathbb{R}$ in this example but it can help the reading to remember the different uses of $\mathbb{R}$ — this would help for example to generalise to more than one coordinate.)

2. Looking again at the same derivative, $\partial L/\partial q$ suggests that q is the name of a real variable, one of the three arguments to L. In the context, which we do not have, we would expect to find somewhere the definition of the Lagrangian as

   $$\begin{aligned} L &: (T, Q, V) \to \mathbb{R} \\ L &\ (t, q, v) \ = \ ... \end{aligned}$$

3. therefore, $\partial L/\partial q$ should also be a function of the same triple of arguments:

   $$(\partial L \ / \ \partial q) : (T, Q, V) \to \mathbb{R}$$

   It follows that the equation expresses a relation between *functions*, therefore the 0 on the right-hand side of the Lagrange equation(s) is *not* the real number 0, but rather the constant function *const* 0:

   $$\begin{aligned} &const\ 0 : (T, Q, V) \to \mathbb{R} \\ &const\ 0 \ \ (t, q, v) \ = 0 \end{aligned}$$

[2] A bit of domain knowledge is necessary here

4. We now have a problem: $d\ /\ dt$ can only be applied to functions of *one* real argument t, and the result is a function of one real argument:

   $$(d\ /\ dt)\ (\partial L\ /\ \partial \dot{q}) : T \to \mathbb{R}$$

   Since we subtract from this the function $\partial L/\partial q$, it follows that this, too, must be of type $T \to \mathbb{R}$. But we already typed it as $(T, Q, V) \to \mathbb{R}$, contradiction!

5. The expression $\partial L/\partial \dot{q}$ appears to also be malformed. We would expect a variable name where we find $\dot{q}$, but $\dot{q}$ is the same as dq/dt, a function.

6. Looking back at the description above, we see that the only immediate candidate for an application of d/dt is "a path that gives the coordinates for each moment of time". Thus, the path is a function of time, let us say

   $$w : T \to Q \quad \text{-- with } T = \mathbb{R} \text{ for time and } Q = \mathbb{R} \text{ for coordinates } (q : Q)$$

   We can now guess that the use of the plural form "equations" might have something to do with the use of "coordinates" in the plural. In an n-dimensional space, a position is given by n coordinates. A path would then be a function

   $$w : T \to Q \quad \text{-- with } Q = \mathbb{R}^n$$

   which is equivalent to n functions of type $T \to \mathbb{R}$, each computing one coordinate as a function of time. We would then have an equation for each of them. But we will come back to use $n = 1$ for the rest of this example.

7. Now that we have a path, the coordinates at any time are given by the path. And because the time derivative of a coordinate is a velocity, we can actually compute the trajectory of the full system state (T, Q, V) starting from just the path.

   $$\begin{aligned} &q : T \to Q \\ &q\ t = w\ t \qquad \text{-- or, equivalently, } q = w \\ &\dot{q} : T \to V \\ &\dot{q}\ t = dw\ /\ dt \quad \text{-- or, equivalently, } \dot{q} = D\ w \end{aligned}$$

   We combine these in the "combinator" *expand*, given by

   $$\begin{aligned} &expand : (T \to Q) \to (T \to (T, Q, V)) \\ &expand\ w\ t = (t, w\ t, D\ w\ t) \end{aligned}$$

8. With *expand* in our toolbox we can fix the typing problem in item 4 above. The Lagrangian is a "function of the system state (time, coordinates, and velocities)" and the "expanded path" (*expand* w) computes the state from just the time. By composing them we get a function

$$L \circ (\mathit{expand}\ w) : T \to \mathbb{R}$$

which describes how the Lagrangian would vary over time if the system would evolve according to the path w.

This particular composition is not used in the equation, but we do have

$$(\partial L\ /\ \partial q) \circ (\mathit{expand}\ w) : T \to \mathbb{R}$$

which is used inside $d\ /\ dt$.

9. We now move to using D for $d\ /\ dt$, D_2 for $\partial\ /\ \partial q$, and D_3 for $\partial\ /\ \partial \dot{q}$. In combination with *expand* w we find these type correct combinations for the two terms in the equation:

$$\begin{aligned} D\ ((D_2\ L) \circ (\mathit{expand}\ w)) &: T \to \mathbb{R} \\ (D_3\ L)\ \circ (\mathit{expand}\ w)\ &: T \to \mathbb{R} \end{aligned}$$

The equation becomes

$$D\ ((D_3\ L) \circ (\mathit{expand}\ w)) - (D_2\ L) \circ (\mathit{expand}\ w) = \mathit{const}\ 0$$

or, after simplification:

$$D\ (D_3\ L \circ \mathit{expand}\ w) = D_2\ L \circ \mathit{expand}\ w$$

where both sides are functions of type $T \to \mathbb{R}$.

10. "A path is allowed if and only if it satisfies the Lagrange equations" means that this equation is a predicate on paths (for a particular L):

$$\mathit{Lagrange}\ (L, w) = D\ (D_3\ L \circ \mathit{expand}\ w) \stackrel{==}{} D_2\ L \circ \mathit{expand}\ w$$

where we use $(\stackrel{==}{})$ to avoid confusion with the equality sign $(=)$ used for the definition of the predicate.

So, we have figured out what the equation "means", in terms of operators that we recognise. If we zoom out slightly we see that the quoted text means something like: If we can describe the mechanical system in terms of "a Lagrangian" $(L : S \to \mathbb{R})$, then we can use the equation to check if a particular candidate path $w : T \to \mathbb{R}$ qualifies as a possible "motion of the system" or not. The unknown of the equation is the path w, and as the equation involves partial derivatives it is an example of a partial differential equation (a PDE). We will not dig into how to solve such PDEs, but they are widely used in physics.

TODO: JP: what is S?

3.6 Incremental analysis with types

So far we have worked on typing mathematics "by hand", but we can actually get the Haskell interpreter to help a bit even when we are still at the specification stage. It is often useful to collect the known (or assumed) facts about types in a Haskell file and regularly check if the type checker agrees. Consider the following text from Mac Lane's *Mathematics Form and Function* (page 182):

> In these cases one tries to find not the values of x which make a given function $y = f\ (x)$ a minimum, but the values of a given function $f\ (x)$ which make a given quantity a minimum. Typically, that quantity is usually measured by an integral whose integrand is some expression F involving both x, values of the function $y = f\ (x)$ at interest and the values of its derivatives — say an integral
>
> $$\int_a^b F(y, y', x)dx, \quad y = f(x).$$

We will use the above example as an example of getting feedback from a type checker. We start by declaring two types, X and Y, and a function f between them:

```
data X   -- X must include the interval [a, b] of the reals
data Y   -- another subset of the reals
f :: X → Y
f = undefined
```

To the Haskell interpreter, such empty **data**-declarations mean that there is no way to construct any element for them, as we saw in Section 2.2.6. But at this stage of the specfication, we will use this notation to indicate that we do not know anything about values of those types. Similarly, f has a type, but no proper implementation. We will declare types for the rest of the variables as well, and as we are not implementing any of them right now, we can just make one "dummy" implementation of a few of them in one go:

```
(x, deriv, ff, a, b, int) = undefined
```

We write *ff* for the capital F (to satisfy Haskell rules for variable names), *deriv* for the postfix prime, and *int* for the integral operator. On line 7 "values of x" hints at the type X for x and the way y is used indicates that it is to be seen as an alias for f (and thus must have the same type) As we have discussed above, the derivative normally preserves the type and thus we can write:

```
x :: X
y :: X → Y
```

$$
\begin{aligned}
&y \;= f\\
&y' :: X \to Y\\
&y' = \mathit{deriv}\, f\\
&\mathit{deriv} :: (X \to Y) \to (X \to Y)
\end{aligned}
$$

Next up (on line 11) is the "expression F" (which we write *ff*). It should take three arguments: y, y', x, and return "a quantity". We can invent a new type Z and write:

$$
\begin{aligned}
&\textbf{data}\ Z \quad \text{-- Probably also some subset of the real numbers}\\
&\mathit{ff} :: (X \to Y) \to (X \to Y) \to X \to Z
\end{aligned}
$$

Then we have the operation of definite integration, which we know should take two limits $a, b :: X$ and a function $X \to Z$. The traditional mathematics notation for integration uses an expression (in x) followed by dx, but we can treat that as a function *expr* binding x:

$$
\begin{aligned}
&a, b :: X\\
&\mathit{integral} = \mathit{int}\ a\ b\ \mathit{expr}\\
&\quad \textbf{where}\ \mathit{expr}\ x = \mathit{ff}\ y\ y'\ x\\
&\mathit{int} :: X \to X \to (X \to Z) \to Z
\end{aligned}
$$

Now we have reached a stage where all the operations have types and the type checker is happy with them. At this point it is possible to experiment with variations based on alternative interpretations of the text. For this kind of "refactoring" is very helpful to have the type checker to make sure the types still make sense. For example, we could write $\mathit{ff2} :: Y \to Y \to X \to Z$ as a variant of *ff* as long as we also change the expression in the integral:

$$
\begin{aligned}
&\mathit{ff2} :: Y \to Y \to X \to Z\\
&\mathit{ff2} = \mathit{undefined}\\
&\mathit{integral2} = \mathit{int}\ a\ b\ \mathit{expr}\\
&\quad \textbf{where}\ \mathit{expr}\ x = \mathit{ff2}\ y\ y'\ x\\
&\qquad\qquad \textbf{where}\ y \;= f\ x\\
&\qquad\qquad\qquad\quad y' = \mathit{deriv}\, f\ x
\end{aligned}
$$

Both versions (and a few more minor variations) would be fine as exam solutions, but something where the types don't match up would not be OK.

The kind of type inference we presented so far in this chapter becomes automatic with experience in a domain, but is very useful in the beginning.

3.7 Type classes

One difficulty when reading mathematics (and applying them to programming) is *overloading*. For our purposes, we say that a symbol is *overloaded* when its meaning depends on the type of the expressions that it applies to.

Consider for example the operator $(+)$. According to usual mathematical notation, one can typically use it to add integers, rational numbers, real numbers, complex numbers, etc. and it poses no difficulty. We explore the mathematical reasons in more detail in Section 4.2.2, but for now we will concentrate on the view of functional programming of this problem: one way to understand overloading is via *type classes*.

In Haskell both $4 == 3$ and $3.4 == 3.2$ typecheck because both integers and floating point values are member of the *Eq* class, which we can safely assume to be defined as follows:

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

The above declaration does two things. First, it defines a set of types which have equality test. One can tell the Haskell compiler that certain types belong to this set by using instance declarations, which additionally provide an implementation for the equality test. For example, we can make *Bool* member of the *Eq* using the following declaration:

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

(The Haskell compiler will in fact provide instances for primitive types).

Second, the *Eq* class declaration provides an operator $(==)$ of type $Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$. One can use the operator on any type a which belongs to the *Eq* set. This is expressed in general by a constraint $Eq\ a$ occuring before the $\Rightarrow$ symbol.

Instance declarations can also be parameterised on another instance. Consider for example:

```
instance Eq a ⇒ Eq [a] where
  (==) = ...   -- exercise
```

In the above, the expression $Eq\ a \Rightarrow Eq\ [a]$ means that for any type a which is already an instance of *Eq* we also make the type $[a]$ an instance of Eq. Thus, for example, by recursion we now have an infinite collection of instances of *Eq*: *Char*, $[Char]$, $[[Char]]$, etc.

3.7.1 Numeric operations

Haskell also provides a *Num* class, containing various numeric types (Int, Double, etc) with several operators $(+, *)$, etc). Unfortunately, the *Num* class was designed with more regard for implementation quirks than mathematical structure, and thus it is a poor choice for us. We take a more principled approach

instead, and define the following classes, which together, serve a similar role as *Num*, and which we study in more detail in Section 4.1:

```
class Additive a where
  zero :: a
  (+) :: a → a → a
class Additive a ⇒ AddGroup a where
  negate :: a → a
class Multiplicative a where
  one :: a
  (*) :: a → a → a
class Multiplicative a ⇒ MulGroup a where
  recip :: a → a    -- reciprocal
```

The operator names clash with the *Num* class, which we will avoid from now one in favour *Additive* and *Multiplicative*.

Exercise 3.2. Consider the exponentiation operator, which we can write $(\hat{})$. Taking advantage of the above classes, propose a possible type for it and sketch an implementation.

Solution: One possibility is $(\hat{}) :: Field\ a \Rightarrow a \to Int \to a$. For positive exponents, one can use repeated multiplication. For negative exponents, one can use repeated division.

TODO: JP: I could not understand what this was referring to: The "trick" of looking for an appropriate combinator with which to pre- or post-compose a function in order to makes type match is often useful.

Given these definitions, both expressions $4 + 3$ and $3.4 + 3.2$ typecheck.

3.7.2 Overloaded integer literals

We will spend some time explaining a convenient Haskell-specific syntactic shorthand which is very useful but which can be confusing: overloaded integers. In Haskell, every use of an integer literal like 2, 1738, etc., is actually implicitly an application of *fromInteger* to the literal typed as an *Integer*.

But what is *fromInteger*? It is a function that maps every value and operator in the numeric classes to an arbitrary type *a* in those classes. We can implement it as follows:

```
fromInteger :: (AddGroup a, Multiplicative a) ⇒ Integer → a
fromInteger n | n < 0 = negate (fromInteger (negate n))
              | n == 0 = zero
              | otherwise = one + fromInteger (n − 1)
```

Exercise 3.3. Define *fromRational* which does the same but also handles rational numbers and has the *MulGroup a* constraint.

This means that the same program text can have various meanings depending on the type of the context (but see also Exercise 4.3): The literal *three* $= 3$, for example, can be used as an integer, a real number, a complex number, or anything which belongs both to *AddGroup* and *Multiplicative*.

3.7.3 Structuring DSLs around type classes

Type classes are related to mathematical structures which, in turn, are related to DSLs.

As an example, consider again the DSL of expressions of one variables. We saw that such expressions can be represented by the shallow embedding $\mathbb{R} \to \mathbb{R}$. Using type classes, we can use the usual operators names instead of *evalPlus*, *evalTimes*, etc. We could write:

instance *Additive* $(\mathbb{R} \to \mathbb{R})$ **where**
$f + g = \lambda x \to f\ x + g\ x$
$zero\ \ = const\ zero$

The instance declaration of the method *zero* above looks recursive, but is not: *zero* is used at a different type on the left- and right-hand-side of the equal sign, and thus refers to two different functions. One the left-hand-side we define $zero :: \mathbb{R} \to \mathbb{R}$, while on the right-hand-side we use $zero :: \mathbb{R}$.

However, as one may suspect, for functions, we can use any domain and any numeric co-domain in place of $\mathbb{R}$. Therefore we prefer to define the following, more general instances:

instance *Additive a* $\Rightarrow$ *Additive* $(x \to a)$ **where**
$f + g\ \ \ = \lambda x \to f\ x + g\ x$
$zero = const\ zero$

instance *Multiplicative a* $\Rightarrow$ *Multiplicative* $(x \to a)$ **where**
$f * g\ \ \ = \lambda x \to f\ x * g\ x$
$one = const\ one$

instance *AddGroup a* $\Rightarrow$ *AddGroup* $(x \to a)$ **where**
$negate\ f = negate \circ f$

instance *MulGroup a* $\Rightarrow$ *MulGroup* $(x \to a)$ **where**
$recip\ f = recip \circ f$

instance *Algebraic a* $\Rightarrow$ *Algebraic* $(x \to a)$ **where**
$\sqrt{f} = \sqrt{\cdot} \circ f$

instance *Transcendental a* $\Rightarrow$ *Transcendental* $(x \to a)$ **where**
$\pi = const\ \pi$
$sin\ f = sin \circ f$
$cos\ f = cos \circ f$
$exp\ f = exp \circ f$

Here we extend our set of type-classes to cover algebraic and transcendental numbers. Together, these type classes represent an abstract language of abstract and standard operations, abstract in the sense that the exact nature of the elements involved is not important from the point of view of the type class, only from that of its implementation. What does matter for the class (but is not captured in the Haskell definition of the class), is the relationship between various operations (for example addition should distribute over multiplication).

These instances for functions allow us to write expressions which are very commonly used in math books, such as $f + g$ for the sum of two functions f and g, say $sin + cos :: Double \to Double$. Somewhat less common notations, like $sq * double :: Integer \to Integer$ are also possible. They have a consistent meaning: the same argument is passed to all functions in an expression. As another example, we can write $sin\hat{}2$, which the above instance assigns the following meaning:

$$sin\hat{}2 = \lambda x \to (sin\ x)\hat{}(const\ 2\ x) = \lambda x \to (sin\ x)\hat{}2$$

thus the typical math notation $\sin^2$ can work fine in Haskell, provided the above instances for functions, assuming a fixed argument. (Note that there is a clash with another common use of superscript for functions in mathematical texts: sometimes $f\hat{}n$ means *composition* of f with itself n times. With that reading sin^2 would mean $\lambda x \to sin\ (sin\ x)$.)

Exercise 3.4. Experiment with this feature using ghci, for example by evaluating $sin + cos$ at various points.

Something which may not be immediately obvious, but is nonetheless useful, is that all the above instances are of the form $C\ a \Rightarrow C\ (x \to a)$ and are therefore parametric. This means that, for example, given the instance $Additive\ a \Rightarrow Additive\ (x \to a)$ and the intstance $Additive\ \mathbb{R}$, we have that the types $a \to \mathbb{R}$, $a \to (b \to \mathbb{R})$, etc. are all instances of *Additive*. Consequently, we can use the usual mathematical operators for functions taking any number of arguments — provided that they match in number and types.

3.8 Computing derivatives

An important part of calculus is the collection of laws, or rules, for computing derivatives. They are provided by Adams and Essex [2010] as a series of theorems, starting at page 108 of their book. We we can summarize those as follows:

$(f + g)$ `'(x)` `= f'` $(x) + g'\ (x)$
$(f * g)$ `'(x)` `= f'` $(x)\ g\ (x) + f\ (x)\ g'\ (x)$
$(C * f)$ `'(x)` `= C*f'` (x)
$(f \circ g)\ (x) = f'\ (g\ (x)) * g'\ (x)$

(After a while, they Adams and Essex switch to differential notation, so we omit corresponding rules for trigonometric and exponential functions.) Using the notation $D\ f$ for the derivative of f and lifting the numeric operations to functions we can fill in a table of examples which can be followed to compute derivatives of many functions:

$$\begin{array}{ll} D\ (f+g) & = D\ f + D\ g \\ D\ (f*g) & = D\ f * g + f * D\ g \\ D\ id & = const\ 1 \\ D\ (const\ a) & = const\ 0 \\ D\ (f \circ g) & = (D\ f \circ g) * D\ g \quad \text{-- the chain rule} \\ D\ sin & = cos \\ D\ cos & = -sin \\ D\ exp & = exp \end{array}$$

and so on.

If we want to get a bit closer to actually implementing D we quickly notice a problem: if D has type $(\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$ have no way to turn the above specification into a program, because the program has no way of telling which of these rules we should apply. That is, given an extensional (semantic) function f, the only thing that we can ever do is to evaluate f at given points, and thus we cannot know if this function was written using a $+$, or *sin* or *exp* as outermost operation. The only thing that a derivative operator could do would be to numerically approximate the derivative, and that is not what we are exploring in this course. Thus we need to take a step back and change the type that we work on. Even though the rules in the table are obtained by reasoning semantically, using the definition of limit for functions (of type $\mathbb{R} \to \mathbb{R}$), they are really intended to be used on *syntactic* functions: abstract syntax trees *representing* the (semantic) functions.

TODO: JP: shallow

We observe that we can compute derivatives for any expression made out of arithmetical functions, standard functions, and their compositions. In other words, the computation of derivatives is based on a domain specific language (a DSL) of expressions (representing functions in one variable). Hence we can then implement the derivative of *FunExp* expressions using the rules of derivatives. Because the specification of derivation rules is already in the right format, the way to obtain this implementation may seem obvious, but we will go through the steps as a way to show the process in a simple case.

Our goal is want to implement a function $derive :: FunExp \to FunExp$ which makes the following diagram commute:

$$\begin{array}{ccc} FunExp & \xrightarrow{eval} & Func \\ \downarrow{\scriptstyle derive} & & \downarrow{\scriptstyle D} \\ FunExp & \xrightarrow{eval} & Func \end{array}$$

That is, we want the following equality to hold:

$$eval \circ derive = D \circ eval$$

in turn this means that for any expression $e :: FunExp$, we want

$$eval\ (derive\ e) = D\ (eval\ e)$$

For example, let us calculate the *derive* function for $Exp\ e$:

$$\begin{aligned}
&eval\ (derive\ (Exp\ e)) && = \{\text{- specification of } derive \text{ above -}\}\\
&D\ (eval\ (Exp\ e)) && = \{\text{- def. } eval \text{ -}\}\\
&D\ (exp\ (eval\ e)) && = \{\text{- def. } exp \text{ for functions -}\}\\
&D\ (exp \circ eval\ e) && = \{\text{- chain rule -}\}\\
&(D\ exp \circ eval\ e) * D\ (eval\ e) && = \{\text{- } D \text{ rule for } exp \text{ -}\}\\
&(exp \circ eval\ e) * D\ (eval\ e) && = \{\text{- specification of } derive \text{ -}\}\\
&(exp \circ eval\ e) * (eval\ (derive\ e)) && = \{\text{- def. of } eval \text{ for } Exp \text{ -}\}\\
&(eval\ (Exp\ e)) * (eval\ (derive\ e)) && = \{\text{- def. of } eval \text{ for } :*: \text{ -}\}\\
&eval\ (Exp\ e :*: derive\ e)
\end{aligned}$$

Therefore, the specification is fulfilled by taking

$$derive\ (Exp\ e) = Exp\ e :*: derive\ e$$

Similarly, we obtain

$$\begin{aligned}
derive\ (Const\ \alpha) &= Const\ 0\\
derive\ X &= Const\ 1\\
derive\ (e_1 :+: e_2) &= derive\ e_1 :+: derive\ e_2\\
derive\ (e_1 :*: e_2) &= (derive\ e_1 :*: e_2) :+: (e_1 :*: derive\ e_2)\\
derive\ (Exp\ e) &= Exp\ e :*: derive\ e
\end{aligned}$$

Exercise 3.5. Complete the *FunExp* type and the *eval* and *derive* functions.

3.9 Exercises

Exercise 3.6. For $f : \mathbb{R}^2 \to \mathbb{R}$ define D_1 and D_2 using only D. In more detail: let the type $F2 = \mathbb{R}^2 \to \mathbb{R}$ and $F1 = \mathbb{R} \to \mathbb{R}$. Then $D_1 : F2 \to F2$ and $D : F1 \to F1$. Start by defining helper functions: $fstFixed : a \to (b \to (a, b))$ and $sndFixed : b \to (a \to (a, b))$. Then use D and the helpers in the definitions of D_1 and D_2.

Exercise 3.7. To get a feeling for the Lagrange equations, let $L\ (t, q, v) = m * v\hat{}2\ /\ 2 - m * g * q$, compute *expand* w, perform the derivatives and check if the equation is satisfied for

- $w_1 = id$ or
- $w_2 = sin$ or
- $w_3 = (q0-) \circ (g*) \circ (/2) \circ (\hat{}2)$

Exercise 3.8. Consider the following text from Mac Lane's *Mathematics: Form and Function* (page 168):

> If $z = g(y)$ and $y = h(x)$ are two functions with continuous derivatives, then in the relevant range $z = g(h(x))$ is a function of x and has derivative
>
> $$z'(x) = g'(y) * h'(x)$$

Give the types of the elements involved (x, y, z, g, h, z', g', h', $*$ and $'$).

Exercise 3.9. Consider the following text from Mac Lane's *Mathematics: Form and Function* (page 182):

> In these cases one tries to find not the values of x which make a given function $y = f\ (x)$ a minimum, but the values of a given function $f\ (x)$ which make a given quantity a minimum. Typically, that quantity is usually measured by an integral whose integrand is some expression F involving both x, values of the function $y = f\ (x)$ at interest and the values of its derivatives - say an integral
>
> $$\int_a^b F(y, y', x)dx, \quad y = f(x).$$

Give the types of the variables involved (x, y, y', f, F, a, b) and the type of the four-argument integration operator:

$$\int_{\cdot}^{\cdot} \cdot d \cdot$$

Exercise 3.10. In the simplest case of probability theory, we start with a *finite*, non-empty set Ω of *elementary events*. *Events* are subsets of Ω, i.e. elements of the powerset of Ω, (that is, $\mathcal{P}\Omega$). A *probability function* P associates to each event a real number between 0 and 1, such that

- $P\,\varnothing = 0$, $P\,\Omega = 1$
- A and B are disjoint (i.e., $A \cap B = \varnothing$), then: $P\,A + P\,B = P\,(A \cup B)$.

Conditional probabilities are defined as follows [Stirzaker, 2003]:

> Let A and B be events with $P\,B > 0$. given that B occurs, the *conditional probability* that A occurs is denoted by $P\,(A \mid B)$ and defined by
> $P\,(A \mid B) = P\,(A \cap B)\,/\,P\,B$

1. What are the types of the elements involved in the definition of conditional probability?
   $(P, \cap, /, |)$
2. In the 1933 monograph that set the foundations of contemporary probability theory, Kolmogorov used, instead of $P\,(A \mid B)$, the expression $P_B\,A$. Type this expression. Which notation do you prefer (provide a *brief* explanation).

Exercise 3.11. (Note that this exam question is now included as an example in this chapter, see Section 3.4. It is kept here in case you want to check if you remember it!)

Consider the following text from page 169 of Mac Lane [1968]:

> [...] a function $z = f\,(x, y)$ for all points (x, y) in some open set U of the cartesian (x, y)-plane. [...] If one holds y fixed, the quantity z remains just a function of x; its derivative, when it exists, is called the *partial derivative* with respect to x. Thus at a point (x, y) in U this derivative for $h \neq 0$ is
>
> $$\partial z/\partial x = f'_x(x, y) = \lim_{h \to 0}(f(x + h, y) - f(x, y))/h$$

What are the types of the elements involved in the equation on the last line? You are welcome to introduce functions and names to explain your reasoning.

Exercise 3.12. Multiplication for matrices (from the matrix algebra DSL).

Consider the following definition, from "Linear Algebra" by Donald H. Pelletier:

Definition: If A is an $m \times n$ matrix and B is an $n \times p$ matrix, then the *product*, AB, is an $m \times p$ matrix; the $(i, j)^{th}$ entry of AB is the sum of the products of the pairs that are obtained when the entries from the i^{th} row of the left factor, A, are paired with those from the j^{th} column of the right factor, B.

1. Introduce precise types for the variables involved: A, m, n, B, p, i, j. You can write *Fin n* for the type of the values $\{0, 1, ..., n-1\}$.

2. Introduce types for the functions *mul* and *proj* where AB = *mul A B* and *proj i j M* = "take the $(i, j)^{th}$ entry of M". What class constraints (if any) are needed on the type of the matrix entries in the two cases?

3. Implement *mul* in Haskell. You may use the functions *row* and *col* specified by *row i M* = "the i^{th} row of M" and *col j M* = "the j^{th} column of M". You don't need to implement them and here you can assume they return plain Haskell lists.

Exercise 3.13. (Extra material outside the course.) In the same direction as the Lagrangian case study in Section 3.5 there are two nice blog posts about Hamiltonian dynamics: one introductory and one more advanced. It is a good exercise to work through the examples in these posts.

Chapter 4

Compositionality and Algebras

```
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE FlexibleInstances, GeneralizedNewtypeDeriving #-}
{-# LANGUAGE ConstraintKinds, RebindableSyntax #-}
module DSLsofMath.W04 where
import Prelude hiding (Monoid, even, Num (..), exp)
import DSLsofMath.FunExp hiding (eval)
import DSLsofMath.Algebra
```

Algebraic structures are fundamental to the structuralist point of view in mathematics, which emphasises relations between objects rather than the objects themselves and their representations. Furthermore, each mathematical domain has its own fundamental structures. Once these structures have been identified, one tries to push their study as far as possible *on their own terms*, without picking any particular representation (which may have richer structure than the one we want to study). For example, in group theory, one starts by exploring the consequences of just the group structure, rather than introducing any particular group (like integers) which have (among others) an order structure and monotonicity.

Furthermore, mappings or (translations) between such structures becomes an important topic of study.

When such mappings preserve the structure, they are called *homomorphisms*. As two examples, we have the homomorphisms *exp* and *log*, specified as follows:

```
exp :: ℝ    → ℝ_{>0}
exp 0       = 1              -- e^0 = 1
```

$$
\begin{array}{lll}
exp\ (a+b) & = exp\ a * exp\ b & \text{-- } e^{a+b} = e^a e^b \\
log & : \mathbb{R}_{>0} \to \mathbb{R} & \\
log\ 1 & = 0 & \text{-- } \log 1 = 0 \\
log\ (a*b) & = log\ a + log\ b & \text{-- } \log(ab) = \log a + \log b
\end{array}
$$

What we recognize as the familiar laws of exponentiation and logarithms arise from homomorphism conditions, which relate the additive and multiplicative structures of reals and positive reals.

Additionally, homomorphisms play a crucial role when relating an abstract syntax (a datatype), and a semantic domain (another type) via an evaluation function between them (the semantics). In this chapter we will explain the notions of algebraic struture and homomorphism in detail and show applications both in mathematics and DSLs in general.

4.1 Algebraic Structures

What is an algebraic structure? Let's turn to Wikipedia as a starting point:

> In universal algebra, an algebra (or algebraic structure) is a set A together with a collection of operations on A (of finite arity) and a collection of axioms which those operation must satisfy.

The fact that a type a is equipped with operations is conveniently captured in Haskell using a type class (Section 3.7).

Example A particularly pervasive structure is that of monoids. A monoid is an algebra which has an associative operation *op* and a *unit*:

```
class Monoid a where
  unit :: a
  op   :: a → a → a
```

The laws cannot be obviously captured in the class, but can be formulated as the following equations:

$$
\begin{array}{l}
\forall\, x : a.\ (unit\ `op`\ x == x \wedge x\ `op`\ unit == x) \\
\forall\, x, y, z : a.\ (x\ `op`\ (y\ `op`\ z) == (x\ `op`\ y)\ `op`\ z)
\end{array}
$$

The first law ensures that *unit* is indeed the unit of *op* and the second law is the familiar associativity law for *op*.

Example Examples of monoids include numbers with addition, $(\mathbb{R}, 0, (+))$, positive numbers with multiplication $(\mathbb{R}_{>0}, 1, (*))$, and even endofunctions with composition $(a \to a, id, (\circ))$. (An "endofunction", also known as "endomorphism" is a function of type $X \to X$ for some set X.)

Exercise 4.1. Define the above monoids and check that the laws are satisfied.

Example To make this a bit more concrete, here are two examples of monoids in Haskell: the additive monoid *ANat* and the multiplicative monoid *MNat*.

```
newtype ANat      = A Natural  deriving (Show, Eq)
instance Monoid ANat where
  unit            = A 0
  op (A m) (A n)  = A (m + n)
newtype MNat      = M Natural deriving (Show, Eq)
instance Monoid MNat where
  unit            = M 1
  op (M m) (M n) = M (m * n)
```

In Haskell there can ever be at most one instance of a given class for a given type, so we cannot define two **instance** *Monoid Natural*: we must make a **newtype** whose role is to indicate which of the two possible monoids (additive or applicative) applies in a given context. But, in mathematical texts the constructors *M* and *A* are usually omitted, and instead the names of the operations suggest which of the monoids one is referring to. To be able to conform to that tradition we can define two separate classes, one for the additive and one for the multiplicative monoids, as follows.

```
class Additive a where
  zero :: a
  (+) :: a → a → a
class Multiplicative a where
  one :: a
  (*) :: a → a → a
```

This is what we have done in Section 3.7.1.

Example Groups and rings Another important structure are groups, which are monoids augmented with an inverse. To continue our mathematically-grounded *Num* replacement, we have also defined the additive group as follows:

```
class Additive a ⇒ AddGroup a where
  negate :: a → a
```

Groups demand that the inverse (called *negate* for the additive group) act like an inverse. Namely, applying the operation to an element and its inverse should yield the unit of the group. Thus, for the additive group, the laws look like this:

$$negate\ a + a = zero$$
$$a + negate\ a = zero$$

And thus we can define subtraction as

$$a - b = a + negate\ b$$

Finally, when the additive monoid is abelian (commutative) and addition distributes over multiplication, we have a *Ring*. As always we cannot conveniently specify laws in Haskell typeclasses and thus define *Ring* simply as the conjunction of *AddGroup* and *Multiplicative*:

$$\textbf{type}\ Ring\ a = (AddGroup\ a, Multiplicative\ a)$$

With that, we have completed the structural motivation of our replacement for the *Num* class!

TODO: JP: Show some instances?

4.2 Homomorphisms

The Wikipedia definition of homomorphism states that "A homomorphism is a structure-preserving map between two algebraic structures of the same type".

4.2.1 (Homo)morphism on one operation

As a stepping stone to capture this idea, we can define a ternary predicate H_2. The first argument h, is the map. The second (Op) and third (op) arguments correspond to the algebraic structures.

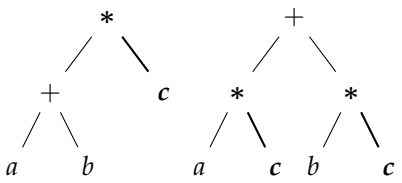
$$H_2\ (h, Op, op) = \forall\ x.\ \forall\ y.\ h\ (Op\ x\ y) == op\ (h\ x)\ (h\ y)$$

If $H_2\ (h, Op, op)$ holds, we say that $h : A \to B$ is a homomorphism from $Op : A \to A \to A$ to $op : B \to B \to B$. Or that h is a homomorphism from Op to op. Or even that h is a homomorphism from A to B if the operators are clear from the context. We have seen several examples in earlier chapters:

1. in Section 1.3 we saw that $evalE : ComplexE \to ComplexD$ is a homomorphism from the syntactic operator *Plus* to the corresponding semantic operator *plusD*.

2. in Chapter 2 we saw de Morgan's laws which can be stated as $H_2\ (\neg, (\wedge), (\vee))$ and $H_2\ (\neg, (\vee), (\wedge))$.
3. in Section 3.1.1 we saw that $eval : FunExp \to Func$ is a homomorphism from syntactic $(:*:)$ to semantic $(*)$ for functions
4. If $(*)$ distributes over $(+)$ for some type A then $(*c) : A \to A$ is a homomorphism from $(+)$ to $(+)$: $H_2\ ((*c), (+), (+))$.

To see how this last item plays out, it can be helpful to study the syntax trees of the left and right hand sides of the distributive law: $((a + b) * c = (a * c) + (b * c))$. We observe that $(*c)$ is "pushed down" to both a and b:



Exercise 4.2. Expand the definition of H_2 in each case and check that the obtained conditions hold.

4.2.2 Homomorphism on structures

So far our definition of homomorphism takes the rather limited view that a single operation is transformed. Usually, homomorphisms map a whole *structure*.

Back to Wikipedia:

> More formally, a homomorphism between two algebras A and B is a function $h : A \to B$ from the set A to the set B such that, for every operation f_A of A and corresponding f_B of B (of arity, say, n), $h\ (f_A\ (x_1, ..., x_n)) = f_B\ (h\ (x_1), ..., h\ (x_n))$.

In our Haskell interpretation, the above would mean that we have $H_2\ (h, op, op)$ for every function op in a given class C. More precisely the first occurence of op comes from the $C\ A$ instance and the second one from $C\ B$.

Example The general monoid homomorphism conditions for $h : A \to B$ are:

```
h unit       = unit          -- h takes units to units
h (x `op` y) = h x `op` h y  -- and distributes over op (for all x and y)
```

Note that both *unit* and *op* have different types on the left and right hand sides. On the left they belong to the monoid $(A, unit_A, op_A)$ and on the right the belong to $(B, unit_B, op_B)$.

Example Hence, the function *exp* is a monoid homomorphism from (*Real*,0,+) to ($\mathbb{R}_{>0}$,1,*).

$$\begin{array}{lll} exp : \mathbb{R} & \rightarrow \mathbb{R}_{>0} & \\ exp\ 0 & = 1 & \text{-- } e^0 = 1 \\ exp\ (a+b) & = exp\ a * exp\ b & \text{-- } e^{a+b} = e^a e^b \end{array}$$

In the above example, we have simply checked the homomorphism conditions for the exponential function. But we can try to go the other way around: knowing that a function h is homomorphism, what is the kind of function that h can be?

Example As an example, let us can characterise the homomorphisms from *ANat* to *MNat* (from Section 4.1).

Let $h : ANat \rightarrow MNat$ be a monoid homomorphism. Then it must satisfy the following conditions:

$$\begin{array}{ll} h\ 0 & = 1 \\ h\ (x+y) & = h\ x * h\ y \quad \text{-- for all } x \text{ and } y \end{array}$$

For example $h\ (x+x) = h\ x * h\ x = (h\ x)\hat{}2$ which for $x = 1$ means that $h\ 2 = h\ (1+1) = (h\ 1)\hat{}2$.

More generally, every natural number can is equal to the sum of n ones: $1 + 1 + ... + 1$. Therefore

$$\begin{array}{l} h\ n = h\ (1 + ... + 1) \\ \quad = h\ 1 * \ ... * h\ 1 \\ \quad = (h\ 1)\hat{}n \end{array}$$

That is, every choice of $h\ 1$ induces a homomorphism from *ANat* to *MNat*. This means that the value of the function h, for any natural number, is fully determined by its value for 1.

In other words, we know that every h (homomorphism from *ANat* to *MNat*) is of the form

$$h\ n = a\hat{}n$$

for a given natual number $a = h\ 1$. So, the set of homomorphisms between the additive monoid and the multiplicative monoid is the set exponential functions, one for every base a.

Exercise 4.3. Assume an arbitrary Ring-homomorphism f from *Integer* to an arbitrary type a. Prove f == *fromInteger*, provided the definition in Section 3.7.2.

Solution: The homomorphism conditions include:

$$
\begin{aligned}
&f\ zero = zero \\
&f\ (one + x) = one + (f\ x) \\
&f\ (negate\ x) = negate\ (f\ x)
\end{aligned}
$$

By substitution we get the following equations:

$$
\begin{aligned}
&f\ zero = zero \\
&f\ x = one + (f\ (x - one)) \\
&f\ x = negate\ (f\ (negate\ x))
\end{aligned}
$$

These are compatible with the behaviour of *fromInteger*, but they also completely fix the behaviour of f if x is an integer, because it can either be zero, positive or negative.

Other homomorphisms

Exercise 4.4. Show that *const* is a homomorphism

TODO: JP: What structure are we talking about here? Also, if the additive one (minus zero), is it reasonable to expect that we remember that instance for functions here?

TODO: JP: This is the first time that this name is used. Also, isn't it simply the homomorphism law?

Solution: The distribution law can be shown as follows:

$$
\begin{aligned}
&h\ a + h\ b && = \{\text{- } h = const \text{ in this case -}\} \\
&const\ a + const\ b && = \{\text{- By def. of } (+) \text{ on functions -}\} \\
&(\lambda x \to const\ a\ x + const\ b\ x) && = \{\text{- By def. of } const\text{, twice -}\} \\
&(\lambda x \to a + b) && = \{\text{- By def. of } const \text{ -}\} \\
&const\ (a + b) && = \{\text{- } h = const \text{ -}\} \\
&h\ (a + b)
\end{aligned}
$$

We now have a homomorphism from values to functions, and you may wonder if there is a homomorphism in the other direction. The answer is "Yes, many".

Exercise 4.5. Show that *apply c* is a homomorphism for all c, where $apply\ x\ f = f\ x$.

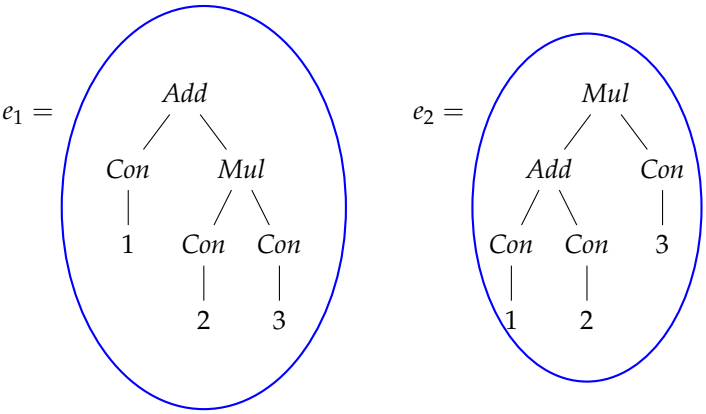
Exercise 4.6. Extend the exponential-logarithm morphism to relate *AddGroup* and *MulGroup*.

4.3 Compositional semantics

4.3.1 Compositional functions are homomorphisms

Consider a datatype of very simple integer expressions:

```
data E = Add E E | Mul E E | Con Integer deriving Eq
e1, e2 :: E                                  -- 1 + 2 * 3
e1 = Add (Con 1) (Mul (Con 2) (Con 3))       -- 1 + (2 * 3)
e2 = Mul (Add (Con 1) (Con 2)) (Con 3)       -- (1 + 2) * 3
```



As the reader may have guessed, the natural evaluator $eval : E \to Integer$ (defined later) is a homomorphism from *Add* to $(+)$ and from *Mul* to $(*)$. But to practice the definition of homomorphism we will here check if *even* or *isPrime* is a homomorphism from *E* to *Bool*.

Let's try to define $even : E \to Bool$ with the usual induction pattern :

```
even (Add x y) = evenAdd (even x) (even y)
even (Mul x y) = evenMul (even x) (even y)
even (Con c)   = evenCon c

evenAdd :: Bool → Bool → Bool
evenMul :: Bool → Bool → Bool
evenCon :: Integer → Bool
```

Note that *even* throws away lots of information: the domain is infinite and the range is a two-element set. This information loss could make it difficult to define the helper functions *evenAdd*, etc. because they only get to work on the small range. Still, in this case we are lucky: we can use the "parity rules" taught in elementary school: even plus even is even, etc. In code we simply get:

```
evenAdd = (==)
evenMul = (∨)
evenCon = (0 ==) ∘ ('mod'2)
```

[1]

Exercise 4.7. Exercise: prove $H_2\ (even, Add, evenAdd)$ and $H_2\ (even, Mul, evenMul)$.

[1] A perhaps more natural alternative would be to taken *odd* instead of *even* as the homomorphism. You can try it out as an exercise.

4.3.2 An example of a non-compositional function

Let's now try to define $isPrime : E \rightarrow Bool$ in the same way to see a simple example of a non-compositional function. In this case it is enough to just focus on one of the cases to already see the problem:

$$
\begin{aligned}
&isPrime\ (Add\ x\ y) = isPrimeAdd\ (isPrime\ x)\ (isPrime\ y)\\
&isPrimeAdd :: Bool \rightarrow Bool \rightarrow Bool\\
&isPrimeAdd = error\ \texttt{"Can this be done?"}
\end{aligned}
$$

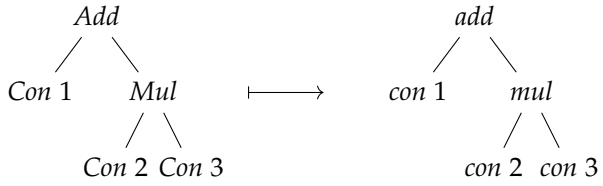
As before, if we can define *isPrimeAdd*, we will get $H_2\ (isPrime, Add, isPrimeAdd)$ "by construction" But it is not possible for *isPrime* to both satisfy its specification and $H_2\ (isPrime, Add, isPrimeAdd)$. (To shorten the calculation we write just *n* for *Con n*.)

$$
\begin{aligned}
&\ \ False\\
&= \{\text{- By spec. of } isPrime \text{ (four is prime). -}\}\\
&\ \ isPrime\ (Add\ 2\ 2)\\
&= \{\text{- by } H_2 \text{ -}\}\\
&\ \ isPrimeAdd\ (isPrime\ 2)\ (isPrime\ 2)\\
&= \{\text{- By spec. of } isPrime \text{ (two is prime). -}\}\\
&\ \ isPrimeAdd\ (isPrime\ 2)\ True\\
&= \{\text{- By spec. of } isPrime \text{ (three is also prime). -}\}\\
&\ \ isPrimeAdd\ (isPrime\ 2)\ (isPrime\ 3)\\
&= \{\text{- by } H_2 \text{ -}\}\\
&\ \ isPrime\ (Add\ 2\ 3)\\
&= \{\text{- By spec. of } isPrime \text{ (five is prime). -}\}\\
&\ \ True
\end{aligned}
$$

But because we also know that $False \neq True$, we have a contradiction. Thus we conclude that *isPrime* is *not* a homomorphism from *E* to *Bool*, regardless of the choice of the operator corresponding to addition.

4.4 Folds

In general, for a syntax *Syn*, and a possible semantics (a type *Sem* and an *eval* function of type $Syn \rightarrow Sem$), we call the semantics *compositional* if we can implement *eval* as a fold. Informally a "fold" is a recursive function which replaces each abstract syntax constructor *Ci* of *Syn* with its semantic interpretation c_i — but without doing any other change in the structure. In particular, moving around constructors is forbidden. For example, in our datatype *E*, a compositional semantics means that *Add* maps to *add*, $Mul \mapsto mul$, and $Con \mapsto con$ for some "semantic functions" *add*, *mul*, and *con*.



As an example we can define a general *foldE* for the integer expressions:

```
foldE :: (s → s → s) → (s → s → s) → (Integer → s) → (E → s)
foldE add mul con = rec
  where rec (Add x y) = add (rec x) (rec y)
        rec (Mul x y) = mul (rec x) (rec y)
        rec (Con i)   = con i
```

Notice that *foldE* has three function arguments corresponding to the three constructors of *E*. The "natural" evaluator to integers is then easy to define:

```
evalE1 :: E → Integer
evalE1 = foldE (+) (*) id
```

and with a minimal modification we can also make it work for other numeric types:

```
evalE2 :: Ring a ⇒ E → a
evalE2 = foldE (+) (*) fromInteger
```

Another thing worth noting is that if we replace each abstract syntax constructor with itself we get an identity function, sometimes known as a "deep copy":

```
idE :: E → E
idE = foldE Add Mul Con
```

Finally, it is useful to capture the semantic functions (the parameters to the fold) in a type class:

```
class IntExp t where
  add :: t → t → t
  mul :: t → t → t
  con :: Integer → t
```

In this way we can turn the arguments to the fold into a constraint on the return type:

```
foldIE :: IntExp t ⇒ E → t
foldIE = foldE add mul con
instance IntExp E where
```

$$
\begin{aligned}
&\quad add = Add\\
&\quad mul = Mul\\
&\quad con = Con\\
&\textbf{instance } IntExp\ Integer \textbf{ where}\\
&\quad add = (+)\\
&\quad mul = (*)\\
&\quad con = id\\
&idE' :: E \to E\\
&idE' = foldIE\\
&evalE' :: E \to Integer\\
&evalE' = foldIE
\end{aligned}
$$

Additionally *IntExp* is the underlying algebraic structure of the fold. The function *foldIE* is a homomorphism which maps the *IntExp E* instance to another (arbitrary) instance *IntExp e*. This is what a fold is in general. Given a structure *C*, a fold is a homomorphism from a realisation of *C* as a data-type. We can note at this point that a class *C a* can be realised as a datatype only if all the functions of **class** *C a* return *a*. (Otherwise the constructors could create another type; and so they are not constructors any more.) This condition was satisfied in the case of our **class** *IntExp t*: all function signatures end with ... $\to t$. When this condition is satisfied, we say that the class is an *algebra* — not just any algebraic structure.[2]

4.4.1 Even folds can be wrong!

When working with expressions it is often useful to have a "pretty-printer" to convert the abstract syntax trees to strings like `"1+2*3"`.

$$pretty :: E \to String$$

We can view *pretty* as an alternative *eval* function for a semantics using *String* as the semantic domain instead of the more natural *Integer*. We can implement *pretty* in the usual way as a fold over the syntax tree using one "semantic constructor" for each syntactic constructor:

$$
\begin{aligned}
&pretty\ (Add\ x\ y) = prettyAdd\ (pretty\ x)\ (pretty\ y)\\
&pretty\ (Mul\ x\ y) = prettyMul\ (pretty\ x)\ (pretty\ y)\\
&pretty\ (Con\ c)\quad\ = prettyCon\ c\\
&prettyAdd :: String \to String \to String\\
&prettyMul :: String \to String \to String\\
&prettyCon :: Integer \to String
\end{aligned}
$$

We can also see *String* and *pretty* as an instance of the *IntExp* class:

[2] Indeed, this terminology can be confusing.

```
instance IntExp String where
  add = prettyAdd
  mul = prettyMul
  con = prettyCon
pretty' :: E → String
pretty' = foldIE
```

Now, if we try to implement the semantic constructors without thinking too much we would get the following:

```
prettyAdd xs ys = xs ++ "+" ++ ys
prettyMul xs ys = xs ++ "*" ++ ys
prettyCon c     = show c
p1, p2 :: String
p1 = pretty e1
p2 = pretty e2
trouble :: Bool
trouble = p1 == p2
```

Note that e_1 and e_2 are not equal, but they still pretty-print to the same string. This means that *pretty* is doing something wrong: the inverse, *parse*, is ambiguous. There are many ways to fix this, some more "pretty" than others. One way to characterise the issue is that some information is lost in the translation: *pretty* is not invertible.

Thus, we can see that a function can be a homomorphism and still be "wrong".

For the curious One solution to the problem with parentheses is to create three (slightly) different functions intended for printing in different contexts. The first of them is for the top level, the second for use inside *Add*, and the third for use inside *Mul*. These three functions all have type $E \to String$ and can thus be combined with the tupling transform into one function returning a triple: $prVersions :: E \to (String, String, String)$. The result is the following:

```
prTop :: E → String
prTop e = let (pTop, _, _) = prVersions e
          in pTop
type ThreeVersions = (String, String, String)
prVersions :: E → ThreeVersions
prVersions = foldE prVerAdd prVerMul prVerCon
prVerAdd :: ThreeVersions → ThreeVersions → ThreeVersions
prVerAdd (xTop, xInA, xInM) (yTop, yInA, yInM) =
  let s = xInA ++ "+" ++ yInA     -- use InA because we are "in Add"
  in (s, paren s, paren s)        -- parens needed except at top level
prVerMul :: ThreeVersions → ThreeVersions → ThreeVersions
```

```
prVerMul (xTop, xInA, xInM) (yTop, yInA, yInM) =
  let s = xInM ++ "*" ++ yInM   -- use InM because we are "in Mul"
  in (s, s, paren s)            -- parens only needed inside Mul
prVerCon :: Integer → ThreeVersions
prVerCon i =
  let s = show i
  in (s, s, s)                  -- parens never needed
paren :: String → String
paren s = "(" ++ s ++ ")"
```

Exercise: Another way to make this example go through is to refine the semantic domain from *String* to *Precedence* → *String*. This can be seen as another variant of the result after the tupling transform: if *Precedence* is an n-element type then *Precedence* → *String* can be seen as an n-tuple. In our case a three-element *Precedence* would be enough.

4.5 Initial and Free Structures

In Section 4.4 we started with a data-type, and derived an algebraic structure (more precisely an algebra) from it. But we can go in the other direction: start with an algebra and derive a datatype which capture the structure of the algebra, but nothing more. This is representation is called the initial algebra.

The Initial Monoid

As a first example, consider an initial algebra for monoids (an initial monoid for short).

We know that we have at least one object: the *unit*. But we can also construct more objects using *op*: *unit* ‘*op*‘ *unit*, *unit* ‘*op*‘ (*unit* ‘*op*‘ *unit*), etc. So a draft for the initial monoid could be:

```
data M where
  Unit :: M
  Op :: M → M → M
```

or:

```
data M = Unit | Op M M
```

But we also have the unit laws, which in particular tell us that *unit* ‘*op*‘ *unit* == *unit*. So, in fact, we are left with a single element: the *unit*. A representation of the initial monoid is then simply:

$$\textbf{data}\ M = Unit$$

As one might guess, there are not many interesting applications of the initial monoid, so let us consider another structure.

The Initial Ring

Gathering all function in various type classes, we find that a *Ring* corresponds to the following algebra — again we start by ignoring laws:

$$\begin{aligned}
&zero :: a\\
&(+) :: a \to a \to a\\
&negate :: a \to a\\
&one :: a\\
&(*) :: a \to a \to a
\end{aligned}$$

In this case, we can start with *zero* and *one*. As before, using addition on *zero* or multiplication on *one* would yield no more elements. But we can use addition on *one*, and get $one + one$, $one + one + one$, etc. Because of associativity, we don't have to — and ought not to — write parentheses. Let's write an addition of n ones as n. What about multiplying? Are we going to get more kinds of numbers from that? No, because of distributivity. For example:

$$(one + one) * (one + one) == one + one + one + one$$

By following this line of reasoning to its conclusion, we will find that the initial *Ring* is the set of integers.

4.5.1 A general initial structure

In Haskell, the type $C\ a \Rightarrow a$ is a generic way to represent the initial algebra for a class C. To get a more concrete feeling for this, let us return to *IntExp*, and consider a few values of type $IntExp\ a \Rightarrow a$.

$$\begin{aligned}
&seven :: IntExp\ a \Rightarrow a\\
&seven = add\ (con\ 3)\ (con\ 4)\\
&testI :: Integer\\
&testI = seven\\
&testE :: E\\
&testE = seven\\
&testP :: String\\
&testP = seven\\
&check :: Bool
\end{aligned}$$

```
check = and [testI == 7
            ,testE == Add (Con 3) (Con 4)
            ,testP == "3+4"
            ]
```

By defining a class *IntExp* (and some instances) we can use the methods (*add*, *mul*, *con*) of the class as "smart constructors" which adapt to the context. An overloaded expression, like *seven* :: *IntExp a* ⇒ *a*, which only uses these smart constructors can be instantiated to different types, ranging from the syntax tree type *E* to any possible semantic interpretations (like *Integer*, *String*, etc.). In general, for any given value *x* of type *IntExp a* ⇒ *a*, all the variants of *x* instanciated at different types are guaranteed to be related by homomorphisms, because one simply replaces *add*, *mul*, *con* by valid instances.

TODO: JP: what is that?

The same kind of reasoning justifies the overloading of Haskell integer literals. They can be given the type *Ring a* ⇒ *a*, and doing in a mathematically meaningful way, because *Ring a* ⇒ *a* is the initial algebra for *Ring*.

4.5.2 Free Structures

Another useful kind of are free structures. They are similar to initial structures, but they also allow to embed an arbitrary set of *generators G*. That is, it is as if we would throw an additional *generate* function in the algebra:

```
class Generate a where
  generate :: G → a
```

(We could parameterize the class over an abstract generator set *g*, but will refain from it to avoid needless complications.)

Free Monoid

As an example, consider the free monoid. Our algebra has the following signature:

```
generate :: G → a
op :: a → a → a
unit :: a → a → a
```

As a first version, we can convert each function to a constructor and obtain the type:

```
data FreeMonoid g = Unit | Op (FreeMonoid g) (FreeMonoid g) | Generator g deriving Show
instance Monoid (FreeMonoid g) where
```

$$unit = Unit$$
$$op = Op$$

Let us consider a fold for the above *FreeMonoid*. We can write its type as follows:

$$evalM :: (Monoid\ a, Generate\ a) \Rightarrow (FreeMonoid\ G \to a)$$

but we can also drop the *Generate* constraint and take the *generate* method as an explicit argument:

$$evalM :: Monoid\ a \Rightarrow (G \to a) \to (FreeMonoid\ G \to a)$$

This form is similar to the evaluators of expressions with variables of type G, which we have seen for example in Section 3.2.2. Once given a function $f :: G \to a$, the homomorphism condition forces *evalM* to be a fold:

$$\begin{aligned} evalM\ f\ Unit &= unit \\ evalM\ f\ (Op\ e_1\ e_2) &= op\ (evalM\ f\ e_1)\ (evalM\ f\ e_2) \\ evalM\ f\ (Generator\ x) &= f\ x \end{aligned}$$

However, before being completely satisfied, we must note that the *FreeMonoid* representation is ignoring monoid laws. By following the same kind of reasoning as before, we find that we only have in fact only two distinct forms for the elements of the free monoid:

- *unit*
- *generate* x_1 ‘*op*‘ *generate* x_2 ‘*op*‘ ... ‘*op*‘ *generate* x_n

Because of associativity we have no parentheses in the second form; and because of the unit laws we need not have *unit* composed with *op* either.

Thus, the free monoid over a generator set G is a list of G.

We seemingly also ignored the laws when defining *evalM*. Is this a problem? For example, is it possible that e_1 ‘*Op*‘ $(e_2$ ‘*Op*‘ *e3*$)$ and $(e_1$ ‘*Op*‘ $e_2)$ ‘*Op*‘ *e3* which are by monoid laws equal, map to different values? By definition of *evalM*, the condition reduces to checking *evalM f* e_1 ‘*op*‘ $($*evalM f* e_2 ‘*op*‘ *evalM f e3*$)$ == $($*evalM f* e_1 ‘*op*‘ *evalM f* $e_2)$ ‘*op*‘ *evalM f e3*. But then, this turns out to be satisfied if *op* is associative. In sum, *evalM* will be correct if the target *Monoid* instance satisfies the laws. This is true in general: folds are always homomorphisms even if the datatype representation that they work on ignore laws.

Functions of one variable as algebras

Earlier we have used (many variants of) data types for arithmetic expressions. Using the free construction, we can easily conceive a suitable type for any such expression language. For example, the type for arithmetic expressions with $+, -, *$ and variables is the free *Ring* with the set of variables as generator set.

Let us consider again our deep-embedding for expressions of one variable Section 3.1.1. According to our analysis, it should be a free structure, and because we have only one variable, we can take the generator set (G) to be the unit type.

```
type G = ()
instance Generate FunExp where
  generate () = Id
instance Additive FunExp where
  (+) = (:+:)
  zero = Const 0
instance Multiplicative FunExp where
  (*) = (:*:)
  one = Const 1
instance AddGroup FunExp where    -- ...
instance MulGroup FunExp where    -- ...
instance Transcendental FunExp where    -- ...
```

and so on.

Exercise 4.8. Complete the instances for *FunExp* (possibly extending the datatype).

Remark: to translate the $Const :: \mathbb{R} \to FunExp$ constructor we need a way to map any $\mathbb{R}$ to the above structures. We know how to do that for integers, (*fromInteger*). For this exercise you can restrict yourself to floating point representation of constants, and use *recip* (from *MulGroup*) to map them to fractions.

We can then check that the evaluator is compositional. For instance, we have

```
eval (e1 :*: e2) = eval e1 * eval e2
eval (Exp e)     = exp (eval e)
```

etc.

We can now also generalise the type of evaluator as follows:

```
type OneVarExp a = (Generate a, Transcendental a)
eval :: OneVarExp a => FunExp -> a
```

With this class in place we can define generic expressions using smart constructors just like in the case of *IntExp* above. For example, we can define

```
varX :: OneVarExp a ⇒ a
varX = generate ()
twoexp :: OneVarExp a ⇒ a
twoexp = 2 ∗ exp varX    -- recall the implicit fromInteger
```

and instantiate *twoexp* to either syntax or semantics:

```
testFE :: FunExp
testFE = twoexp
testFu :: Func
testFu = twoexp
```

provided a suitable instance for *Generate Func*:

```
instance Generate Func where
  generate () = id
```

Exercise 4.9. Find another instance of *OneVarExp*.

TODO: JP: What is the intent of the exercise?

As before, we can always define a homomorphism from *FunExp* to *any* instance of *OneVarExp*, in a unique way, using the fold pattern. This is because the datatype *FunExp* is an initial *OneVarExp*. Working with *OneVarExp a ⇒ a* can be more economical than using *FunExp*: one does not need any *eval*.

The DSL of expressions, whose syntax is given by the type *FunExp*, turns out to be almost identical to the DSL defined via type classes in Section 3.7. The correspondence between them is given by the *eval* function. The difference between the two implementations is that the first one separates more cleanly from the semantical one. For example, :+: *stands for* a function, while + *is* that function.

TODO: JP: Have we said that we defined a DSL?

4.5.3 ∗A generic Free construction

We can use the same trick as for initial algebras to construct free algebras: $(C\ a, Generate\ a) \Rightarrow a$ is the free C-structure. However, it is often more convenient to pass the embedding function explicitly rather than via the *Generate* class. In this case, we obtain the type: $C\ a \Rightarrow (g \to a) \to a$ if g is the set of generators. In modern versions of Haskell, we can even parameterize over the C class, and write:

```
newtype Free c g = Free (forall a ∘ c a ⇒ (g → a) → a)
```

Embedding a generator is then done like so:

```
embed :: g → Free c g
embed g = Free (λgenerate → generate g)
```

Unfortunately the *Free c* type is not automatically an instance of *c*: we have to implement those manually. Let us see how this plays out for monoid:

```
instance Monoid (Free Monoid g) where
  unit = Free (\_ → unit)
  Free f `op` Free g = Free (λx → f x `op` g x)
```

We can also check the monoid laws for the free monoid. For example, here is the proof that the right identity law holds:

```
    Free f `op` unit
== {- def. -}
    Free f `op` Free (\_ → unit)
== {- def. -}
    Free (λx → f x `op` unit)
== {- law of the underlying monoid -}
    Free (λx → f x)
== {- eta-reduction -}
    Free f
```

Exercise 4.10. Prove group laws for *Free AdditiveGroup*.

But we can also recover the whole structure which was used to build an element of this type, for example we could use lists (recall that they are isomorphic to free monoids):

```
extract :: Free Monoid g → [g]
extract (Free f) = f (λg → [g])
```

As an example, we can extract the value of the following:

```
example :: Free Monoid Int
example = embed 1 `op` embed 10 `op` unit `op` embed 11
  -- >>> extract example
  -- [1,10,11]
```

Exercise 4.11. Show that *Free Ring* () is in bijection with *FunExp*.

4.6 Computing Derivatives, reprise.

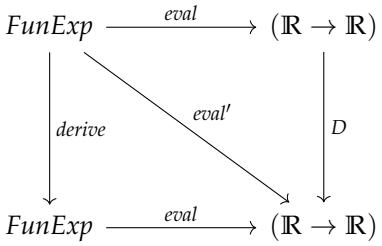
As discussed in Section 4.5.2, it can sometimes be economical to use the *OneVarExp a* ⇒ *a* representation rather than the *FunExp* data type. However, in Section 3.8 we argued that the rules for derivatives were naturally operating on a syntactic representation.

The question is: can we implement *derive* in the shallow embedding? As a reminder, the reason that the shallow embedding ($\mathbb{R} \to \mathbb{R}$) works is that the *eval* function is a *fold*: first evaluate the sub-expressions of e, then put the evaluations together without reference to the sub-expressions.

Let us now check whether the semantics of derivatives is compositional. This evaluation function for derivatives is given by composition as below:

$$
\begin{aligned}
&\textbf{type } \mathit{Func} = \mathbb{R} \to \mathbb{R} \\
&\mathit{eval}' :: \mathit{FunExp} \to \mathit{Func} \\
&\mathit{eval}' = \mathit{eval} \circ \mathit{derive}
\end{aligned}
$$

In a diagram:



Let us consider the *Exp* case:

$$
\begin{aligned}
&\mathit{eval}'\ (\mathit{Exp}\ e) && = \{\text{- def. } \mathit{eval}', \text{ function composition -}\} \\
&\mathit{eval}\ (\mathit{derive}\ (\mathit{Exp}\ e)) && = \{\text{- def. } \mathit{derive} \text{ for } \mathit{Exp} \text{ -}\} \\
&\mathit{eval}\ (\mathit{Exp}\ e :*: \mathit{derive}\ e) && = \{\text{- def. } \mathit{eval} \text{ for } :*: \text{ -}\} \\
&\mathit{eval}\ (\mathit{Exp}\ e) * \mathit{eval}\ (\mathit{derive}\ e) && = \{\text{- def. } \mathit{eval} \text{ for } \mathit{Exp} \text{ -}\} \\
&\mathit{exp}\ (\mathit{eval}\ e) * \mathit{eval}\ (\mathit{derive}\ e) && = \{\text{- def. } \mathit{eval}' \text{ -}\} \\
&\mathit{exp}\ (\mathit{eval}\ e) * \mathit{eval}'\ e && = \{\text{- let } f = \mathit{eval}\ e, f' = \mathit{eval}'\ e \text{ -}\} \\
&\mathit{exp}\ f * f'
\end{aligned}
$$

Thus, given *only* the derivative $f' = \mathit{eval}'\ e$, it is impossible to compute $\mathit{eval}'\ (\mathit{Exp}\ e)$. Another example of the problem is $\mathit{derive}\ (f :*: g)$ where the result involves not only $\mathit{derive}\ f$ and $\mathit{derive}\ g$, but also f and g. In general, the problem is that some of the rules for computing the derivative depend not only on the derivative of the subexpressions, but also on the subexpressions before taking the derivative.

Consequently, eval' is in fact non-compositional (just like *isPrime*). There is no way to implement $\mathit{eval}' :: \mathit{FunExp} \to \mathit{Func}$ as a fold *if Func is the target type*.

One way of expressing this is to say that in order to implement $\mathit{eval}' :: \mathit{FunExp} \to \mathit{Func}$ we need to also compute $\mathit{eval} :: \mathit{FunExp} \to \mathit{Func}$.

Thus we need to implement a pair of *eval*-functions $(\mathit{eval}, \mathit{eval}')$ together.

In practice, the solution is to extend the return type of eval' from one semantic value f of type $\mathit{Func} = \mathbb{R} \to \mathbb{R}$ to two such values $(f, f') :: (\mathit{Func}, \mathit{Func})$ where $f' = D\,f$. That is, we are using the "tupling transform": we are computing just

one function $evalD :: FunExp \to (Func, Func)$ returning a pair of f and $D\ f$ at once. (At this point, you are advised to look up and solve Exercise 1.8 in case you have not done so already.)

```
type FD a = (a → a, a → a)
evalD :: FunExp → FD Double
evalD  e        = (eval e, eval' e)
```

Is *evalD* compositional? We compute, for example:

```
evalD (Exp e)                      = {- specification of evalD -}
(eval (Exp e), eval' (Exp e))      = {- def. eval for Exp and reusing the computation above -}
(exp (eval e), exp (eval e) * eval' e) = {- introduce names for subexpressions -}
let f  = eval e
    f' = eval' e
in (exp f, exp f * f')             = {- def. evalD -}
let (f, f') = evalD e
in (exp f, exp f * f')
```

This semantics *is* compositional and the *Exp* case is as follows:

```
evalDExp :: FD Double → FD Double
evalDExp  (f, f') = (exp f, exp f * f')
```

In general, while $eval'$ is non-compositional, *evalD* is a more complex, but compositional, semantics. We can then get $eval'$ back as the second component of *evalD e*:

```
eval' :: FunExp → Func
eval' = snd ∘ evalD
```

Because all compositional functions can be expressed as a fold for a given algebra, we can now define a shallow embedding for the combined computation of functions and derivatives, using the numerical type classes.

```
instance Additive a ⇒ Additive (a → a, a → a) where    -- same as Num a ⇒ Num (FD a)
  (f, f') + (g, g') = (f + g, f'      + g'      )
instance Multiplicative a ⇒ Multiplicative (a → a, a → a) where    -- same as Num a ⇒ Num (FD a)
  (f, f') * (g, g') = (f * g, f' * g + f * g' )
```

Exercise 4.12. Implement the rest of the *Num* instance for *FD a*.

4.7 Summary

The following correspondence table summarises the discussion so far:

Computer Science	Mathematics
DSL	structure (category, algebra, ...)
deep embedding, abstract syntax	initial algebra
shallow embedding	any other algebra
semantics	homomorphism from the initial algebra

TODO: JP: I disagree with this table.
1. Initial algebras can be constructed without reference to data types (*Class a ⇒ a*)
2. Shallow embeddings can be initial algebras (*Class a ⇒ x → a* is an initial algebra of *Class* + unit generator.)
3. We saw that homomorphisms are not necessarily judicious semantics.
4. And isn't a DSL akin to a mathematical domain?

The underlying theory of this table is a fascinating topic but mostly out of scope for this book.

4.7.1 Structures and representations

One take home message of this chapter is that one should, as a rule, start with structural definitions first, and consider representation second. For example, in Section 4.6 we defined a *Ring* structure on pairs $(\mathbb{R}, \mathbb{R})$ by requiring the operations to be compatible with the interpretation $(f\ a, f'\ a)$. This requirement yields the following definition for multiplication for pairs:

$$(x, x') \circledast (y, y') = (x * y, x' * y + x * y')$$

But there is nothing in the "nature" of pairs of $\mathbb{R}$ that forces this definition upon us. We chose it, because of the intended interpretation.

This multiplication is obviously not the one we need for *complex numbers*. It would be instead:

$$(x, x') *. (y, y') = (x * y - x' * y', x * y' + x' * y)$$

Again, there is nothing in the nature of pairs that foists this operation on us. In particular, it is, strictly speaking, incorrect to say that a complex number *is* a pair of real numbers. The correct interpretation is that a complex number can be *represented* by a pair of real numbers, provided that we define the operations on these pairs in a suitable way.

The distinction between definition and representation is similar to the one between specification and implementation, and, in a certain sense, to the one between syntax and semantics. All these distinctions are frequently obscured, for example, because of prototyping (working with representations / implementations / concrete objects in order to find out what definition / specification / syntax is most adequate). They can also be context-dependent (one man's specification is another man's implementation). Insisting on the difference between definition and representation can also appear quite pedantic (as in the

discussion of complex numbers in Section 1.3). In general though, it is a good idea to be aware of these distinctions, even if they are suppressed for reasons of brevity or style. We will encounter this distinction again in Section 5.1.

4.8 Beyond Algebras: Co-algebra and the Stream calculus

In the coming chapters there will be quite a bit of material on infinite structures. These are often captured not by algebras, but by co-algebras. We will not build up a general theory of co-algebras in this book, but because we will be using infinite streams in the upcoming chapters we will expose right here their co-algebraic structure.

Streams as an abstract datatype. Consider the API for streams of values of type A represented by some abstract type X:

$$
\begin{aligned}
&\textbf{data}\ X\\
&\textbf{data}\ A\\
&\mathit{head}\ :: X \to A\\
&\mathit{tail}\ \ :: X \to X\\
&\mathit{cons}\ :: A \to X \to X\\
&\mathit{law1}\ \ s = \quad\ \ s == \mathit{cons}\ (\mathit{head}\ s)\ (\mathit{tail}\ s)\\
&\mathit{law2}\ a\ s = \quad\ \ s == \mathit{tail}\ (\mathit{cons}\ a\ s)\\
&\mathit{law3}\ a\ s = \quad\ \ a == \mathit{head}\ (\mathit{cons}\ a\ s)
\end{aligned}
$$

With this API we can use *head* to extract the first element of the stream, and *tail* to extract the rest as a new stream of type X. Using *head* and *tail* recursively we can extract an infinite list of values of type A:

$$
\begin{aligned}
&\mathit{toList} :: X \to [A]\\
&\mathit{toList}\ x = \mathit{head}\ x : \mathit{toList}\ (\mathit{tail}\ x)
\end{aligned}
$$

In the other direction, if we want to build a stream we only have one constructor: *cons* but no "base case". In Haskell, thanks to laziness, we can still define streams directly using *cons* and recursion As an example, we can construct a constant stream as follows:

$$
\begin{aligned}
&\mathit{constS} :: A \to X\\
&\mathit{constS}\ a = ca\\
&\quad \textbf{where}\ ca = \mathit{cons}\ a\ ca
\end{aligned}
$$

Instead of specifying a stream in terms of how to contruct it, we could describe it in terms of how to take it apart; by specifying its *head* and *tail*. In the constant stream example we would get something like:

$$head\ (constS\ a) = a$$
$$tail\ (constS\ a) = constS\ a$$

but this syntax is not supported in Haskell.

The last part of the API are a few laws we expect to hold. The first law simply states that if we first take a stream s apart into its head and its tail, we can get back to the original stream by *cons*ing them back together. The second and third are variant on this theme, and together the three laws specify how the three operations interact.

4.9 ????

TODO: JP: Some lost text:
As we saw, another way of phrasing that is to say that *eval* is a homomorphism, while $eval'$ is not.
These properties do not hold for $eval'$, but do hold for *evalD*.

TODO: JP: I don't understand the point of this section

TODO: JP: I have no idea what this tries to say.

In Section 3.7.3, we defined a *Ring* instance for functions with a *Ring* codomain. If we have an element of the domain of such a function, we can use it to obtain a homomorphism from functions to their codomains:

$$Ring\ a \Rightarrow x \to (x \to a) \to a$$

As suggested by the type, the homomorphism is just function application:

$$apply :: a \to (a \to b) \to b$$
$$apply\ a = \lambda f \to f\ a$$

Indeed, writing $h = apply\ c$ for some fixed c, we have

$$\begin{aligned} h\ (f + g) &= \{\text{- def. } apply \text{ -}\} \\ (f + g)\ c &= \{\text{- def. } + \text{ for functions -}\} \\ f\ c + g\ c &= \{\text{- def. } apply \text{ -}\} \\ h\ f + h\ g & \end{aligned}$$

etc.

Can we do something similar for *FD*? The elements of *FD a* are pairs of functions, so we can take

$$\textbf{type}\ Dup\ a = (a, a)$$
$$\textbf{type}\ FD\ a = (a \to a, a \to a)$$
$$applyFD :: a \to FD\ a \to Dup\ a$$
$$applyFD\quad c\quad ((f, f')) = (f\ c, f'\ c)$$

We now have the domain of the homomorphism $(FD\ a)$ and the homomorphism itself $(applyFD\ c)$, but we are missing the structure on the codomain,

which now consists of pairs $Dup\ a = (a, a)$. In fact, we can *compute* this structure from the homomorphism condition. For example (we skip the constructor *FD* for brevity):

$$
\begin{array}{ll}
h\ ((f,f') * (g,g')) & = \{\text{- def. } * \text{ for } FD\ a \text{ -}\} \\
h\ (f * g, f' * g + f * g') & = \{\text{- def. } h = applyFD\ c \text{ -}\} \\
((f * g)\ c, (f' * g + f * g')\ c) & = \{\text{- def. } * \text{ and } + \text{ for functions -}\} \\
(f\ c * g\ c, f'\ c * g\ c + f\ c * g'\ c) & = \{\text{- } \textbf{let } x = f\ c; y = g\ c; x' = f'\ c; y' = g'\ c \text{ -}\} \\
(x * y, x' * y + x * y') & = \{\text{- } \textbf{introduce } \circledast \textbf{ to make the ends meet -}\} \\
(x, x')\ \circledast\ (y, y') & = \{\text{- expand shorter names again -}\} \\
(f\ c, f'\ c)\ \circledast\ (g\ c, g'\ c) & = \{\text{- def. } h = applyFD\ c \text{ -}\} \\
h\ (f,f')\ \circledast\ h\ (g,g') &
\end{array}
$$

The identity will hold if we take

$$
\begin{array}{l}
(\circledast) :: Ring\ a \Rightarrow Dup\ a \to Dup\ a \to Dup\ a \\
(x, x') \circledast (y, y') = (x * y, x' * y + x * y')
\end{array}
$$

Thus, if we define a "multiplication" on pairs of values using $(\circledast)$, we get that $(applyFD\ c)$ is a *Multiplicative*-homomorphism for all c. We can now define an instance

```
instance Ring a ⇒ Multiplicative (Dup a) where
  (*) = (⊛)
    -- ... exercise
```

Exercise 4.13. Complete the instance declarations for $Dup\ \mathbb{R}$.

Note: because this computation goes through also for the other cases we can actually work with just pairs of values (at an implicit point $c :: a$) instead of pairs of functions. Thus we can define a variant of $FD\ a$ to be **type** $Dup\ a = (a, a)$

4.10 Exercises

Exercise 4.14. Homomorphisms. Consider the following definitions:

$$
\begin{aligned}
&\text{-- } h : A \to B \text{ is a homomorphism from } Op : A \to A \to A \text{ to } op : B \to B \to B \\
&H_2\ (h, Op, op) = \forall\, x.\ \forall\, y.\ h\ (Op\ x\ y) == op\ (h\ x)\ (h\ y) \\
&\text{-- } h : A \to B \text{ is a homomorphism from } F : A \to A \text{ to } f : B \to B \\
&H_1\ (h, F, f) \quad = \forall\, x.\ h\ (F\ x) == f\ (h\ x) \\
&\text{-- } h : A \to B \text{ is a homomorphism from } E : A \text{ to } e : B \\
&H_0\ (h, E, e) \quad = h\ E == e
\end{aligned}
$$

Prove or disprove the following claims:

- $H_2\ ((2*), (+), (+))$
- $H_2\ ((2*), (*), (*))$
- $H_2\ (exp, (+), (*))$
- $H_2\ (eval', (:+:), (+))$
- $H_1\ (\sqrt{\cdot}, (4*), (2*))$
- $\exists f.\ H_1\ (f, (2*) \circ (1+), (1+) \circ (2*))$

Exercise 4.15. Complete the instance declarations for *FunExp* (for *Num*, *Fractional*, and *Floating*).

Exercise 4.16. Complete the instance declarations for $Dup\ \mathbb{R}$, deriving them from the homomorphism requirement for *applyFD* (in Section 4.9).

Exercise 4.17. We now have three different ways of computing the derivative of a function such as $f\ x = sin\ x + exp\ (exp\ x)$ at a given point, say $x = \pi$.

1. Find $e :: FunExp$ such that $eval\ e = f$ and use $eval'$.
2. Find an expression of type $FD\ \mathbb{R}$ and use *apply*.
3. Apply f directly to the appropriate (x, x') and use *snd*.

Do you get the same result?

Exercise 4.18. In Exercise 1.16 we looked at the datatype $SR\ v$ for the language of semiring expressions. We will now use some of the concepts discussed in this chapter to expand on this language.

1. Define a type class *SemiRing* that corresponds to the semiring structure.

2. Define a *SemiRing* instance for the datatype *SR v* that you defined in exercise 1.3.

3. Find two other instances of the *SemiRing* class.

4. Specialise the evaluator that you defined in Exercise 1.16 to the two *SemiRing* instances defined above. Take three semiring expressions of type *SR String*, give the appropriate assignments and compute the results of evaluating, in each case, the three expressions.

Exercise 4.19. Show that arithmetic modulo n satisfies the semiring laws (it is even a ring). In more details: show that $\mathbb{Z}_n = \{0, 1, ..., n-1\}$ with $plus\ x\ y = (x + y)\%n$ and $times\ x\ y = (x * y)\%n$ forms a semiring.

With $h\ x = x\%n$, show that h is a homomorphism from $\mathbb{Z}$ to $\mathbb{Z}_n$.

Exercise 4.20. In Exercise 1.17, we looked a datatype for the language of lattice expressions. We will now use some of the concepts discussed in this chapter to expand on this language.

1. Define a type class *Lattice* that corresponds to the lattice structure.

2. Define a *Lattice* instance for the datatype for lattice expressions that you defined in 1.4.1.

3. Find two other instances of the *Lattice* class.

4. Specialise the evaluator you defined in exercise 1.4.2 to the two *Lattice* instances defined above. Take three lattice expressions, give the appropriate assignments and compute the results of evaluating, in each case, the three expressions.

Exercise 4.21. In Exercise 1.18, we looked a datatype for the language of abelian monoid expressions. We will now use some of the concepts discussed in this chapter to expand on this language.

1. Define a type class *AbMonoid* that corresponds to the abelian monoid structure.

2. Define an *AbMonoid* instance for the datatype for abelian monoid expressions that you defined in exercise 1.5.1.

3. Find one other instance of the *AbMonoid* class and give an example which is *not* an instance of *AbMonoid*.

4. Specialise the evaluator that you defined in exercise 1.5.2 to the *AbMonoid* instance defined above. Take three 'AbMonoidExp' expressions, give the appropriate assignments and compute the results of evaluating the three expressions.

Exercise 4.22. A *ring* is a set A together with two constants (or nullary operations), 0 and 1, one unary operation, *negate*, and two binary operations, $+$ and $*$, such that

1. 0 is the neutral element of $+$

   $$\forall\, x \in A.\ x + 0 = 0 + x = x$$

2. $+$ is associative

   $$\forall\, x, y, z \in A.\ x + (y + z) = (x + y) + z$$

3. *negate* inverts elements with respect to addition

   $$\forall\, x \in A.\ x + \mathit{negate}\ x = \mathit{negate}\ x + x = 0$$

4. $+$ is commutative

   $$\forall\, x, y \in A.\ x + y = y + x$$

5. 1 is the unit of $*$

   $$\forall\, x \in A.\ x * 1 = 1 * x = x$$

6. $*$ is associative

   $$\forall\, x, y, z \in A.\ x * (y * z) = (x * y) * z$$

7. $*$ distributes over $+$

   $$\forall\, x, y, z \in A.\ x * (y + z) = (x * y) + (x * z)$$
   $$\forall\, x, y, z \in A.\ (x + y) * z = (x * z) + (y * z)$$

Remarks:

- a. and b. say that $(A, 0, +)$ is a monoid
- a–c. say that $(A, 0, +, \mathit{negate})$ is a group
- a–d. say that $(A, 0, +, \mathit{negate})$ is a commutative group
- e. and f. say that $(A, 1, *)$ is a monoid

i Define a type class *Ring* that corresponds to the ring structure.

ii Define a datatype for the language of ring expressions (including variables) and define a *Ring* instance for it.

iii Find two other instances of the *Ring* class.

iv Define a general evaluator for *Ring* expressions on the basis of a given assignment function.

v Specialise the evaluator to the two *Ring* instances defined at point iii. Take three ring expressions, give the appropriate assignments and compute the results of evaluating, in each case, the three expressions.

Exercise 4.23. Recall the type of expressions of one variable from Section 3.1.1.

```
data FunExp = Const Rational       | Id
            | FunExp :+: FunExp    | Exp FunExp
            | FunExp :*: FunExp    | Sin FunExp
            | FunExp :/: FunExp    | Cos FunExp
              -- and so on
            deriving Show
```

and consider the function

$$f :: \mathbb{R} \to \mathbb{R}$$
$$f\ x = exp\ (sin\ x) + x$$

1. Find an expression e such that $eval\ e == f$ and show this using equational reasoning.

2. Implement a function *deriv2* such that, for any $f : Fractional\ a \Rightarrow a \to a$ constructed with the grammar of *FunExp* and any x in the domain of f, we have that $deriv2\ f\ x$ computes the second derivative of f at x. Use the function $derive :: FunExp \to FunExp$ from the lectures ($eval\ (derive\ e)$ is the derivative of $eval\ e$). What instance declarations do you need?

   The type of $deriv2\ f$ should be $Fractional\ a \Rightarrow a \to a$.

Exercise 4.24. Based on the lecture notes, complete all the instance and datatype declarations and definitions in the files FunNumInst.lhs, FunExp.lhs, Derive.lhs, EvalD.lhs, and ShallowD.lhs.

Exercise 4.25. Write a function

$$simplify :: FunExp \to FunExp$$

to simplify the expression resulted from *derive*. For example, the following tests should work:

$$
\begin{aligned}
&\mathit{simplify}\ (\mathit{Const}\ 0 :*: \mathit{Exp}\ \mathit{Id}) &&== \mathit{Const}\ 0\\
&\mathit{simplify}\ (\mathit{Const}\ 0 :+: \mathit{Exp}\ \mathit{Id}) &&== \mathit{Exp}\ \mathit{Id}\\
&\mathit{simplify}\ (\mathit{Const}\ 2 :*: \mathit{Const}\ 1) &&== \mathit{Const}\ 2\\
&\mathit{simplify}\ (\mathit{derive}\ (\mathit{Id} :*: \mathit{Id})) &&== \mathit{Const}\ 2 :*: \mathit{Id}
\end{aligned}
$$

As a motivating example, note that $\mathit{derive}\ (\mathit{Id} :*: \mathit{Id})$ evalutes to $(\mathit{Const}\ 1.0 :*: \mathit{Id}) :+: (\mathit{Id} :*: \mathit{Const}\ 1.0)$ without *simplify*, and that the second derivative looks even worse.

Chapter 5

Polynomials and Power Series

```
{-# LANGUAGE GADTs #-}
{-# LANGUAGE TypeSynonymInstances #-}
{-# LANGUAGE RebindableSyntax #-}
{-# LANGUAGE TypeApplications #-}
module DSLsofMath.W05 where
import Prelude hiding (Num (..), (/), (^))
import DSLsofMath.FunNumInst
import DSLsofMath.Algebra
```

5.1 Polynomials

From Adams and Essex [2010], page 39:

> A **polynomial** is a function P whose value at x is
>
> $$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$
>
> where a_n, a_{n-1}, ..., a_1, and a_0, called the **coefficients** of the polynomial [*sic*], are constants and, if $n > 0$, then $a_n \neq 0$. The number n, the degree of the highest power of x in the polynomial, is called the **degree** of the polynomial. (The degree of the zero polynomial is not defined.)

This definition raises a number of questions, for example "what is the zero polynomial?".

The types of the elements involved in the definition appear to be

$n \in \mathbb{N}$, $P : \mathbb{R} \to \mathbb{R}$, $x \in \mathbb{R}$, $a_0, \ldots, a_n \in \mathbb{R}$ with $a_n \neq 0$ if $n > 0$

The phrasing should be "whose value at *any* x is". The remark that the a_i are constants is probably meant to indicate that they do not depend on x, otherwise every function would be a polynomial. The zero polynomial is, according to this definition, the *const* 0 function. Thus, what is meant is

> A **polynomial** is a function $P : \mathbb{R} \to \mathbb{R}$ which either is the constant zero, or there exist $a_0, \ldots, a_n \in \mathbb{R}$ with $a_n \neq 0$ such that, for any $x \in \mathbb{R}$
>
> $$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Given the coefficients a_i we can evaluate P at any given x. Ignoring the condition on coefficients for now, we can assume that the coefficients are given as a list

$$as = [a_0, a_1, ..., a_n]$$

(we prefer counting up), then the evaluation function is written

```
evalL :: [ℝ] → ℝ → ℝ
evalL  []      x  = 0
evalL  (a : as) x  = a + x * evalL as x
```

Note that we can read the type as $evalL :: [\mathbb{R}] \to (\mathbb{R} \to \mathbb{R})$ and thus identify $[\mathbb{R}]$ as the type for the (abstract) syntax (for polynomials) and $(\mathbb{R} \to \mathbb{R})$ as the type of the semantics (for polynomial functions).

Exercise 5.1. Show that this evaluation function gives the same result as the formula above.

Using the *Ring* instance for functions we can rewrite *eval* into a one-argument function (returning a polynomial function):

```
evalL :: Ring a ⇒ [a] → (a → a)
evalL []       = const 0
evalL (a : as) = const a + id * evalL as
```

As an example, the polynomial which is usually written just x is represented by the list $[0, 1]$ and the polynomial function $\lambda x \to x\hat{}2 - 1$ is represented by the list $[-1, 0, 1]$.

It is worth noting that the definition of what we call a "polynomial function" is semantic, not syntactic. A syntactic defintion would talk about the form of the expression (a sum of coefficients times natural powers of x). This semantic definition only requires that the function *P behaves like* such a sum. (Has the same

value for all x.) This may seem pedantic, but here is an interesting example of a family of functions which syntactically looks very trigonometric:

TODO: JP: what is the subtext here?

$$T_n(x) = \cos(n * \arccos(x)) \ .$$

It can be shown that T_n is a polynomial function of degree n. (Exercise 5.8 guides you to a proof. At this point you could just compute T_0, T_1, and T_2 by hand to get a feeling for how it works.)

Not every list of coefficients is valid according to the definition. In particular, the empty list is not a valid list of coefficients, so we have a conceptual, if not empirical, type error in our evaluator.

The valid lists are those *finite* lists in the set

$$\{\,[0]\,\} \cup \{\,(a : as) \mid last\ (a : as) \neq 0\,\}$$

The fact that the element should be non-zero is easy to express as a Haskell expression ($last\ (a : as) \neq 0$), but not so easy to express in the *types*. But we can easily express the condition that the list should not be empty:

```
data Poly a = Single a | Cons a (Poly a)
            deriving (Eq, Ord)
```

Note that if we drop the requirement of what constitutes a "valid" list of coefficients we can use $[a]$ instead of *Poly a*. Basically, we then use $[\,]$ as the syntax for the "zero polynomial" and $(c : cs)$ for all non-zero polynomials.

TODO: JP: But it appears that this "validity" is there only to define the degree. Terrible.

The relationship between *Poly a* and $[a]$ is given by the following functions:

```
toList :: Poly a →      [a]
toList (Single a)   = a : []
toList (Cons a as) = a : toList as

fromList :: Ring a ⇒ [a] → Poly a
fromList (a : [])       = Single a
fromList (a0 : a1 : as) = Cons a0 (fromList (a1 : as))
fromList []             = Single 0    -- to complete the pattern match

instance Show a ⇒ Show (Poly a) where
  show = show ∘ toList
```

Since we only use the arithmetical operations, we can generalise our evaluator to an arbitrary *Ring* type.

```
evalPoly :: Ring a ⇒ Poly a → (a → a)
evalPoly (Single a)  x = a
evalPoly (Cons a as) x = a + x * evalPoly as x
```

Since we have *Ring a*, there is a *Ring* structure on $a \to a$, and *evalPoly* looks like a homomorphism. Question: is there a *Ring* structure on *Poly a*, such that *evalPoly* is a homomorphism?

For example, the homomorphism condition gives for $(+)$

$$evalPoly\ as + evalPoly\ bs = evalPoly\ (as + bs)$$

Both sides are functions, they are equal iff. they are equal for every argument. For an arbitrary x

$$\begin{aligned}
&(evalPoly\ as + evalPoly\ bs)\ x = evalPoly\ (as + bs)\ x \\
\Leftrightarrow\ &\{\text{- } + \text{ on functions is defined point-wise -}\} \\
&evalPoly\ as\ x + evalPoly\ bs\ x = evalPoly\ (as + bs)\ x
\end{aligned}$$

To proceed further, we need to consider the various cases in the definition of *evalPoly*. We give here the computation for the last case (where *as* has at least one *Cons*), using the traditional list notation $(:)$ for brevity.

$$evalPoly\ (a : as)\ x + evalPoly\ (b : bs)\ x = evalPoly\ ((a : as) + (b : bs))\ x$$

For the left-hand side, we have:

$$\begin{aligned}
&evalPoly\ (a : as)\ x + evalPoly\ (b : bs)\ x && = \{\text{- def. } evalPoly \text{ -}\} \\
&(a + x * evalPoly\ as\ x) + (b + x * eval\ bs\ x) && = \{\text{- properties of } +, \text{ valid in any ring -}\} \\
&(a + b) + x * (evalPoly\ as\ x + evalPoly\ bs\ x) && = \{\text{- homomorphism condition -}\} \\
&(a + b) + x * (evalPoly\ (as + bs)\ x) && = \{\text{- def. } evalPoly \text{ -}\} \\
&evalPoly\ ((a + b) : (as + bs))\ x
\end{aligned}$$

The homomorphism condition will hold for every x if we define

$$(a : as) + (b : bs) = (a + b) : (as + bs)$$

This definition looks natural (we could probably have guessed it early on) but it is still interesting to see that we can derive it as the form that it has to take for the proof to go through.

We leave the derivation of the other cases and operations as an exercise. Here, we just give the corresponding definitions.

```
instance Additive a ⇒ Additive (Poly a) where
  (+) = polyAdd
  zero = Single zero
instance Ring a ⇒ Multiplicative (Poly a) where
  (∗) = polyMul
  one = Single one
```

```
instance AddGroup a ⇒ AddGroup (Poly a) where
  negate = polyNeg
polyAdd :: Additive a ⇒ Poly a → Poly a → Poly a
polyAdd (Single a)  (Single b)  = Single (a + b)
polyAdd (Single a)  (Cons b bs) = Cons (a + b) bs
polyAdd (Cons a as) (Single b)  = Cons (a + b) as
polyAdd (Cons a as) (Cons b bs) = Cons (a + b) (polyAdd as bs)
polyMul :: Ring a ⇒ Poly a → Poly a → Poly a
polyMul (Single a)  (Single b)  = Single (a * b)
polyMul (Single a)  (Cons b bs) = Cons (a * b) (polyMul (Single a) bs)
polyMul (Cons a as) (Single b)  = Cons (a * b) (polyMul as (Single b))
polyMul (Cons a as) (Cons b bs) = Cons (a * b) (polyAdd (polyMul as (Cons b bs))
                                                        (polyMul (Single a) bs))
polyNeg :: AddGroup a ⇒ Poly a → Poly a
polyNeg = mapPoly negate
mapPoly :: (a → b) → (Poly a → Poly b)
mapPoly f (Single a)  = Single (f a)
mapPoly f (Cons a as) = Cons (f a) (mapPoly f as)
```

Therefore, we *can* define a *Ring* structure on *Poly a*, and we have arrived at the canonical definition of polynomials, as found in any algebra book (see, for example, Rotman [2006] for a very readable text):

> Given a commutative ring A, the commutative ring given by the set *Poly A* together with the operations defined above is the ring of **polynomials** with coefficients in A.

The family of functions *evalPoly as* for every possible *as* are known as *polynomial functions*.

Caveat: The canonical representation of polynomials in algebra does not use finite lists, but the equivalent

$$Poly'\ A = \{\, a : \mathbb{N} \to A \mid \text{\{- } a \text{ has only a finite number of non-zero values -\}} \ \}$$

Exercise 5.2. What are the ring operations on $Poly'\ A$? Hint: they are different from the operation induced by the ring operations on A.

For example, here is addition:

$$a + b = c \Leftrightarrow a\ n + b\ n = c\ n \quad \text{-- } \forall n : \mathbb{N}$$

Remark: Using functions from $\mathbb{N}$ in the definition has certain technical advantages over using finite lists. For example, consider adding $[a_0, a_1, ..., a_n]$

and $[b_0, b_1, ..., b_m]$, where $n > m$. Then, we obtain a polynomial of degree n: $[c_0, c_1, ..., cn]$. The formula for the c_i must now be given via a case distinction:

$$c_i = \textbf{if } i > m \textbf{ then } a_i \textbf{ else } a_i + b_i$$

since b_i does not exist for values greater than m.

Compare this with the above formula for functions, where no case distinction necessary. The advantage is even clearer in the case of multiplication.

Observations:

1. If one considers arbitrary rings, polynomials are not isomorphic (in one-to-one correspondence) to polynomial functions. For any finite ring A, there is a finite number of functions $A \to A$, but there is a countable number of polynomials. That means that the same polynomial function on A will be the evaluation of many different polynomials.

   For example, consider the ring $\mathbb{Z}_2$ ($\{0, 1\}$ with addition and multiplication modulo 2). In this ring, we have that $p\ x = x + x\hat{}2$ is actually a constant function. The only two input values to p are 0 and 1 and we can easily check that $p\ 0 = 0$ and also $p\ 1 = (1 + 1\hat{}2)\%2 = 2\%2 = 0$. Thus

   $$\mathit{evalPoly}\ [0, 1, 1] = p = \mathit{const}\ 0 = \mathit{evalPoly}\ [0]\{\text{- in } \mathbb{Z}_2 \to \mathbb{Z}_2 \text{ -}\}$$

   but

   $$[0, 1, 1] \neq [0]\{\text{- in } \mathit{Poly}\ \mathbb{Z}_2 \text{ -}\}$$

   Therefore, it is not generally a good idea to conflate polynomials and polynomial functions.

2. Following the DSL terminology, we can say that the polynomial functions are the semantics of the language of polynomials. We started with polynomial functions, we wrote the evaluation function and realised that we have the makings of a homomorphism. That suggested that we could create an adequate language for polynomial functions. Indeed, this turns out to be the case; in so doing, we have recreated an important mathematical achievement: the algebraic definition of polynomials.

   Let

   $$x :: \mathit{Ring}\ a \Rightarrow \mathit{Poly}\ a$$
   $$x = \mathit{Cons}\ 0\ (\mathit{Single}\ 1)$$

   Then (again, using the list notation for brevity) for any polynomial $as = [a_0, a_1, ..., a_n]$ we have

   $$as = a_0 + a_1 * x + a_2 * x\hat{}2 + ... + a_n * x\hat{}n$$

Exercise 5.3. Prove the above equality.

This equality justifies the standard notation

$$as = \sum_{i=0}^{n} a_i * x^i$$

5.2 Aside: division and the degree of the zero polynomial

Recall the fundamental property of division that we learned in high school:

For all natural numbers a, b, with $b \neq 0$, there exist *unique* integers q and r, such that

$$a = b * q + r, with\ r < b$$

When $r = 0$, a is divisible by b. Questions of divisibility are essential in number theory and its applications (including cryptography).

A similar theorem holds for polynomials (see, for example, Adams and Essex [2010] page 40):

For all polynomials as, bs, with $bs \neq Single\ 0$, there exist *unique* polynomials qs and rs, such that

$$as = bs * qs + rs, with\ degree\ rs < degree\ bs$$

The condition $r < b$ is replaced by $degree\ rs < degree\ bs$. However, we now have a problem. Every polynomial is divisible by any non-zero constant polynomial, resulting in a zero polynomial remainder. But the degree of a constant polynomial is zero. If the degree of the zero polynomial were a natural number, it would have to be smaller than zero. For this reason, it is either considered undefined (as in Adams and Essex [2010]), or it is defined as $-\infty$. The next section examines this question from a different point of view, that of homomorphisms.

5.3 Polynomial degree as a homomorphism

It is often the case that a certain function is *almost* a homomorphism and the domain or range structure is *almost* a monoid. In Section 4.6, we have seen "tupling" as one way to fix such a problem and here we will introduce another way.

TODO: JP: This is referring to the evaluator of derivatives. But probably is should be moved after this, in the derivatives chapter.

The *degree* of a polynomial is a good candidate for being a homomorphism: if we multiply two polynomials we can normally add their degrees. If we try to

check that $degree :: Poly\ a \to \mathbb{N}$ is the function underlying a monoid morphism we need to decide on the monoid structure to use for the source and for the target, and we need to check the homomorphism laws. We can use $unit = Single\ 1$ and $op = polyMul$ for the source monoid and we can try to use $unit = 0$ and $op = (+)$ for the target monoid. Then we need to check that

$$degree\ (Single\ 1) = 0$$
$$\forall\, x, y.\ degree\ (x\ `op`\ y) = degree\ x + degree\ y$$

The first law is no problem and for most polynomials the second law is also straighforward to prove (exercise: prove it). But we run into trouble with one special case: the zero polynomial.

Looking back at the definition from Adams and Essex [2010], page 55 it says that the degree of the zero polynomial is not defined. Let's see why that is the case and how we might "fix" it. Assume that there exists a natural number z such that $degree\ 0 = z$. Assume additionally a polynomial p with $degree\ p = n$. Then we get

$$\begin{aligned}
&z && = \{\text{- assumption -}\}\\
&degree\ 0 && = \{\text{- simple calculation -}\}\\
&degree\ (0 * p) && = \{\text{- homomorphism condition -}\}\\
&degree\ 0 + degree\ p && = \{\text{- assumption -}\}\\
&z + n
\end{aligned}$$

Thus we need to find a z such that $z = z + n$ for all natural numbers n! At this stage we could either give up, or think out of the box. Intuitively we could try to use $z = -Infinity$, which would seem to satisfy the law but is not a natural number (not even an integer). More formally what we need to do is to extend the monoid $(\mathbb{N}, 0, +)$ by one more element. In Haskell we can do that using the *Maybe* type constructor:

```
class Monoid a where
  unit :: a
  op   :: a → a → a
instance Monoid a ⇒ Monoid (Maybe a) where
  unit = Just unit
  op   = opMaybe
opMaybe Nothing  m        = Nothing  -- −Inf + m = −Inf
opMaybe m        Nothing  = Nothing  -- m + (−Inf) = −Inf
opMaybe (Just m1) (Just m2) = Just (op m1 m2)
```

Thus, to sum up, *degree* is a monoid homomorphism from $(Poly\ \ a, 1, *)$ to $(Maybe\ \mathbb{N}, Just\ 0, opMaybe)$.

Exercise 5.4. Check all the Monoid and homomorphism properties.

5.4 Power Series

Consider the following "pseudo proof":

TODO: JP: Proposition?

Theorem 1 (Fake theorem)**.** *Let $m, n \in \mathbb{N}$ and let cs and as be any polynomials of degree $m + n$ and n, respectively, and with $a_0 \neq 0$. Then cs is divisible by as.*

TODO: JP: Proof attempt?

Proof. We need to find $bs = [b_0, ..., b_m]$ such that $cs = as * bs$. From the multiplication of polynomials, we know that

$$c_k = \sum_{i=0}^{k} a_i * b_{k-i}$$

Therefore:

$$c_0 = a_0 * b_0$$

Since c_0 and a_0 are known, computing $b_0 = c_0 \ / \ a_0$ is trivial. Next

$$c_1 = a_0 * b_1 + a_1 * b_0$$

Again, we are given c_1, a_0 and a_1, and we have just computed b_0, therefore we can obtain b_1. Similarly

$$c_2 = a_0 * b_2 + a_1 * b_1 + a_2 * b_0$$

from which we obtain, exactly as before, the value of b_2.

It is clear that this process can be continued, yielding at every step a value for a coefficient of *bs*, and thus we have obtained *bs* satisfying $cs = as * bs$.

□

The problem with this "proof" is in the statement "it is clear that this process can be continued". In fact, it is rather clear that it cannot (for polynomials)! Indeed, *bs* only has $m + 1$ coefficients, therefore for all remaining n equations of the form $c_k = \sum_{i=0}^{k} a_i * b_{k-i}$, the values of b_k have to be zero. But in general this will not satisfy the equations.

However, we can now see that, if we were able to continue for ever, we would be able to divide *cs* by *as* exactly. The only obstacle is the "finite" nature of our lists of coefficients.

Power series are obtained from polynomials by removing in *Poly′* the restriction that there should be a *finite* number of non-zero coefficients; or, in, the case of *Poly*, by going from lists to streams.

$$PowerSeries' \ a = \{f : \mathbb{N} \to a\}$$

type *PowerSeries a* = *Poly a* -- finite and infinite non-empty lists

The operations are still defined as before. If we consider only infinite lists, then only the equations which do not contain the patterns for singleton lists will apply.

Power series are usually denoted

$$\sum_{n=0}^{\infty} a_n * x^n$$

the interpretation of x being the same as before. The simplest operation, addition, can be illustrated as follows:

$$\begin{array}{lcl}
\sum_{i=0}^{\infty} a_i * x^i & \cong & [a_0, \quad a_1, \quad \ldots] \\
\sum_{i=0}^{\infty} b_i * x^i & \cong & [b_0, \quad b_1, \quad \ldots] \\
\sum_{i=0}^{\infty} (a_i + b_i) * x^i & \cong & [a_0 + b_0, \quad a_1 + b_1, \quad \ldots]
\end{array}$$

The evaluation of a power series represented by $a : \mathbb{N} \to A$ is defined, in case the necessary operations make sense on A, as a function

$$\begin{array}{l}
\mathit{eval}\ a : A \to A \\
\mathit{eval}\ a\ x = \mathit{lim}\ s\ \textbf{where}\ s\ n = \sum_{i=0}^{n} a_i * x^i
\end{array}$$

Note that *eval a* is, in general, a partial function (the limit might not exist).

We will consider, as is usual, only the case in which $A = \mathbb{R}$ or $A = \mathbb{C}$.

The term *formal* refers to the independence of the definition of power series from the ideas of convergence and evaluation. In particular, two power series represented by a and b, respectively, are equal only if $a = b$ (as infinite series of numbers). If $a \neq b$, then the power series are different, even if $\mathit{eval}\ a = \mathit{eval}\ b$.

TODO: JP: What formal? Formal power series have never been introduced.

Since we cannot in general compute limits, we can use an "approximative" *eval*, by evaluating the polynomial resulting from an initial segment of the power series.

```
eval :: Ring a ⇒ Integer → PowerSeries a → (a → a)
eval n as x = evalPoly (takePoly n as) x

takePoly :: Integer → PowerSeries a → Poly a
takePoly n (Single a)  = Single a
takePoly n (Cons a as) = if n ⩽ 1
                         then Single a
                         else Cons a (takePoly (n − 1) as)
```

Note that *eval n* is not a homomorphism: for example:

$$
\begin{array}{ll}
\mathit{eval}\ 2\ (x * x)\ 1 & = \\
\mathit{evalPoly}\ (\mathit{takePoly}\ 2\ [0,0,1])\ 1 & = \\
\mathit{evalPoly}\ [0,0]\ 1 & = \\
0 &
\end{array}
$$

but

$$
\begin{array}{ll}
(\mathit{eval}\ 2\ x\ 1) & = \\
\mathit{evalPoly}\ (\mathit{takePoly}\ 2\ [0,1])\ 1 & = \\
\mathit{evalPoly}\ [0,1]\ 1 & = \\
1 &
\end{array}
$$

and thus $\mathit{eval}\ 2\ (x * x)\ 1 = 0 \neq 1 = 1 * 1 = (\mathit{eval}\ 2\ x\ 1) * (\mathit{eval}\ 2\ x\ 1)$.

5.5 Operations on power series

Power series have a richer structure than polynomials. For example, we also have division (this is similar to the move from $\mathbb{Z}$ to $\mathbb{Q}$). We start with a special case: trying to compute $p = \frac{1}{1-x}$ as a power series. The specification of $a\ /\ b = c$ is $a = c * b$, thus in our case we need to find a p such that $1 = (1 - x) * p$. For polynomials there is no solution to this equation. One way to see that is by using the homomorphism *degree*: the degree of the left hand side is 0 and the degree of the RHS is $1 + \mathit{degree}\ p \neq 0$. But there is still hope if we move to formal power series.

TODO: JP: Wasn't this explained just above in the "fake proof"?

Remember that p is then represented by a stream of coefficients $[p_0, p_1, ...]$. We make a table of the coefficients of the RHS $= (1 - x) * p = p - x * p$ and of the LHS $= 1$ (seen as a power series).

$$
\begin{array}{lll}
p & == & [p_0, p_1, \quad p_2, \quad ... \\
x * p & == & [0, \ p_0, \quad p_1, \quad ... \\
p - x * p & == & [p_0, p_1 - p_0, p_2 - p_1, \ ... \\
1 & == & [1, \ 0, \quad 0, \quad ...
\end{array}
$$

Thus, to make the last two lines equal, we are looking for coefficients satisfying $p_0 = 1, p_1 - p_0 = 0, p_2 - p_1 = 0, \ldots$. The solution is unique: $1 = p_0 = p_1 = p_2 = \ldots$ but only exists for streams (infinite lists) of coefficients. In the common math notation we have just computed

$$\frac{1}{1-x} = \sum_{i=0}^{\infty} x^i$$

Note that this equation holds when we interpret both sides as formal power series, but not necessarily if we try to evaluate the expressions for a particular x. That works for $|x| < 1$ but not for $x = 2$, for example.

For a more general case of power series division, consider p / q with $p = a : as$, $q = b : bs$, and assume that $a * b \neq 0$. Then we want to find, for any given $(a : as)$ and $(b : bs)$, the series $(c : cs)$ satisfying

$$
\begin{aligned}
&(a : as) \;/\; (b : bs) = (c : cs) && \Leftrightarrow \text{\{- def. of division -\}}\\
&(a : as) = (c : cs) * (b : bs) && \Leftrightarrow \text{\{- def. of } * \text{ for } Cons \text{ -\}}\\
&(a : as) = (c * b) : (cs * (b : bs) + [c] * bs) && \Leftrightarrow \text{\{- equality on components, def. of division -\}}\\
&c \;= a \;/\; b \qquad \text{\{- and -\}}\\
&as = cs * (b : bs) + [c] * bs && \Leftrightarrow \text{\{- arithmetics -\}}\\
&c \;= a \;/\; b \qquad \text{\{- and -\}}\\
&cs = (as - [c] * bs) \;/\; (b : bs)
\end{aligned}
$$

This leads to the implementation:

```
instance (Eq a, Field a) ⇒ MulGroup (PowerSeries a) where
  (/) = divPS
divPS :: (Eq a, Field a) ⇒ PowerSeries a → PowerSeries a → PowerSeries a
divPS as          (Single b)  = as * Single (1 / b)
divPS (Single 0)  (Cons b bs) = Single 0
divPS (Single a)  (Cons b bs) = divPS (Cons a (Single 0)) (Cons b bs)
divPS (Cons a as) (Cons b bs) = Cons c (divPS (as − (Single c) * bs) (Cons b bs))
                                where c = a / b
```

The first two equations allow us to also use division on polynomials, but the result will, in general, be a power series, not a polynomial. The first one should be self-explanatory. The second one extends a constant polynomial, in a process similar to that of long division.

For example:

```
ps0, ps1, ps2 :: (Eq a, Field a) ⇒ PowerSeries a
ps0 = 1 / (1 − x)
ps1 = 1 / (1 − x)^2
ps2 = (x^2 − 2 * x + 1) / (x − 1)
```

Every *ps* is the result of a division of polynomials: the first two return power series, the third is a polynomial (almost: it has a trailing 0.0).

```
example0 :: (Eq a, Field a) ⇒ PowerSeries a
example0  = takePoly 10 ps0
example01 :: (Eq a, Field a) ⇒ PowerSeries a
example01 = takePoly 10 (ps0 * (1 − x))
```

We can get a feeling for the definition by computing ps_0 "by hand". We let $p = [1]$ and $q = [1, -1]$ and seek $r = p$ / q.

$$
\begin{array}{ll}
\mathit{divPS}\ p\ q & = \{\text{- def. of } p \text{ and } q \text{ -}\} \\
\mathit{divPS}\ [1]\quad (1:[-1]) & = \{\text{- 3rd case of } \mathit{divPS} \text{ -}\} \\
\mathit{divPS}\ (1:[0])\ (1:[-1]) & = \{\text{- 4th case of } \mathit{divPS} \text{ -}\} \\
(1\ /\ 1) : \mathit{divPS}\ ([0]-[1]*[-1])\ (1:[-1]) & = \{\text{- simplification, def. of } (*) \text{ -}\} \\
1 : \mathit{divPS}\ ([0]-[-1])\ (1:[-1]) & = \{\text{- def. of } (-) \text{ -}\} \\
1 : \mathit{divPS}\ [1]\ (1:[-1]) & = \{\text{- def. of } p \text{ and } q \text{ -}\} \\
1 : \mathit{divPS}\ p\ q &
\end{array}
$$

Thus, the answer r starts with 1 and continues with r! In other words, we have that $1\ /\ [1,-1] = [1,1\ldots]$ as infinite lists of coefficients and $\frac{1}{1-x} = \sum_{i=0}^{\infty} x^i$ in the more traditional mathematical notation.

5.6 Formal derivative

Considering the analogy between power series and polynomial functions (via polynomials), we can arrive at a formal derivative for power series through the folowing computation:

$$
\begin{aligned}
\left(\sum_{n=0}^{\infty} a_n * x^n\right)' &= \sum_{n=0}^{\infty} (a_n * x^n)' = \sum_{n=0}^{\infty} a_n * (x^n)' = \sum_{n=0}^{\infty} a_n * (n * x^{n-1}) \\
&= \sum_{n=0}^{\infty} (n * a_n) * x^{n-1} = \sum_{n=1}^{\infty} (n * a_n) * x^{n-1} = \sum_{m=0}^{\infty} ((m+1) * a_{m+1}) * x^m
\end{aligned}
\tag{5.1}
$$

Thus the mth coefficient of the derivative is $(m+1) * a_{m+1}$.

We can implement this, for example, as

$$
\begin{array}{l}
\mathit{deriv}\ (\mathit{Single}\ a) \quad = \mathit{Single}\ 0 \\
\mathit{deriv}\ (\mathit{Cons}\ a\ as) = \mathit{deriv}'\ as\ 1 \\
\quad \textbf{where}\ \mathit{deriv}'\ (\mathit{Single}\ a) \quad n = \mathit{Single}\ (n * a) \\
\qquad\qquad \mathit{deriv}'\ (\mathit{Cons}\ a\ as)\ n = \mathit{Cons} \quad (n * a)\ (\mathit{deriv}'\ as\ (n+1))
\end{array}
$$

Side note: we cannot in general implement a decidable (Boolean) equality test for *PowerSeries*. For example, we know that $\mathit{deriv}\ ps_0$ equals ps_1 but we cannot compute *True* in finite time by comparing the coefficients of the two power series.

$$
\begin{array}{l}
\mathit{checkDeriv} :: \mathit{Integer} \to \mathit{Bool} \\
\mathit{checkDeriv}\ n = \mathit{takePoly}\ n\ (\mathit{deriv}\ ps_0) == \mathit{takePoly}\ n\ (ps_1 :: \mathit{Poly}\ \mathit{Rational})
\end{array}
$$

Recommended reading: the Functional pearl: "Power series, power serious" McIlroy [1999].

5.7 Helpers

```
instance Functor Poly where
  fmap = mapPoly
po1 :: (Eq a, Field a) ⇒ Poly a
po1 = 1 + x^2 − 3 ∗ x^4
instance Ring a ⇒ Monoid′ (Poly a) where
  unit = Single 1
  op = (∗)
instance Monoid′ Integer where
  unit = 0
  op = (+)
type ℕ = Integer
degree :: (Eq a, Ring a) ⇒ Poly a → Maybe ℕ
degree (Single 0) = Nothing
degree (Single x) = Just 0
degree (Cons x xs) = maxd (degree (Single x)) (fmap (1+) (degree xs))
  where maxd x          Nothing = x
        maxd Nothing (Just d) = Just d
        maxd (Just a) (Just b) = Just (max a b)
checkDegree0 = degree (unit :: Poly Integer) == unit
checkDegreeM :: Poly Integer → Poly Integer → Bool
checkDegreeM p q = degree (p ∗ q) == op (degree p) (degree q)
```

5.8 Exercises

The first few exercises are about filling in the gaps in the chapter above.

Exercise 5.5. Polynomial multiplication. To get a feeling for the definition it can be useful to take it step by step, starting with some easy cases.

$$\begin{aligned} &mulP\ [\,]\ p = && \text{-- TODO} \\ &mulP\ p\ [\,] = && \text{-- TODO} \end{aligned}$$

$$\begin{aligned} &mulP\ [a]\ p = && \text{-- TODO} \\ &mulP\ p\ [b] = && \text{-- TODO} \end{aligned}$$

$$\begin{aligned} &mulP\ (0 : as)\ p = && \text{-- TODO} \\ &mulP\ p\ (0 : bs) = && \text{-- TODO} \end{aligned}$$

Finally we reach the main case

$$mulP\ (a : as)\ p@(b : bs) = \quad \text{-- TODO}$$

Exercise 5.6. Show (by induction) that the evaluation function *evalL* gives the same result as the formula

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

.

Exercise 5.7. Prove that, with the definition of $x = [0, 1]$ we really have

$$as = a_0 + a_1 * x + a_2 * x\hat{}2 + ... + a_n * x\hat{}n$$

Exercise 5.8. Chebyshev polynomials. Let $T_n(x) = \cos(n * \arccos(x))$. Compute T_0, T_1, and T_2 by hand to get a feeling for how it works. Note that they all turn out to be (simple) polynomial functions. In fact, T_n is a polynomial function of degree n for all n. To prove this, here are a few hints:

- $cos(\alpha) + cos(\beta) = 2\cos((\alpha + \beta)/2)\cos((\alpha - \beta)/2)$
- let $\alpha = (n + 1) * \arccos(x)$ and $\beta = (n - 1) * \arccos(x)$
- Simplify $T_{n+1}(x) + T_{n-1}(x)$ to relate it to $T_n(x)$.
- Note that the relation can be seen as an inductive definition of $T_{n+1}(x)$.
- Use induction on n.

Exercise 5.9. Another view of T_n from Exercise 5.8 is as a homomorphism. Let $H_1\ (h,F,f) = \forall\ x.\ h\ (F\ x) == f\ (h\ x)$ be the predicate that states "$h : A \to B$ is a homomorphism from $F : A \to A$ to $f : B \to B$". Show that $H_1\ (cos, (n*), T_n)$ holds, where $cos : \mathbb{R}_{\geq 0} \to [-1,1]$, $(n*) : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, and $T_n : [-1,1] \to [-1,1]$.

Exercise 5.10. Complete the following definition for polynomials represented as a plain list of coefficients:

```
instance Num a => Num [a] where
  (+) = addP
  (*) = mulP
    -- ... TODO
addP :: Num a => [a] -> [a] -> [a]
addP = zipWith' (+)
mulP :: Num a => [a] -> [a] -> [a]
mulP =    -- TODO
```

Note that *zipWith'* is almost, but not quite, the definition of *zipWith* from the standard Haskell prelude.

Exercise 5.11. What are the ring operations on $Poly'\ A$ where

$$Poly'\ A = \{\, a : \mathbb{N} \to A \mid \text{\{- } a \text{ has only a finite number of non-zero values -\}}\ \}$$

Exercise 5.12. Prove the *degree* law

$$\forall\ x, y.\ degree\ (x\ `op`\ y) = degree\ x + degree\ y$$

for polynomials.

Exercise 5.13. Check all the *Monoid* and homomorphism properties in this claim: "*degree* is a monoid homomorphism from $(Poly\ a, 1, *)$ to $(Maybe\ \mathbb{N}, Just\ 0, opMaybe)$".

Exercise 5.14. The helper function $mapPoly :: (a \to b) \to (Poly\ a \to Poly\ b)$ that was used in the implementation of *polyNeg* is a close relative of the usual $map :: (a \to b) \to ([a] \to [b])$. Both these are members of a typeclass called *Functor*:

```
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
```

Implement an instance of *Functor* for *Maybe* and *ComplexSyn* from Chapter 1 and for *Rat* from Chapter 2.

Is *fmap f* a homomorphism?

```
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE RebindableSyntax #-}
```

```
{-# LANGUAGE TypeSynonymInstances #-}
module DSLsofMath.W06 where
import DSLsofMath.FunExp hiding (eval, f)
import DSLsofMath.W05
import DSLsofMath.Algebra
import Prelude hiding (Num (..), (/), (^), Fractional (..), Floating (..))
import Prelude (abs)
```

Chapter 6

Higher-order Derivatives and their Applications

6.1 Review

- key notion *homomorphism*: $S_1 \to S_2$ (read "from S_1 to S_2")
- questions ("equations"):
  - $S_1 \overset{?}{\to} S_2$ what is the homomorphism between two given structures
    - e.g., $apply\ c : Num\ (x \to a) \to Num\ a$
  - $S_1? \to S_2$ what is S_1 compatible with a given homomorphism
    - e.g., $eval : Poly\ a \to (a \to a)$
  - $S_1 \to S_2?$ what is S_2 compatible with a given homomorphism
    - e.g., $applyFD\ c : FD\ a \to (a, a)$
  - $S_1 \overset{?}{\to} S_2?$ can we find a good structure on S_2 so that it becomes homomorphic w. S_1?
    - e.g., $evalD : FunExp \to FD\ a$

TODO: JP: I don't get it. More words are needed in the itemize. Is this connected with what follows? It seems that the review ends before the end of the subsection. It seems that the flow goes all over the place. Using an example or generalising, using several methods, etc. I have a really hard time following where one is going, perhaps the whole sub section should be rewritten.

TODO: JP: Can we derive some function which is a homomorphism between those two structures?

TODO: JP: If the function and the 2nd structure are given, what is the 1st structure that one has to have?

TODO: JP: (Say that we have the composition fo syntactic derivative and eval.)

The importance of *applyFD* (from Section 4.9) and *evalD* (Section 4.6) lies in that they offer "automatic differentiation", i.e., any function constructed according to the grammar of *FunExp*, can be "lifted" to a function that computes the derivative (e.g., a function on pairs).

Example $f :: Transcendental\ a \Rightarrow a \to a$
$f\ x = sin\ x + 2 * x$

We have: $f\ 0 = 0, f\ 2 = 4.909297426825682$, etc.

To compute the derivative at some point, say 2, we have several choices.

1. Using *FunExp*

   Recall expressions (or functions) of one variables, from Section 3.1.1:

   $$\begin{aligned}
   \textbf{data}\ \mathit{FunExp} = {} & \mathit{Const}\ \mathit{Rational} \\
   \mid {} & \mathit{Id} \\
   \mid {} & \mathit{FunExp} :+: \mathit{FunExp} \\
   \mid {} & \mathit{FunExp} :*: \mathit{FunExp} \\
   \mid {} & \mathit{FunExp} :/: \mathit{FunExp} \\
   \mid {} & \mathit{Exp}\ \mathit{FunExp} \\
   \mid {} & \mathit{Sin}\ \mathit{FunExp} \\
   \mid {} & \mathit{Cos}\ \mathit{FunExp} \\
   & \text{-- and so on} \\
   \textbf{deriving}\ & (\mathit{Eq}, \mathit{Show})
   \end{aligned}$$

   What is the expression e for which $f = eval\ e$?

   We have

   $$\begin{aligned}
   & eval\ e\ x = f\ x \\
   \Leftrightarrow\ & eval\ e\ x = sin\ x + 2 * x \\
   \Leftrightarrow\ & eval\ e\ x = eval\ (Sin\ Id)\ x + eval\ (Const\ 2 :*: Id)\ x \\
   \Leftrightarrow\ & eval\ e\ x = eval\ ((Sin\ Id) :+: (Const\ 2 :*: Id))\ x \\
   \Leftarrow\ & e = Sin\ Id :+: (Const\ 2 :*: Id)
   \end{aligned}$$

   Finally, we can apply *derive* and obtain

   $$\begin{aligned}
   & e = Sin\ Id :+: (Const\ 2 :*: Id) \\
   & f'\ 2 = evalFunExp\ (derive\ e)\ 2
   \end{aligned}$$

   This can hardly be called "automatic", look at all the work we did in deducing e! However, consider this definition:

   $$\begin{aligned}
   & e_2 :: FunExp \\
   & e_2 = f\ Id
   \end{aligned}$$

   As $Id :: FunExp$, the Haskell interpreter will look for *FunExp* instances of *Num* and other numeric classes and build the syntax tree for f instead of computing its semantic value. (Perhaps it would have been better to use, in the definition of *FunExp*, the constructor name *X* instead of *Id*.)

   In general, to find the derivative of a function $f :: Transcendental\ a \Rightarrow a \rightarrow a$, we can use

   $$drv\ f = evalFunExp\ (derive\ (f\ Id))$$

2. Using *FD* (pairs of functions)

   Recall

   $$\textbf{type } FD\ a = (a \to a, a \to a)$$
   $$applyFD\ x\ (f, g) = (f\ x, g\ x)$$

   The operations (the numeric type class instances) on $FD\ a$ are such that, if $eval\ e = f$, then

   $$(eval\ e, eval'\ e) = (f, f')$$

   We are looking for (g, g') such that

   $$f\ (g, g') = (f, f') \quad \text{-- (*)}$$

   so we can then do

   $$f'\ 2 = snd\ (applyFD\ 2\ (f\ (g, g')))$$

   We can fullfill (*) if we can find a pair (g, g') that is a sort of "unit" for $FD\ a$:

   $$sin\ (g, g') = (sin, cos)$$
   $$exp\ (g, g') = (exp, exp)$$

   and so on.

   In general, the chain rule gives us

   $$f\ (g, g') = (f \circ g, (f' \circ g) * g')$$

   Therefore, we need: $g = id$ and $g' = const\ 1$.

   Finally

   $$f'\ 2 = snd\ (applyFD\ 2\ (f\ (id, const\ 1)))$$

   In general

   $$drvFD\ f\ x = snd\ (applyFD\ x\ (f\ (id, const\ 1)))$$

   computes the derivative of f at x.

   $$f_1 :: FD\ Double \to FD\ Double$$
   $$f_1 = f$$

3. Using pairs.

   We have **instance** $Transcendental\ a \Rightarrow Transcendental\ (a, a)$, moreover, the instance declaration looks exactly the same as that for $FD\ a$:

```
instance Transcendental a ⇒ Transcendental (FD a) where    -- pairs of functions
  exp (f,f') = (exp f, (exp f) * f')
  sin (f,f') = (sin f, (cos f) * f')
  cos (f,f') = (cos f, -(sin f) * f')
instance Transcendental a ⇒ Transcendental (a,a) where     -- just pairs
  exp (f,f') = (exp f, (exp f) * f')
  sin (f,f') = (sin f, cos f * f')
  cos (f,f') = (cos f, -(sin f) * f')
```

In fact, the latter (just pairs) represents a generalisation of the former (pairs of functions). To see this, note that if we have a *Transcendental* instance for some A, we get a floating instance for $x \to A$ for all x from the module *FunNumInst*. Then from the instance for pairs we get an instance for any type of the form $(x \to A, x \to A)$. As a special case when $x = A$ this includes all $(A \to A, A \to A)$ which is *FD A*. Thus it is enough to have *FunNumInst* and the pair instance to get the "pairs of functions" instance (and more).

TODO: JP: What is this module? Was it ever introduced?

The pair instance is also the "maximally general" such generalisation (discounting the "noise" generated by the less-than-clean design of *Num*, *Fractional*, *Transcendental*).

Still, we need to use this machinery. We are now looking for a pair of values (g, g') such that

$$f\ (g,g') = (f\ 2, f'\ 2)$$

In general

$$f\ (g,g') = (f\ g, (f'\ g) * g')$$

Therefore

$$\begin{aligned} & f\ (g,g') = (f\ 2, f'\ 2) \\ \Leftrightarrow\ & (f\ g, (f'\ g) * g') = (f\ 2, f'\ 2) \\ \Leftarrow\ & g = 2, g' = 1 \end{aligned}$$

Introducing

$$var\ x = (x, 1)$$

we can, as in the case of *FD*, simplify matters a little:

$$f'\ x = snd\ (f\ (var\ x))$$

In general

$$drvP\ f\ x = snd\ (f\ (x, 1))$$

computes the derivative of f at x.

$$\begin{aligned} & f_2 :: (Double, Double) \to (Double, Double) \\ & f_2 = f \end{aligned}$$

TODO: JP: That's probably a bad structure: the example is very long.

We have seen three different ways to use a generic $f :: Transcendental\ a \Rightarrow a \to a$ to compute $f'\ 2$:

- fully symbolic (using *FunExp*),
- using pairs of functions (*FD*),
- or just pairs of values.

6.2 Higher-order derivatives

TODO: JP: Title of the section = title of the chapter? Change?

TODO: JP: Rename *Stream* to *Taylor*?

TODO: JP: Fold ../04/UnusualStream.hs in here

Consider

$$[f, f', f'', \ldots] :: [a \to a]$$

representing the evaluation of an expression (of one variable x) as a function, and all its derivatives:

$$evalAll\ e = (evalFunExp\ e) : evalAll\ (derive\ e)$$

Notice that, if

$$[f, f', f'', \ldots] = evalAll\ e$$

then

$$[f', f'', \ldots] = evalAll\ (derive\ e)$$

Thus $evalAll\ (derive\ e) == tail\ (evalAll\ e)$ which can be written $evalAll \circ derive = tail \circ evalAll$. Thus *evalAll* is a homomorphism from *derive* to *tail*, or in other words, $H_1\ (evalAll, derive, tail)$ (defined in Exercise 4.14).

We want to define the other operations on lists of functions in such a way that *evalAll* is a homomorphism. For example:

$$evalAll\ (e_1 :*: e_2) = evalAll\ e_1 * evalAll\ e_2$$

where the $(*)$ sign stands for the multiplication of infinite lists of functions, the operation we are trying to determine. We assume that we have already derived the definition of $+$ for these lists (it is $zipWith\ (+)$ — and because the lists are infinite one needs not worry about differing lengths).

We have the following (writing *eval* for *evalFunExp* and *d* for *derive* in order to save ink):

$$\begin{aligned} & LHS \\ = & \{\text{- def. -}\} \end{aligned}$$

$$
\begin{aligned}
& \mathit{evalAll}\ (e_1 :*: e_2) \\
= & \ \{\text{- def. of } \mathit{evalAll} \text{ -}\} \\
& \mathit{eval}\ (e_1 :*: e_2) : \mathit{evalAll}\ (d\ (e_1 :*: e_2)) \\
= & \ \{\text{- def. of } \mathit{eval} \text{ for } (:*:) \text{ -}\} \\
& (\mathit{eval}\ e_1 * \mathit{eval}\ e_2) : \mathit{evalAll}\ (d\ (e_1 :*: e_2)) \\
= & \ \{\text{- def. of } \mathit{derive} \text{ for } (:*:) \text{ -}\} \\
& (\mathit{eval}\ e_1 * \mathit{eval}\ e_2) : \mathit{evalAll}\ (d\ e_1 :*: e_2 :+: e_1 * d\ e_2) \\
= & \ \{\text{- we assume } H_2\ (\mathit{evalAll}, (:+:), (+)) \text{ -}\} \\
& (\mathit{eval}\ e_1 * \mathit{eval}\ e_2) : (\mathit{evalAll}\ (d\ e_1 :*: e_2) + \mathit{evalAll}\ (e_1 :*: d\ e_2))
\end{aligned}
$$

Similarly, starting from the other end we get

$$
\begin{aligned}
& \mathit{evalAll}\ e_1 * \mathit{evalAll}\ e_2 \\
= & \\
& (\mathit{eval}\ e_1 : \mathit{evalAll}\ (d\ e_1)) * (\mathit{eval}\ e_2 : \mathit{evalAll}\ (d\ e_2))
\end{aligned}
$$

Now, to see the pattern it is useful to give simpler names to some common subexpressions: let $a = \mathit{eval}\ e_1$ and $b = \mathit{eval}\ e_2$.

$$
\begin{aligned}
& (a * b) : (\mathit{evalAll}\ (d\ e_1 :*: e_2) + \mathit{evalAll}\ (e_1 * d\ e_2)) \\
=? & \\
& (a : \mathit{evalAll}\ (d\ e_1)) * (b : \mathit{evalAll}\ (d\ e_2))
\end{aligned}
$$

Now we can solve part of the problem by defining $(*)$ as

$$(a : as) * (b : bs) = (a * b) : \mathit{help}\ a\ b\ as\ bs$$

The remaining part is then

$$
\begin{aligned}
& \mathit{evalAll}\ (d\ e_1 :*: e_2) + \mathit{evalAll}\ (e_1 * d\ e_2) \\
=? & \\
& \mathit{help}\ a\ b\ (\mathit{evalAll}\ (d\ e_1))\ (\mathit{evalAll}\ (d\ e_2))
\end{aligned}
$$

Informally, we can refer to (co-)induction at this point and rewrite $\mathit{evalAll}\ (d\ e_1 :*: e_2)$ to $\mathit{evalAll}\ (d\ e_1) * \mathit{evalAll}\ e_2$. We also have $\mathit{evalAll} \circ d = \mathit{tail} \circ \mathit{evalAll}$ which leads to:

TODO: JP: what does this mean here?

$$
\begin{aligned}
& \mathit{tail}\ (\mathit{evalAll}\ e_1) * \mathit{evalAll}\ e_2 + \mathit{evalAll}\ e_1 * \mathit{tail}\ (\mathit{evalAll}\ e_2) \\
=? & \\
& \mathit{help}\ a\ b\ (\mathit{tail}\ (\mathit{evalAll}\ e_1))\ (\mathit{tail}\ (\mathit{evalAll}\ e_2))
\end{aligned}
$$

Finally we rename common subexpressions: let $a : as = \mathit{evalAll}\ e_1$ and $b : bs = \mathit{evalAll}\ e_2$.

$$
\begin{aligned}
& \mathit{tail}\ (a : as) * (b : bs) + (a : as) * \mathit{tail}\ (b : bs) \\
=? & \\
& \mathit{help}\ a\ b\ (\mathit{tail}\ (a : as))\ (\mathit{tail}\ (b : bs))
\end{aligned}
$$

This equality is clearly solved by defining *help* as follows:

$$help\ a\ b\ as\ bs = as * (b : bs) + (a : as) * bs$$

Thus, we can eliminate *help* to arrive at a definition for multiplication:

$$mulStream\ (a : as)\ (b : bs) = (a * b) : (as * (b : bs) + (a : as) * bs)$$

As in the case of pairs, we find that we do not need any properties of functions, other than their *Num* structure, so the definitions apply to any infinite list of *Num a*:

TODO: JP: This is different from *polyMul*, but this is because the coefficients here are "bigger". In fact, we compute several times the same thing here and add them together.

```
type Stream a = [a]
instance Additive a => Additive (Stream a) where
  (+) = addStream
instance Ring a => Multiplicative (Stream a) where
  (*) = mulStream
addStream :: Additive a => Stream a -> Stream a -> Stream a
addStream (a : as) (b : bs) = (a + b) : (as + bs)
mulStream :: Ring a => Stream a -> Stream a -> Stream a
```

Exercise 6.1. Complete the instance declarations for *Fractional* and *Transcendental*.

Note that it may make more sense to declare a **newtype** for *Stream a* first, for at least two reasons. First, because the type $[a]$ also contains finite lists, but we use it here to represent only the infinite lists (also known as streams). Second, because there are competing possibilities for *Num* instances for infinite lists, for example applying all the operations "pointwise" as with "FunNumInst". We used just a type synonym here to avoid cluttering the definitions with the newtype constructors.

Write a general derivative computation, similar to *drv* functions above:

```
drvList k f x = undefined    -- kth derivative of f at x
```

Exercise 6.2. Compare the efficiency of different ways of computing derivatives.

TODO: JP: This is a pretty tough exercise...

6.3 Polynomials

TODO: JP: What is this doing here? and not in the relevant chapter?

TODO: JP: Why insisting on non-empty lists? Isn't this trivial?

```
data Poly a = Single a | Cons a (Poly a)
              deriving (Eq, Ord)
evalPoly ::  Num a => Poly a -> a -> a
evalPoly (Single a)   x = a
evalPoly (Cons a as) x = a + x * evalPoly as x
```

6.4 Formal power series

TODO: JP: Are we saying anything non-trivial and new in this section? If so indicate what it is right away.

As we mentioned above, the Haskell list type contains both finite and infinite lists. The same holds for the type *Poly* that we designed as "syntax" for polynomials. Thus we can reuse that type also as "syntax for power series": potentially infinite "polynomials".

$$\textbf{type}\ \mathit{PowerSeries}\ a = \mathit{Poly}\ a \quad \text{-- finite and infinite non-empty lists}$$

Now we can divide, as well as add and multiply.

We can also compute derivatives:

$$\begin{aligned}
&\mathit{deriv}\ (\mathit{Single}\ a) \quad = \mathit{Single}\ 0\\
&\mathit{deriv}\ (\mathit{Cons}\ a\ as) = \mathit{deriv}'\ as\ 1\\
&\quad \textbf{where}\ \mathit{deriv}'\ (\mathit{Single}\ a) \quad n = \mathit{Single}\ (n * a)\\
&\qquad\quad \mathit{deriv}'\ (\mathit{Cons}\ a\ as)\ n = \mathit{Cons} \quad (n * a)\ (\mathit{deriv}'\ as\ (n+1))
\end{aligned}$$

and integrate:

$$\begin{aligned}
&\mathit{integ}\ :: \mathit{Field}\ a \Rightarrow a \to \mathit{PowerSeries}\ a \to \mathit{PowerSeries}\ a\\
&\mathit{integ}\ a_0\ as = \mathit{Cons}\ a_0\ (\mathit{integ}'\ as\ 1)\\
&\quad \textbf{where}\ \mathit{integ}'\ (\mathit{Single}\ a) \quad n = \mathit{Single}\ (a\ /\ n)\\
&\qquad\quad \mathit{integ}'\ (\mathit{Cons}\ a\ as)\ n = \mathit{Cons} \quad (a\ /\ n)\ (\mathit{integ}'\ as\ (n+1))
\end{aligned}$$

Note that a_0 is the constant that we need due to indefinite integration.

These operations work on the type *PowerSeries a* which we can see as the syntax of power series, often called "formal power series". The intended semantics of a formal power series a is, as we saw in Chapter 5, an infinite sum

$$\begin{aligned}
&\mathit{eval}\ a : \mathbb{R} \to \mathbb{R}\\
&\mathit{eval}\ a = \lambda x \to \mathit{lim}\ s\ \textbf{where}\ s\ n = \textstyle\sum_{i=0}^{n} a_i * x^i
\end{aligned}$$

For any n, the prefix sum, $s\ n$, is finite and it is easy to see that the derivative and integration operations are well defined. We we take the limit, however, the sum may fail to converge for certain values of x. Fortunately, we can often ignore that, because seen as operations from syntax to syntax, all the operations are well defined, irrespective of convergence.

If the power series involved do converge, then *eval* is a morphism between the formal structure and that of the functions represented:

$$\begin{aligned}
&\mathit{eval}\ as + \mathit{eval}\ bs \quad = \mathit{eval}\ (as + bs) \quad \text{-- } H_2\ (\mathit{eval}, (+), (+))\\
&\mathit{eval}\ as * \mathit{eval}\ bs \quad = \mathit{eval}\ (as * bs) \quad \text{-- } H_2\ (\mathit{eval}, (*), (*))\\
&\mathit{eval}\ (\mathit{derive}\ as) \quad = D\ (\mathit{eval}\ as) \quad \text{-- } H_1\ (\mathit{eval}, \mathit{derive}, D)\\
&\mathit{eval}\ (\mathit{integ}\ c\ as)\ x = c + \textstyle\int_0^x (\mathit{eval}\ as\ t)\ dt
\end{aligned}$$

6.5 Simple differential equations

Many first-order differential equations have the structure

$$f'\ x = g\ f\ x, \qquad f\ 0 = f_0$$

i.e., they are defined in terms of the higher-order function g.

The fundamental theorem of calculus gives us

$$f\ x = f_0 + \int_0^x (g\ f\ t)\ dt$$

If $f = eval\ as$

$$eval\ as\ x = f_0 + \int_0^x (g\ (eval\ as)\ t)\ dt$$

Assuming that g is a polymorphic function defined both for the syntax (*PowerSeries*) and the semantics ($\mathbb{R} \to \mathbb{R}$), and that

$$\forall\ as.\ \ eval\ (g_{syn}\ as) == g_{sem}\ (eval\ as)$$

or simply $H_1\ (eval, g, g)$. (This particular use of H_1 is read "g commutes with *eval*".) Then we can move *eval* outwards step by step:

$$\begin{aligned} & eval\ as\ x = f_0 + \int_0^x (eval\ (g\ as)\ t)\ dt \\ \Leftrightarrow\ & eval\ as\ x = eval\ (integ\ f_0\ (g\ as))\ x \\ \Leftarrow\ & as = integ\ f_0\ (g\ as) \end{aligned}$$

Finally, we have arrived at an equation expressed in only syntactic operations, which is implementable in Haskell (for a reasonable g).

Which functions g commute with *eval*? All the ones in *Num*, *Fractional*, *Transcendental*, by construction; additionally, as above, *deriv* and *integ*.

Therefore, we can implement a general solver for these simple equations:

```
solve :: Field a ⇒ a → (PowerSeries a → PowerSeries a) → PowerSeries a
solve f0 g = f    -- solves f' = g f, f 0 = f0
  where f = integ f0 (g f)
```

On the face of it, the solution f appears not well defined, because its definition depends on itself. We come back to this point soon, but first we observe *solve* in action on simple instances of g, starting with *const* 1 and *id*:

TODO: JP: It'd be helpful to restate what the differential equation looks like using usual mathematical notation.

```
idx :: Field a ⇒ PowerSeries a
idx = solve 0 (λf → 1)
idf :: Field a ⇒ a → a
idf = eval 100 idx

expx :: Field a ⇒ PowerSeries a
expx = solve 1 (λf → f)
expf :: Field a ⇒ a → a
expf = eval 100 expx
```

Exercise 6.3. Write *expx* as a recursive equation (inline *solve* in the definition above).

TODO: JP: *eval* was declared $\mathbb{R} \to \mathbb{R}$ above. Which eval is this referring to?

The first solution, *idx* is just the polynomial $[0, 1]$ — i.e. just x in usual mathematical notation. We can easily check that its derivative is constantly 1 and its value at 0 is 0. The function *idf* is just there to check that the semantics behaves as expected.

TODO: JP: So what should we check? Is this meant to be a sanity check? Then we should print something?

The second solution *expx* is a formal power series representing the exponential function. It is equal to its derivative and it starts at 1. The function *expf* is a good approximation of the semantics for small values of its argument.

$$
\begin{aligned}
&\mathit{testExp} :: \mathit{Double}\\
&\mathit{testExp} = \mathit{maximum}\,\$\,\mathit{map}\;\mathit{diff}\;[0, 0.001\,..\,1 :: \mathit{Double}]\\
&\quad \textbf{where}\;\mathit{diff} = \mathit{abs}\;(\mathit{expf} - \mathit{exp}) \quad \text{-- using the function instances for } \mathit{abs} \text{ and } \mathit{exp}\\
&\mathit{testExpUnits} :: \mathit{Double}\\
&\mathit{testExpUnits} = \mathit{testExp}\;/\;\epsilon\\
&\epsilon :: \mathit{Double} \quad \text{-- one bit of } \mathit{Double} \text{ precision}\\
&\epsilon = \mathit{last}\,\$\,\mathit{takeWhile}\;(\lambda x \to 1 + x \neq 1)\;(\mathit{iterate}\;(/2)\;1)
\end{aligned}
$$

We can also use mutual recursion to define sine and cosine in terms of each other: TODO: JP: why "also"?

$$
\begin{aligned}
&\mathit{sinx} = \mathit{integ}\;0\;\mathit{cosx}\\
&\mathit{cosx} = \mathit{integ}\;1\;(-\mathit{sinx})\\
&\mathit{sinf} = \mathit{eval}\;100\;\mathit{sinx}\\
&\mathit{cosf} = \mathit{eval}\;100\;\mathit{cosx}\\
&\mathit{sinx}, \mathit{cosx} :: \mathit{Field}\;a \Rightarrow \mathit{PowerSeries}\;a\\
&\mathit{sinf}, \mathit{cosf} :: \mathit{Field}\;a \Rightarrow\; a \to a
\end{aligned}
$$

Exercise 6.4. Write the differential equations characterising sine and cosine, using usual mathematical notation.

The reason that these definitions "work" (in the sense of not looping) is because *integ* immediately returns the first element of the stream before requesting any information about its first input. It is instructive to mimic part of what the lazy evaluation machinery is doing "by hand" as follows. We know that both *sinx* and *cosx* are streams, thus we can start by filling in just the very top level structure:

$$
\begin{aligned}
&\mathit{sx} = \mathit{sh} : \mathit{st}\\
&\mathit{cx} = \mathit{ch} : \mathit{ct}
\end{aligned}
$$

where *sh* & *ch* are the heads and *st* & *ct* are the tails of the two streams. Then we notice that *integ* fills in the constant as the head, and we can progress to:

$$sx = 0 : st$$
$$cx = 1 : ct$$

At this stage we only know the constant term of each power series, but that is enough for the next step: the head of st is $\frac{1}{1}$ and the head of ct is $\frac{\text{-}0}{1}$:

$$sx = 0 : 1 \ : \_$$
$$cx = 1 : \text{-}0 : \_$$

As we move on, we can always compute the next element of one series by the previous element of the other series (divided by n, for cx negated).

$$sx, cx :: [Double]$$
$$sx = 0 : 1 \ : \text{-}0 : \tfrac{\text{-}1}{6} : error\ \texttt{"TODO"}$$
$$cx = 1 : \text{-}0 : \tfrac{\text{-}1}{2} : 0 \ : error\ \texttt{"TODO"}$$

6.6 Exponentials and trigonometric functions for *PowerSeries*

Can we compute *exp as*?

TODO: JP: Compute in what sense? We already have *expx* in the above section. Does this mean using the differential rather than the integral? I don't get the point.

Specification:

$$eval\ (exp\ as) = exp\ (eval\ as)$$

Differentiating both sides, we obtain

TODO: JP: What definition of D are we using?

$$\begin{aligned}
&D\ (eval\ (exp\ as)) = exp\ (eval\ as) * D\ (eval\ as) \\
\Leftrightarrow\ &\{\text{- } eval \text{ morphism -}\} \\
&eval\ (deriv\ (exp\ as)) = eval\ (exp\ as * deriv\ as) \\
\Leftarrow\ &\\
&deriv\ (exp\ as) = exp\ as * deriv\ as
\end{aligned}$$

Adding the "initial condition" $eval\ (exp\ as)\ 0 = exp\ (head\ as)$, we obtain

TODO: JP: What is this coming from? Why suddenly using *head*? I am lost.

$$exp\ as = integ\ (exp\ (head\ as))\ (exp\ as * deriv\ as)$$

Note: we cannot use *solve* here, because the g function uses both *exp as* and *as* (it "looks inside" its argument).

TODO: JP: So what's happening now? Why this code suddenly?

$$\begin{aligned}
&\textbf{instance}\ (Eq\ a, Transcendental\ a) \Rightarrow Transcendental\ (PowerSeries\ a)\ \textbf{where} \\
&\quad \pi \ = Single\ \pi \\
&\quad exp \ = expPS \\
&\quad sin \ = sinPS \\
&\quad cos \ = cosPS
\end{aligned}$$

$$
\begin{aligned}
&expPS, sinPS, cosPS :: (Eq\ a, Transcendental\ a) \Rightarrow PowerSeries\ a \to PowerSeries\ a \\
&expPS\ \ fs\ \ = integ\ (exp\ (val\ fs))\ (exp\ fs\ \ * deriv\ fs) \\
&sinPS\ \ fs\ \ = integ\ (sin\ \ (val\ fs))\ (cos\ fs\ \ * deriv\ fs) \\
&cosPS\ \ fs\ \ = integ\ (cos\ \ (val\ fs))\ (-sin\ fs * deriv\ fs) \\
&val :: PowerSeries\ a \to \ a \\
&val\ \ (Single\ a) \ \ = \ a \\
&val\ \ (Cons\ a\ as) \ \ = \ a
\end{aligned}
$$

In fact, we can implement *all* the operations needed for evaluating *FunExp* functions as power series!

TODO: JP: Wasn't it done already when first talking about power series? There does not seem to be anything pertaining derivatives here.

$$
\begin{aligned}
&evalP :: (Eq\ r, Transcendental\ r) \Rightarrow FunExp \to PowerSeries\ r \\
&evalP\ (Const\ x) \ \ = Single\ (fromRational\ (toRational\ x)) \\
&evalP\ (e_1\ :+:\ e_2) = evalP\ e_1 + evalP\ e_2 \\
&evalP\ (e_1\ :*:\ e_2) \ \ = evalP\ e_1 * evalP\ e_2 \\
&evalP\ (e_1\ :/:\ e_2) \ \ = evalP\ e_1\ /\ evalP\ e_2 \\
&evalP\ Id \qquad\quad = idx \\
&evalP\ (Exp\ e) \qquad = exp\ (evalP\ e) \\
&evalP\ (Sin\ e) \qquad = sin\ (evalP\ e) \\
&evalP\ (Cos\ e) \qquad = cos\ (evalP\ e)
\end{aligned}
$$

6.7 Taylor series

If $f = eval\ [a_0, a_1, ..., a_n, ...]$, then

TODO: JP: Which eval is that, and what is the meaning of the list here? polynomial? power series? derivatives?

$$
\begin{aligned}
&f\ 0 \ \ = a_0 \\
&f' \quad = eval\ (deriv\ [a_0, a_1, ..., a_n, ...]) \\
&\qquad = eval\ ([1 * a_1, 2 * a_2, 3 * a_3, ..., n * a_n, ...]) \\
&\Rightarrow \\
&f'\ 0 \ \ = a_1 \\
&f'' \quad = eval\ (deriv\ [a_1, 2 * a_2, ..., n * a_n, ...]) \\
&\qquad = eval\ ([2 * a_2, 3 * 2 * a_3, ..., n * (n-1) * a_n, ...]) \\
&\Rightarrow \\
&f''\ 0 = 2 * a_2
\end{aligned}
$$

In general:

$$f^{(k)}0 = fact\ k * a_k$$

Therefore

$$f = eval\ [f\ 0, f'\ 0, f''\ 0\ /\ 2, ..., f^{(n)}0\ /\ (fact\ n), ...]$$

That is, there is a simple mapping between the representation of f as a power series (the coefficients *a_k*), and the value of all derivatives of f at 0 (our *Stream a* type above).

The series $[f\ 0, f'\ 0, f''\ 0\ /\ 2, ..., f^{(n)}0\ /\ (fact\ n), ...]$ is called the Taylor series centred in 0, or the Maclaurin series.

```
derivs :: Ring a ⇒ PowerSeries a → PowerSeries a
derivs as = derivs1 as 0 1
  where
  derivs1 (Cons a as) n factn = Cons   (a ∗ factn)
                                       (derivs1 as (n + 1) (factn ∗ (n + 1)))
  derivs1 (Single a)   n factn = Single (a ∗ factn)
  -- remember that x = Cons 0 (Single 1)
ex3, ex4 :: Poly Double
ex3 = takePoly 10 (derivs (x^3 + 2 ∗ x))
ex4 = takePoly 10 (derivs sinx)
```

In this way, we can compute all the derivatives at 0 for all functions f constructed with the grammar of *FunExp*. That is because, as we have seen, we can represent all of them by power series!

What if we want the value of the derivatives at $a \neq 0$?

We then need the power series of the "shifted" function g:

$$g\ x = f\ (x + a) \Leftrightarrow g = f \circ (+a)$$

If we can represent g as a power series, say $[b_0, b_1, ...]$, then we have

$$g^{(k)}0 = fact\ k * b_k = f^{(k)}a$$

In particular, we would have

$$f\ x = g\ (x - a) = \sum b_n * (x - a)^n$$

which is called the Taylor expansion of f at a.

Example:

We have that $idx = [0, 1]$, thus giving us indeed the values

$$[id\ 0, id'\ 0, id''\ 0, ...]$$

In order to compute the values of

$$[id\ a, id'\ a, id''\ a, ...]$$

for $a \neq 0$, we compute

$$ida\ a = takePoly\ 10\ (derivs\ (evalP\ (Id :+: Const\ a)))$$

More generally, if we want to compute the derivative of a function f constructed with *FunExp* grammar, at a point a, we need the power series of $g\ x = f\ (x + a)$:

$$d\ f\ a = takePoly\ 10\ (derivs\ (evalP\ (f\ (Id :+: Const\ a))))$$

Use, for example, our $f\ x = sin\ x + 2 * x$ above.

As before, we can use directly power series:

$$dP\ f\ a = takePoly\ 10\ (derivs\ (f\ (idx + Single\ a)))$$

6.8 Associated code

TODO: JP: Feels like this should be moved upwards as the concepts are introduced

```
evalFunExp :: Transcendental a ⇒ FunExp → a → a
evalFunExp (Const α)  = const (fromRational (toRational α))
evalFunExp Id         = id
evalFunExp (e1 :+: e2) = evalFunExp e1 + evalFunExp e2   -- note the use of "lifted +"
evalFunExp (e1 :*: e2) = evalFunExp e1 *  evalFunExp e2   -- "lifted *"
evalFunExp (Exp e1)   = exp (evalFunExp e1)              -- and "lifted exp"
evalFunExp (Sin e1)   = sin (evalFunExp e1)
evalFunExp (Cos e1)   = cos (evalFunExp e1)
  -- and so on
derive (Const α)  = Const 0
derive Id         = Const 1
derive (e1 :+: e2) = derive e1 :+: derive e2
derive (e1 :*: e2) = (derive e1 :*: e2) :+: (e1 :*: derive e2)
derive (Exp e)    = Exp e :*: derive e
derive (Sin e)    = Cos e :*: derive e
derive (Cos e)    = Const (−1) :*: Sin e :*: derive e
instance Additive FunExp where
  (+) = (:+:)
  zero = Const 0
instance AddGroup FunExp where
  negate x  = Const (−1) * x
instance Multiplicative FunExp where
  (*)  = (:*:)
  one  = Const 1
instance MulGroup FunExp where
```

```
  (/) = (:/:)
instance Transcendental FunExp where
  exp =       Exp
  sin =       Sin
  cos =       Cos
```

6.8.1 Not included to avoid overlapping instances

```
instance Num a => Num (FD a) where
  (f,f') + (g,g') = (f + g,f' + g')
  zero = (zero,zero)
instance Multiplicative (FD a) where
  (f,f') * (g,g')  = (f * g,f' * g + f * g')
  one = (one,zero)
instance Field a => MulGroup (FD a) where
  (f,f') / (g,g')  = (f / g,(f' * g - g' * f) / (g * g))
instance Transcendental a => Transcendental (FD a) where
  exp (f,f')       = (exp f,(exp f) * f')
  sin (f,f')       = (sin f,(cos f) * f')
  cos (f,f')       = (cos f,-(sin f) * f')
```

6.8.2 This is included instead

```
instance Additive a => Additive (a,a) where
  (f,f') + (g,g') = (f + g,f' + g')
  zero = (zero,zero)
instance AddGroup a => AddGroup (a,a) where
  negate (f,f') = (negate f,negate f')
instance Ring a => Multiplicative (a,a) where
  (f,f') * (g,g')  = (f * g,f' * g + f * g')
  one = (one,zero)
instance Field a => MulGroup (a,a) where
  (f,f') / (g,g')  = (f / g,(f' * g - g' * f) / (g * g))
instance Transcendental a => Transcendental (a,a) where
  exp (f,f')       = (exp f,(exp f) * f')
  sin (f,f')       = (sin f,cos f * f')
  cos (f,f')       = (cos f,-(sin f) * f')
```

TODO: JP: What about log?

TODO: JP: What about adding composition to the language of functions? See attempt in Taylor.hs

6.9 Exercises

Exercise 6.5. As shown at the start of the chapter, we can find expressions $e :: FunExp$ such that $eval\ e = f$ automatically using the assignment $e = f\ Id$. This is possible thanks to the *Num*, *Fractional*, and *Floating* instances of *FunExp*. Use this method to find *FunExp* representations of the functions below, and show step by step how the application of the function to *Id* is evaluated in each case.

1. $f_1\ x = x\hat{}2 + 4$
2. $f_2\ x = 7 * exp\ (2 + 3 * x)$
3. $f_3\ x = 1\ /\ (sin\ x + cos\ x)$

Exercise 6.6. For each of the expressions $e :: FunExp$ you found in Exercise 6.5, use *derive* to find an expression $e' :: FunExp$ representing the derivative of the expression, and verify that e' is indeed the derivative of e.

Exercise 6.7. At the start of this chapter, we saw three different ways of computing the value of the derivative of a function at a given point:

1. Using *FunExp*
2. Using *FD*
3. Using pairs

Try using each of these methods to find the values of $f_1'\ 2$, $f_2'\ 2$, and $f_3'\ 2$, i.e. the derivatives of each of the functions in Exercise 6.5, evaluated at the point 2. You can verify that the result is correct by comparing it with the expressions e_1', e_2' and e_3' that you found in 6.6.

Exercise 6.8. The exponential function $exp\ t = e\hat{}t$ has the property that $\int exp\ t\ dt = exp\ t + C$. Use this fact to express the functions below as *PowerSeries* using *integ*. *Hint: the definitions will be recursive.*

1. $\lambda t \rightarrow exp\ t$
2. $\lambda t \rightarrow exp\ (3 * t)$
3. $\lambda t \rightarrow 3 * exp\ (2 * t)$

Exercise 6.9. In the chapter, we saw that a representation $expx :: PowerSeries$ of the exponential function can be implemented using *solve* as $expx = solve\ 1\ (\lambda f \rightarrow f)$. Use the same method to implement power series representations of the following functions:

1. $\lambda t \to exp\ (3 * t)$
2. $\lambda t \to 3 * exp\ (2 * t)$

Exercise 6.10.

1. Implement *idx′*, *sinx′* and *cosx′* using *solve*
2. Complete the instance *Floating* (*PowerSeries a*)

Exercise 6.11. Consider the following differential equation:

$$f''\ t + f'\ t - 2 * f\ t = e^{3*t}, \quad f\ 0 = 1, \quad f'\ 0 = 2$$

We will solve this equation assuming that f can be expressed by a power series *fs*, and finding the three first coefficients of *fs*.

1. Implement *expx3* :: *PowerSeries* $\mathbb{R}$, a power series representation of e^{3*t}
2. Find an expression for *fs″*, the second derivative of *fs*, in terms of *expx3*, *fs′*, and *fs*.
3. Find an expression for *fs′* in terms of *fs″*, using *integ*.
4. Find an expression for *fs* in terms of *fs′*, using *integ*.
5. Use *takePoly* to find the first three coefficients of *fs*. You can check that your solution is correct using a tool such as MATLAB or WolframAlpha, by first finding an expression for $f\ \ t$, and then getting the Taylor series expansion for that expression.

Exercise 6.12. *From exam 2016-03-15*

Consider the following differential equation:

$$f''\ t - 2 * f'\ t + f\ t = e^{2*t}, \quad f\ 0 = 2, \quad f'\ 0 = 3$$

Solve the equation assuming that f can be expressed by a power series *fs*, that is, use *deriv* and *integ* to compute *fs*. What are the first three coefficients of *fs*?

Exercise 6.13. *From exam 2016-08-23*

Consider the following differential equation:

$$f''\ t - 5 * f'\ t + 6 * f\ t = e^{t}, \quad f\ 0 = 1, \quad f'\ 0 = 4$$

Solve the equation assuming that f can be expressed by a power series *fs*, that is, use *deriv* and *integ* to compute *fs*. What are the first three coefficients of *fs*?

Exercise 6.14. *From exam 2016-Practice*

Consider the following differential equation:

$$f'' t - 2 * f' t + f t - 2 = 3 * e^{2*t}, \quad f 0 = 5, \quad f' 0 = 6$$

Solve the equation assuming that f can be expressed by a power series *fs*, that is, use *deriv* and *integ* to compute *fs*. What are the first three coefficients of *fs*?

Exercise 6.15. *From exam 2017-03-14*

Consider the following differential equation:

$$f'' t + 4 * f t = 6 * \cos t, \quad f 0 = 0, \quad f' 0 = 0$$

Solve the equation assuming that f can be expressed by a power series *fs*, that is, use *integ* and the differential equation to express the relation between *fs*, *fs*′, *fs*″, and *rhs* where *rhs* is the power series representation of $(6*) \circ cos$. What are the first four coefficients of *fs*?

Exercise 6.16. *From exam 2017-08-22*

Consider the following differential equation:

$$f'' t - 3\sqrt{2} * f' t + 4 * f t = 0, \quad f 0 = 2, \quad f' 0 = 3\sqrt{2}$$

Solve the equation assuming that f can be expressed by a power series *fs*, that is, use *integ* and the differential equation to express the relation between *fs*, *fs*′, and *fs*″. What are the first three coefficients of *fs*?

Chapter 7

Elements of Linear Algebra

```
{-# LANGUAGE ConstraintKinds #-}
{-# LANGUAGE GADTs, FlexibleInstances, UndecidableInstances #-}
{-# LANGUAGE GeneralizedNewtypeDeriving, RebindableSyntax #-}
module DSLsofMath.W07 where
import DSLsofMath.Algebra
import Prelude hiding (Num (..), (/), (^), Fractional (..), Floating (..), sum)
import Data.List (nub)
type ℝ = Double
```

Often, especially in engineering textbooks, one encounters the following definition: a vector is an $n+1$-tuple of real or complex numbers, arranged as a column:

$$v = \begin{bmatrix} v_0 \\ \vdots \\ v_n \end{bmatrix}$$

Other times, this is supplemented by the definition of a row vector:

$$v = \begin{bmatrix} v_0 & \cdots & v_n \end{bmatrix}$$

The v_is are real or complex numbers, or, more generally, elements of a *field*.

TODO: JP: track down the first time we use fields and define *Field* there.

However, following our theme, we will first characterize vectors algebraically. From this perpective a *vector space* is an algebraic structure that captures a set of vectors, with zero, a commutative addition, and scaling by a set of scalars (i.e., elements of the field). In terms of typeclasses, we can characterize this structure as follows:

```
class (Field s, AddGroup v) ⇒ VectorSpace v s where
  (⊛) :: s → v → v
```

Additionally, vector scaling $(s\circledast)$ must be a homomorphism over (from and to) the additive group structure of v:

$$\begin{aligned} s \circledast (a + b) &= s \circledast a + s \circledast b \\ s \circledast zero &= zero \\ s \circledast (negate\ a) &= negate\ (s \circledast a) \end{aligned}$$

And, on the other side, $(\circledast a)$ is a homomorphism from the additive group structure of s to the group structure of v:

$$\begin{aligned} (s + t) \circledast a &= s \circledast a + t \circledast a \\ zero \circledast a &= zero \\ negate\ s \circledast a &= negate\ (s \circledast a) \end{aligned}$$

The multiplicative structure of s maps the monoid of endofunctions (See Section 4.1) as follows:

$$\begin{aligned} one \circledast a &= id & a &= a \\ (s * t) \circledast a &= ((s\circledast) \circ (t\circledast)) & a &= s \circledast (t \circledast a) \end{aligned}$$

[1]

An important consequence of their algebraic structure is that vectors can be expressed as a simple sort of combination of other special vectors. More precisely, we can *uniquely* represent any vector v in the space in terms of a fixed set of *basis* vectors $\{b_0, ..., b_n\}$ which cover the whole space and are *linearly independent*:

$$(s0 \circledast b_0 + ... + sn \circledast b_n = 0) \Leftrightarrow (s0 = ... = sn = 0)$$

One can prove the uniqueness of representation as follows.

Proof. Assume two representations of v, given by si and ti. The difference of those representations is given by $si - ti$. But because they represent the same vector, they must be equal to the zero vector: $(s0 - t_0) \circledast b_0 + ... + (sn - tn) \circledast b_n = 0$. By the basis being linearly independent, we find $si - ti = 0$, and $si = ti$. □

This representation is what justifies the introduction of vectors as columns (or rows) of numbers. (According to our red thread, this representation is akin to the notion of "syntax".) Indeed, we can define:

$$v = \begin{bmatrix} v_0 \\ \vdots \\ v_n \end{bmatrix} = v_0 \circledast \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} + v_1 \circledast \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix} + \cdots + v_n \circledast \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

[1] Traditionally some of the above laws are omitted because they are consequences of other laws.

So, for our column vectors, we can define the operations as follows:

$$v + w = \begin{bmatrix} v_0 \\ \vdots \\ v_n \end{bmatrix} + \begin{bmatrix} w_0 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} v_0 + w_0 \\ \vdots \\ v_n + w_n \end{bmatrix}$$

$$s \circledast v = \begin{bmatrix} s * v_0 \\ \vdots \\ s * v_n \end{bmatrix}$$

In the following we denote by

$$e_k = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \leftarrow \text{position } k$$

the canonical base vectors, ie. the vector that is everywhere 0 except at position k, where it is 1, so that $v = v_0 \circledast e_0 + ... + v_n \circledast e_n$. This formula maps the syntax (coefficients) to the semantics (a vector).

7.1 Representing vectors as functions

In what follows we will systematically use the represention of vectors as a linear combination of basis vectors. There is a temptation to model the corresponding set of coefficients as a lists or tuples, but a more general (and conceptually simpler) way is to view them as *functions* from a set of indices G:

newtype *Vector s g* = *V* $(g \to s)$ **deriving** $(Additive, AddGroup)$

We define right away the notation $a\,!\,i$ for the coefficient of the base vector $e\ i$, as follows:

```
infix 9 !
(!) :: Vector s g → g → s
V f ! i = f i
```

As discussed, the S parameter in *Vector S* has to be a field ($\mathbb{R}$, or *Complex*, or $\mathbb{Z}_n$, etc.) for values of type *Vector S G* to represent elements of a vector space.

The cardinality of G, which we sometimes denote *card* G, is number of basis vectors, and thus the dimension of the vector space. Often G is finite and in the examples so far we have used indices from $G = \{0, \ldots, n\}$. Thus the dimension of the space would be $n + 1$.

In Haskell finiteness of G can be captured by the conjunction of *Bounded* (there is a minimum and a maximum element in G) and *Enum*erable (there is a notion of enumeration from a given element of G) and *Eq*. Hence, the list of all elements of G can be extracted:

$$\begin{aligned}&\textbf{type } \mathit{Finite}\ g = (\mathit{Bounded}\ g, \mathit{Enum}\ g, \mathit{Eq}\ g)\\ &\mathit{finiteDomain} :: \mathit{Finite}\ a \Rightarrow [a]\\ &\mathit{finiteDomain} = [\mathit{minBound} \,..\, \mathit{maxBound}]\end{aligned}$$

We know from the previous lectures that if S is an instance of *AddGroup* then so is $G \to S$, with the pointwise definitions. However, multiplication of vectors does not in general work pointwise. In fact, attempting to lift multiplication from the *Multiplicative* class would give a homogenous multiplication operator $(*) :: v \to v \to v$, but such an operator is not part of the definition of vector spaces. Consequently, vector spaces are *not* rings.

Instead, the scaling operator $(\circledast) :: s \to v \to v$, is inhomogenous: the first argument is a scalar and the second one is a vector. For our representation it can be defined as follows:

$$\begin{aligned}&\textbf{infixr } 7 \circledast\\ &(\circledast) :: \mathit{Multiplicative}\ s \Rightarrow s \to \mathit{Vector}\ s\ g \to \mathit{Vector}\ s\ g\\ &s \circledast V\ a = V\ \$\ \lambda i \to s * (a\ i)\end{aligned}$$

The canonical basis vectors are given by

$$\begin{aligned}&e :: (\mathit{Eq}\ g, \mathit{Ring}\ s) \Rightarrow g \to \mathit{Vector}\ s\ g\\ &e\ i = V\ (\lambda j \to i\ \text{`}\mathit{is}\text{`}\ j)\end{aligned}$$

In linear algebra textbooks, the function *is* is often referred to as the Kronecker-delta function and *is i j* is written $\delta_{i,j}$.

$$\begin{aligned}&\mathit{is} :: (\mathit{Eq}\ g, \mathit{Ring}\ s) \Rightarrow g \to g \to s\\ &\mathit{is}\ i\ j = \textbf{if } i == j \textbf{ then } 1 \textbf{ else } 0\end{aligned}$$

It is 1 if its arguments are equal and 0 otherwise. Thus *e i* has zeros everywhere, except at position j where it has a 1.

This way, every $v : G \to S$ is a linear combination of vectors *e i*:

$$v = v\ 0 \circledast e\ 0 + \ldots + v\ n \circledast e\ n$$

7.2 Linear transformations

As we have seen in earlier chapters, morphisms between structures are often important. Vector spaces are no different: if we have two vector spaces *Vector S G* and *Vector S G′* for the same set of scalars *S*, we can study functions $f : \mathit{Vector}\ S\ G \to \mathit{Vector}\ S\ G'$:

$$f\ v = f\ (v\ 0 \circledast e\ 0 + ... + v\ n \circledast e\ n)$$

It is particularly interesting to study vector-space homomorphisms, which are more commonly called "linear maps" (to avoid unnecessary confusion with the Haskell *map* function we will refer to them by the slightly less common name "linear transformation".). The function f is a linear transformation if it maps the operations in *Vector S G* into operations in *Vector S G′* as follows:

$$\begin{aligned} f\ (u + v) &= f\ u + f\ v \\ f\ (s \circledast u) &= s \circledast f\ u \end{aligned}$$

Since $v = (v\ 0 \circledast e\ 0 + ... + v\ n \circledast e\ n)$, we also have:

$$f\ v = f\ (v\ 0 \circledast e\ 0 + ... + v\ n \circledast e\ n) = v\ 0 \circledast f\ (e\ 0) + ... + v\ n \circledast f\ (e\ n)$$

But this means that we can determine the values of $f : \mathit{Vector}\ S\ G \to \mathit{Vector}\ S\ G'$ from just the values of $f \circ e : G \to \mathit{Vector}\ S\ G'$, which has a much smaller domain. Let $m = f \circ e$. Then

$$f\ v = v\ 0 \circledast m\ 0 + ... + v\ n \circledast m\ n$$

Each of the $m\ k$ is a *Vector S G′*, as is the resulting $f\ v$. We have

$$\begin{aligned} &f\ v\ g' && = \{\text{- as above -}\} \\ &(v\ 0 \circledast m\ 0 + ... + v\ n \circledast m\ n)\ g' && = \{\text{- Def. of } (\circledast) \text{ and } (+) \text{ -}\} \\ &v\ 0 * m\ 0\ g' + ... + v\ n * m\ n\ g' && = \{\text{- using } \mathit{sum}, \text{ and } (*) \text{ commutative -}\} \\ &\mathit{sum}\ [m\ j\ g' * v\ j \mid j \leftarrow [0\,..\,n]] \end{aligned}$$

That is, it suffices to know the behaviour of f on the basis vectors to know its behaviour on the whole vector space.

It is enlightening to compare the above sum with the standard vector-matrix multiplication. Let us define M as follows:

$$M = [m\ 0 \mid ... \mid m\ n] \quad \text{-- where } m : G \to \mathit{Vector}\ S\ G'$$

That is, the columns of M are the images of the canonical base vectors $e\ i$ through f (or, in other words, the columns of M are $f\ (e\ i)$). Every $m\ k$ has *card G′* elements, and it has become standard to write $M\ i\ j$ to mean the ith element of the jth column, i.e., $M\ i\ j = m\ j\ i$, so that, with the usual matrix-vector multiplication

$$(M * v)\ i = sum\ [M\ i\ j * v\ j \mid j \leftarrow [0\,..\,n]]$$

therefore, one has

$$\begin{array}{lll} (M * v)\ i & = & \text{-- by def. of matrix-vector multiplication} \\ sum\ [M\ i\ j * v\ j \mid j \leftarrow [0\,..\,n]] & = & \text{-- by def. of M i j} \\ sum\ [m\ j\ i * v\ j \mid j \leftarrow [0\,..\,n]] & = & \text{-- by } f\ v\ g' = sum\ [m\ j\ g' * v\ j \mid j \leftarrow [0\,..\,n]] \text{ with } g' = i \\ f\ v\ i & & \end{array}$$

If we take *Matrix* to be just a synonym for functions of type $G \to Vector\ S\ G'$:

$$\textbf{type}\ Matrix\ s\ g\ g' = g' \to Vector\ s\ g$$

then we can implement matrix-vector multiplication as follows:

$$\begin{array}{l} mulMV :: (Finite\ g, Ring\ s) \Rightarrow Matrix\ s\ g\ g' \to Vector\ s\ g \to Vector\ s\ g' \\ mulMV\ m\ v = V\ (\lambda i \to sum\ [m\ i\,!\,j * v\,!\,j \mid j \leftarrow finiteDomain]) \end{array}$$

Note that in the terminology of the earlier chapters we can see *Matrix s g g'* as a type of syntax and the linear transformation (of type *Vector S G* $\to$ *Vector S G'*) as semantics. With this view, *mulMV* is just another *eval* :: *Syntax* $\to$ *Semantics*.

Example Consider the multiplication of a matrix with a basis vector:

$$(M * e\ k)\ i = sum\ [M\ i\ j * e\ k\ j \mid j \leftarrow [0\,..\,n]] = sum\ [M\ i\ k] = M\ i\ k$$

i.e., $e\ k$ extracts the kth column from M (hence the notation "e" for "extract").

We have seen how a linear transformation f can be fully described by a matrix of scalars, M. Similarly, in the opposite direction, given an arbitrary matrix M, we can define

$$f\ v = M * v$$

and obtain a linear transformation $f = (M*)$. Moreover $((M*) \circ e)\ g\ g' = M\ g'\ g$, i.e., the matrix constructed as above for f is precisely M.

Exercise 7.6: compute $((M*) \circ e)\ g\ g'$.

Therefore, every linear transformation is of the form $(M*)$ and every $(M*)$ is a linear transformation. There is a bijection between these two sets. Matrix-matrix multiplication is defined in order to ensure associativity (note here the overloading of the operator $*$):

$$(M' * M) * v = M' * (M * v)$$

that is

$$((M' * M)*) = (M'*) \circ (M*)$$

Exercise 7.7: work this out in detail.

TODO: JP: Note that *Matrix* form a category with mulMV being the composition and e as the identity.

Exercise 7.8: show that matrix-matrix multiplication is associative.

TODO: JP: Not sure what the difference is from the previous exercise.

A simple vector space is obtained for $G = ()$, the singleton index set. In this case, the vectors $s : () \to S$ are functions that can take exactly one value as argument, therefore they have exactly one value: $s\ ()$, so they are isomorphic with S. But, for any $v : G \to S$, we have a function $fv : G \to (() \to S)$, namely

$$fv\ g\ () = v\ g$$

fv is similar to our m function above. The associated matrix is

$$M = [m\ 0 \mid ... \mid m\ n] = [fv\ 0 \mid ... \mid fv\ n]$$

having $n + 1$ columns (the dimension of *Vector G*) and one row (dimension of *Vector* $()$). Let w :: *Vector S G*:

$$M * w = w\ 0 \circledast fv\ 0 + ... + w\ n \circledast fv\ n$$

$M * v$ and each of the $fv\ k$ are "almost scalars": functions of type $() \to S$, thus, the only component of $M * w$ is

$$(M * w)\ () = w\ 0 * fv\ 0\ () + ... + w\ n * fv\ n\ () = w\ 0 * v\ 0 + ... + w\ n * v\ n$$

i.e., the scalar product of the vectors v and w.

Remark: We have not discussed the geometrical point of view.

7.3 Dot products

An important concept is the dot product between vectors.

$$dot :: (Ring\ s, Finite\ g) \Rightarrow Vector\ s\ g \to Vector\ s\ g \to s$$
$$dot\ v\ w = sum\ [v\,!\,i * w\,!\,i \mid i \leftarrow finiteDomain]$$

Dot products have (at least) two aspects. First, they yield a notion of how "big" a vector is, the *norm*.

TODO: JP: If we take the algebraic view then this definition is only correct if the canonical basis is orthonormal.

$$sqNorm :: (Ring\ s, Finite\ g) \Rightarrow Vector\ s\ g \to s$$
$$sqNorm\ v = dot\ v\ v$$
$$norm\ v = \sqrt{sqNorm\ v}$$

Additionally, the dot product often serves as a measure of how much vectors are similar to (or correlated with) each other.

For two non-zero vectors u and v, we can define:

$$similarity\ u\ v = dot\ u\ v\ /\ norm\ u\ /\ norm\ v$$

Dividing by the norms mean that $abs\ (similarity\ u\ v)$ is at most 1 — in the [-1,1] interval.

TODO: JP: For real fields. For complex ones one would use the inner product instead.

In fact, for Euclidean spaces $similarity\ u\ v$ is the cosine of the angle between u and v.

For this reason, one says that two vectors are orthogonal when their dot product is 0 — even in non-Euclidean spaces.

Orthogonal transformations An important subclass of the linear transformations are those which preserve the dot product.

$$dot\ (f\ u)\ (f\ v) = dot\ u\ v$$

In Euclidean spaces, such a transformation preserve angles. In general, they are called orthogonal transformations.

Exercise 7.1. Can you express this condition as a homomorphism condition?

Such transformations necessarily preserve the dimension of the space (otherwise at least one base vector would be squished to nothing and dot products involving it become zero). (When the dimension is preseved, one often uses the term "linear operator".) The corresponding matrices are square.

Exercise 7.2. Prove that orthogonal operators form a monoid.

If angles are preserved what about distances? An isometry f is a distance-preserving transformation:

$$norm\ (f\ v) = norm\ v$$

We can prove that f is orthogonal iff. it is an isometry. The proof in the left-to-right direction is easy and left as an exercise. In the other direction one uses the equality:

$$4 * dot\ u\ v = sqNorm\ (u + v) - sqNorm\ (u - v)$$

In Euclidean spaces, this means that preserving angles and preserving distances go hand-in-hand.

Orthogonal transformations enjoy many more useful properties: we have barely scrached the surface here. Among others, their rows (and columns) are orthogonal to each other. The are also invertible (and so they form a group), and the inverse is the (conjugate-) transpose of the matrix. (In the context of a complex scalar field, one would use the word "unitary" instead of "orthogonal", but it's a straightforward generalisation.)

7.4 Examples of matrix algebra

7.4.1 Polynomials and their derivatives

We have represented polynomials of degree $n+1$ by the list of their coefficients. This is the same representation as the vectors represented by $n+1$ coordinates which we referred to in the introduction to this chapter. This suggests that polynomials of degree n form a vector space, and we could interpret that as $\{0,...,n\} \to \mathbb{R}$ (or, more generally, *Field a* $\Rightarrow \{0,...,n\} \to a$). The operations, $(+)$ for vector addition and $(\circledast)$ for vector scaling, are defined in the same way as they are for functions.

To give an intuition for the vector space it is useful to consider the interpretation of the canonical base vectors. Recall that they are:

$$e\ i : \{0,...,n\} \to \mathbb{R}, e\ i\ j = i\ `is`\ j$$

but how do we interpret them as polynomial functions?

When we represented a polynomial by its list of coefficients, we saw that the polynomial function $\lambda x \to x\hat{}3$ could be represented as $[0,0,0,1]$, where 1 is the coefficient of $x\hat{}3$. Similarly, representing this list of coefficients as a vector (a function from $\{0,...,n\} \to \mathbb{R}$), we get the vector $\lambda j \to$ **if** $j == 3$ **then** 1 **else** 0, which is $\lambda j \to 3\ `is`\ j$ or simply $e\ 3$.

In general, $\lambda x \to x\hat{}i$ is represented by $e\ i$, which is another way of saying that $e\ i$ should be interpreted as $\lambda x \to x\hat{}i$, a monomial. Any other polynomial function p equals the linear combination of monomials, and can therefore be represented as a linear combination of our base vectors $e\ i$. For example, $p\ x = 2 + x\hat{}3$ is represented by $2 \circledast e\ 0 + e\ 3$.

In general, the evaluator from the *Vector g s* representation to polynomial functions is as follows:

$$\begin{aligned} &evalP :: Vector\ \mathbb{R}\ \{0,...,n\} \to (\mathbb{R} \to \mathbb{R}) \\ &evalP\ (V\ v)\ x = sum\ (map\ (\lambda i \to v\ i * x\hat{}i)\ [0\,..\,n]) \end{aligned}$$

The *derive* function takes polynomials of degree $n+1$ to polynomials of degree n, and since $D\ (f+g) = D\ f + D\ g$ and $D\ (s \circledast f) = s \circledast D\ f$, we know that it is a linear transformation.

The associated matrix will be obtained by appling the linear transformation to every base vector:

$$M = [derive\ (e\ 0), derive\ (e\ 1), ..., derive\ (e\ n)]$$

where each *derive* $(e\ i)$ has length n. The vector $e\ (i+1)$ represents $\lambda x \to x\hat{}(i+1)$ and thus we want *derive* $(e\ (i+1))$ to represent the derivative of $\lambda x \to x\hat{}(i+1)$:

$$
\begin{array}{ll}
evalP\ (derive\ (e\ (i+1))) & = \text{\{- by spec. -\}} \\
D\ (evalP\ (e\ (i+1))) & = \text{\{- by def. of } e, evalP \text{ -\}} \\
D\ (\lambda x \to x\hat{}(i+1)) & = \text{\{- properties of } D \text{ from lecture 3 -\}} \\
\lambda x \to (i+1) * x\hat{}i & = \text{\{- by def. of } e, evalP, (\circledast) \text{ -\}} \\
evalP\ ((i+1) \circledast (e\ i)) &
\end{array}
$$

Thus

$$derive\ (e\ (i+1)) = (i+1) \circledast (e\ i)$$

Also, the derivative of $evalP\ (e\ 0) = \lambda x \to 1$ is $\lambda x \to 0$ and thus $derive\ (e\ 0)$ is the zero vector:

$$derive\ (e\ 0) = 0$$

Example: $n + 1 = 3$:

$$M = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

Take the polynomial function $p\ x = 1 + 2 * x + 3 * x\hat{}2$ as a vector

$$v = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

and we have

$$M * v = \begin{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} & \begin{bmatrix} 1 \\ 0 \end{bmatrix} & \begin{bmatrix} 0 \\ 2 \end{bmatrix} \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 2 \\ 6 \end{bmatrix}$$

representing the polynomial function $p'\ x = 2 + 6 * x$.

Exercise 7.9: write the (infinite-dimensional) matrix representing D for power series.

Exercise 7.10: write the matrix In associated with integration of polynomials.

7.4.2 $*$Dot product for functions and Fourier series

We said before that the dot product yields a notion of norm and similarity. Unfortunately, the dot product (as defined above) is not very useful in this respect for polynomials represented as monomial coefficients: it is not clear what kind of similarity it corresponds to. To find a more useful dot product, we can return to the semantics of polynomials in terms of functions. But for now we consider them over the restricted domain $I = [-\pi, \pi]$.

Assume for a moment that we would define the dot product of functions u and v as follows:

$$dotF\ u\ v = \int_I eval\ ux eval\ vx dx$$

Then, the norm of a function would be a measure of how far it gets from zero, using a quadratic mean. Likewise, the corresponding similarity measure corresponds to how much the functions "agree" on the interval. That is, if the signs of *eval u* and *eval v* are the same on a sub-interval I then the integral is positive on I, and negative if they are different.

As we suspected, using $dot = dotF$, the straightforward representation of polynomials as list of coefficients is not an orthogonal basis. There is, for example, a positive correlation between x and $x\hat{}3$.

If we were using instead a set of basis polynomials b_n which are orthogonal using the above semantic-oriented definition of *dot*, then we could compute it by multiplying pointwise and summing as before, and this would be a lot more efficient than to compute the integral by 1. computing the product using *polyMul* 2. integrating using *integ*. 3. using *eval* on the end points of the domain.

Let us consider as a base the functions $b_n = sin\ (n * x)$, prove that they are orthogonal.

We first use trigonometry to rewrite the product of bases:

$$\begin{aligned} & 2 * (b_i * bj) \\ = & \ 2 * sin\ (i * x) * sin\ (j * x) \\ = & \ cos\ ((i - j) * x) - cos\ ((i + j) * x) \end{aligned}$$

Assuming $i \neq j$, we can take the indefinite integral of both sides, safely ignoring any constant term:

$$\begin{aligned} & 2\lambda int\ b_i\ bj\ dx \\ = & \ sin\ ((i - j) * x)\ /\ (i - j) - sin\ ((i + j) * x)\ /\ (i + j) \end{aligned}$$

But $sin\ (k * \pi) = 0$ for any integer k, and thus the definite integral over I is also equal to zero.

We can now compute *sqNorm* of b_i. Trigonometry says:

$$\begin{aligned} & 2 * (b_i * b_i) \\ = & \ 2 * sin\ (i * x) * sin\ (i * x) \\ = & \ cos\ (0 * x) - cos\ (2\ i * x) \\ = & \ 1 - cos\ (2\ i * x) \end{aligned}$$

When taking the integral on I, the cosine disappears using the same argument as before, and there remains: $2 * sqNorm\ b_i = 2\ \pi$. Thus to normalise the base vectors we need to scale them by $1\ /\ \sqrt{\pi}$. In sum $b\_i = sin\ (i * x)\ /\ \sqrt{\pi}$ is an orthonormal basis:

$$bj\ `dot`\ b_i = is\ i\ j$$

As interesting as it is, this basis does not cover all functions over I. To start, $eval\ b_i\ 0 == 0$ for every i, and thus linear combinations can only ever be zero at the origin.

But if we were to include $cos\ (n * x)\ /\ \sqrt{\pi}$ in the set of base vectors, then the space would cover all periodic functions with period 2π. This representation is called the Fourier series. Let us define a meaningful index (G) for the basis:

```
data Periodic where
  Sin :: Int → Periodic
  Cos :: Int → Periodic
```

A useful property of an orthonormal basis is that its representation as coefficients can be obtained by taking the dot product with each base vectors. Indeed:[2]

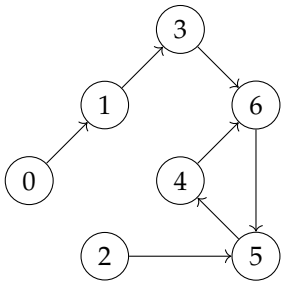
$$\begin{aligned} & v & = sum\ [v_i \circledast b_i \mid i \leftarrow finiteDomain] \\ \Rightarrow\ & v\ `dot`\ bj & = sum\ [v_i \circledast b_i \mid i \leftarrow finiteDomain]\ `dot`\ bj \\ \Rightarrow\ & v\ `dot`\ bj & = sum\ [v_i \circledast (b_i\ `dot`\ bj) \mid i \leftarrow finiteDomain] \\ \Rightarrow\ & v\ `dot`\ bj & = sum\ [v_i \circledast is\ i\ j \mid i \leftarrow finiteDomain] \\ \Rightarrow\ & v\ `dot`\ bj & = vj \end{aligned}$$

Thus, in our application, given a periodic function f, one can compute its Fourier series by taking the *dotF* product of it with each of $sin\ (n * x)\ /\ \sqrt{\pi}$ and $cos\ (n * x)\ /\ \sqrt{\pi}$.

Exercise 7.3. Derive *derive* for this representation.

7.4.3 Simple deterministic systems (transition systems)

Simple deterministic systems are given by endo-functions[3] on a finite set $next : G \to G$. They can often be conveniently represented as a graph, for example



[2] The proof can be easily adapted to infinite sums.

[3] An *endo-function* is a function from a set X to itself: $f : X \to X$.

Here, $G = \{0, ..., 6\}$. A node in the graph represents a state. A transition $i \to j$ means *next* $i = j$. Since *next* is an endo-function, every node must be the source of exactly one arrow.

We can take as vectors the characteristic functions of subsets of G, i.e., $G \to \{0, 1\}$. $\{0, 1\}$ is not a field w.r.t. the standard arithmetical operations (it is not even closed w.r.t. addition), and the standard trick to workaround this issue is to extend the type of the functions to $\mathbb{R}$.

The canonical basis vectors are, as usual, $e\ i = V\ (is\ i)$. Each $e\ i$ is the characteristic function of a singleton set, $\{i\}$.

We can interpret $e\ (next\ 0), ..., e\ (next\ 6)$ as the images of the basis vectors $e\ 0, ..., e\ 6$ of *Vector* $\mathbb{R}\ G$ under the transformation

$$f : Vector\ \mathbb{R}\ G \to Vector\ \mathbb{R}\ G$$
$$f\ (e\ i) = e\ (next\ i)$$

To write the matrix associated to f, we have to compute what vector is associated to each canonical base vector vector:

$$M = [f\ (e\ 0), f\ (e\ 1), ..., f\ (e\ n)]$$

Therefore:

$$M = \begin{array}{c} \\ r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \end{array} \begin{array}{c} \begin{array}{ccccccc} c_0 & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Notice that row 0 and row 2 contain only zero, as one would expect from the graph of *next*: no matter where we start from, the system will never reach node 0 or node 2.

Starting with a canonical base vector $e\ i$, we obtain $M * e\ i = f\ (e\ i)$, as we would expect. The more interesting thing is if we start with something different from a basis vector, say $[0, 0, 1, 0, 1, 0, 0] == e\ 2 + e\ 4$. We obtain $\{f\ 2, f\ 4\} = \{5, 6\}$, the image of $\{2, 4\}$ through f. In a sense, we can say that the two transitions happened in parallel. But that is not quite accurate: if start with $\{3, 4\}$, we no longer get the characteristic function of $\{f\ 3, f\ 4\} = \{6\}$, instead, we get a vector that does not represent a characteristic function at all: $[0, 0, 0, 0, 0, 0, 2] = 2 \circledast e\ 6$. In general, if we start with an arbitrary vector, we can interpret this as starting with various quantities of some unspecified material in each state, simultaneously. If f were injective, the respective quantities would just get shifted around, but in our case, we get a more general behaviour.

What if we do want to obtain the characteristic function of the image of a subset? In that case, we need to use other operations than the standard arithmetical ones, for example *min* and *max*.

However, $(\{0,1\}, max, min)$ is not a field, and neither is $(\mathbb{R}, max, min)$. This means that we do not have a vector space, but rather a *module*. One can still do a lot with modules: for example the definition of multiplication only demands a *Ring* rather than a *Field*. Therefore, having just a module is not a problem if all we want is to compute the evolutions of possible states, but we cannot apply most of the deeper results of linear algebra.

TODO: JP: But perhaps we have not defined those so far... Here it begs to say that the behaviour at the limit is given by the inverted matrix. But where does it fit in the chapter?

In the example above, we have:

```
newtype G = G Int deriving (Eq, Show, Additive, Multiplicative, AddGroup)
instance Bounded G where
  minBound = G 0
  maxBound = G 6
instance Enum G where
  toEnum           = G
  fromEnum (G n) = n
```

Note that the *Ring G* instance is given just for convenient notation (integer literals): vector spaces in general do not rely on any numeric structure on the indices (G). The transition function has type $G \to G$ and the following implementation:

```
next1 :: G → G
next1 0 = 1; next1 1 = 3; next1 2 = 5; next1 3 = 6; next1 4 = 6; next1 5 = 4; next1 6 = 5
```

Its associated matrix is

```
m g'
  = {- m is the matrix associated with f -}
V (λg → toF (f (e g)) g')
  = {- by the spec. of f -}
V (λg → toF (e (next g)) g')
  = {- by def. of e -}
V (λg → toF (V (is (next g))) g')
  = {- by def. of toF -}
V (λg → is (next g) g')
```

where

```
toF :: Vector s g → g → s
toF (V v) = v
```

Thus we can implement m as:

TODO: JP: This is the same as indexing (!)

```
m1 :: Ring s ⇒ G → Vector s G
m1 g' = V (λg → (next1 g) `is` g')
```

Test:

$$
\begin{aligned}
&t1' :: \mathit{Vector\ Int\ G}\\
&t1' = \mathit{mulMV}\ m_1\ (e\ 3 + e\ 4)\\
&t_1\ \ = \mathit{toL}\ t1' \quad \text{-- } [0,0,0,0,0,0,2]
\end{aligned}
$$

TODO: JP: Fold this implementation in the text

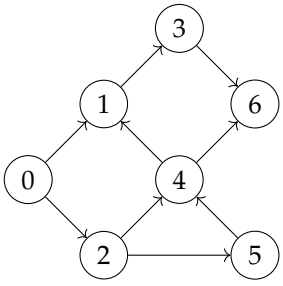
7.4.4 Non-deterministic systems

Another interpretation of the application of M to characteristic functions of a subset is the following: assuming that all I know is that the system is in one of the states of the subset, where can it end up after one step? (this assumes the *max-min* algebra as above).

The general idea for non-deterministic systems, is that the result of applying the step function a number of times from a given starting state is a list of the possible states one could end up in.

In this case, the uncertainty is entirely caused by the fact that we do not know the exact initial state. However, there are cases in which the output of f is not known, even when the input is known. Such situations are modelled by endo-relations: $R : G \to G$, with $g\ R\ g'$ if g' is a potential successor of g. Endo-relations can also be pictured as graphs, but the restriction that every node should be the source of exactly one arrow is lifted. Every node can be the source of one, none, or many arrows.

For example:



Now, starting in 0 we might end up either in 1 or 2 (but not both!). Starting in 6, the system breaks down: there is no successor state.

The matrix associated to R is built in the same fashion: we need to determine what vectors the canonical base vectors are associated with:

$$M = \begin{array}{c} \\ r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \end{array} \begin{array}{c} \begin{array}{ccccccc} c_0 & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Exercise 7.11: start with $e\ 2 + e\ 3$ and iterate a number of times, to get a feeling for the possible evolutions. What do you notice? What is the largest number of steps you can make before the result is the zero vector? Now invert the arrow from 2 to 4 and repeat the exercise. What changes? Can you prove it?

Implementation:

The transition relation has type $G \to (G \to \mathit{Bool})$:

$$\begin{aligned} &f_2 :: G \to (G \to \mathit{Bool}) \\ &f_2\ 0\ g = g == 1 \vee g == 2 \\ &f_2\ 1\ g = g == 3 \\ &f_2\ 2\ g = g == 4 \vee g == 5 \\ &f_2\ 3\ g = g == 6 \\ &f_2\ 4\ g = g == 1 \vee g == 6 \\ &f_2\ 5\ g = g == 4 \\ &f_2\ 6\ g = \mathit{False} \end{aligned}$$

The associated matrix:

$$m_2\ g' = V\ (\lambda g \to f_2\ g\ g')$$

We need a *Ring* instance for *Bool* (not a field!):

$$\begin{aligned} &\textbf{instance}\ \mathit{Additive}\ \mathit{Bool}\ \textbf{where} \\ &\quad (+) = (\vee) \\ &\quad \mathit{zero} = \mathit{False} \\ &\textbf{instance}\ \mathit{AddGroup}\ \mathit{Bool}\ \textbf{where} \\ &\quad \mathit{negate} = \neg \\ &\textbf{instance}\ \mathit{Multiplicative}\ \mathit{Bool}\ \textbf{where} \\ &\quad \mathit{one} = \mathit{True} \\ &\quad (*)\ = (\wedge) \end{aligned}$$

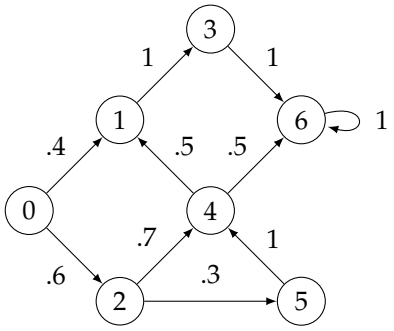
Test:

$$\begin{aligned} &t2' = \mathit{mulMV}\ m_2\ (e\ 3 + e\ 4) \\ &t_2 = \mathit{toL}\ t2' \quad \text{-- } [\mathit{False}, \mathit{True}, \mathit{False}, \mathit{False}, \mathit{False}, \mathit{False}, \mathit{True}] \end{aligned}$$

TODO: JP: Fold in the text

7.4.5 Stochastic systems

Quite often, we have more information about the transition to possible future states. In particular, we can have *probabilities* of these transitions. For example



One could say that this case is a generalisation of the previous one, in which we can take all probabilities to be equally distributed among the various possibilities. While this is plausible, it is not entirely correct. For example, we have to introduce a transition from state 6 above. The nodes must be sources of *at least* one arrow.

In the case of the non-deterministic example, the "legitimate" inputs were characteristic functions, i.e., the "vector space" was $G \to \{0,1\}$ (the scare quotes are necessary because, as discussed, the target is not a field). In the case of stochastic systems, the inputs will be *probability distributions* over G, that is, functions $p : G \to [0,1]$ with the property that

$$sum\ [p\ g \mid g \leftarrow G] = 1$$

If we know the current probability distributions over states, then we can compute the next one by using the *total probability formula*, normally expressed as

$$p\ a = sum\ [p\ (a \mid b) * p\ b \mid b \leftarrow G]$$

This formula in itself is worth a lecture (see Chapter 9). But for now, let's just remark that the notation is extremely suspicious. $(a \mid b)$, which is usually read "a, given b", is clearly not of the same type as a or b, so cannot really be an argument to p. We discuss this notation at length in Chapter 9. Additionally, the $p\ a$ we are computing with this formula is not the $p\ a$ which must eventually appear in the products on the right hand side. We do not know how this notation came about: it is neither in Bayes' memoir, nor in Kolmogorov's monograph.

TODO: JP: According to Jaynes in "Probability Theory, The Logic of Science, p. 17. The notation A | B for "A given B" is attributable to Keynes (1921)."

TODO: JP: citations needed

Regardless, at this stage, what we need to know is that the conditional probability $p\ (a \mid b)$ gives us the probability that the next state is a, given that the current state is b. But this is exactly the information summarised in the graphical representation. Moreover, it is clear that, at least formally, the total probability formula is identical to a matrix-vector multiplication.

As usual, we write the associated matrix by looking at how the canonical base vectors are transformed. In this case, the canonical base vector $e\ i = \lambda j \to i\text{`}is\text{`}\ j$ is the probability distribution *concentrated* in i. This means that the probability to be in state i is 100% and the probability of being anwhere else is 0.

$$M = \begin{array}{c} \\ r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \end{array} \begin{array}{c} \begin{array}{ccccccc} c_0 & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ .4 & 0 & 0 & 0 & .5 & 0 & 0 \\ .6 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & .7 & 0 & 0 & 1 & 0 \\ 0 & 0 & .3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & .5 & 0 & 1 \end{pmatrix} \end{array}$$

Exercise 7.12: starting from state 0, how many steps do you need to take before the probability is concentrated in state 6? Reverse again the arrow from 2 to 4. What can you say about the long-term behaviour of the system now?

TODO: JP: what does that mean how do we make it so that columns sum to 1? Also NiBo:There are many ways of reversing the arrow from 2 to 4.

Exercise 7.13: Implement the example. You will need to define:

The transition function (giving the probability of getting to g' from g)

$f_3 :: G \to \mathit{Vector}\ \mathbb{R}\ G$ -- but we want only $G \to \mathit{Vector}\ [0,1]\ G$, the unit interval

and the associated matrix

$m_3 :: G \to \mathit{Vector}\ \mathbb{R}\ G$

7.4.6 ∗Quantum Systems

Instead of real numbers for probabilities, we could consider using complex numbers — one then speaks of "amplitudes". An amplitude represented by a complex number z is converted to a probability by taking the square of the modulus of z:

$$p\ z = \mathit{conj}\ z * z$$

We can then rewrite the law of total probability as follows:

$$\begin{aligned} & \mathit{sum}\ [p\ !\ i \mid i \leftarrow \mathit{finiteDomain}] \\ = {} & \mathit{sum}\ [\mathit{conj}\ (z\ !\ i) * (z\ !\ i) \mid i \leftarrow \mathit{finiteDomain}] \\ = {} & \mathit{inner}\ z\ z \end{aligned}$$

Where the *inner* product generalises the dot product for vector spaces with complex scalars (the left operand is conjugated indexwise). Hence, rather conveniently, the law of total probability is replaced by conservation of the norm

of state vectors. In particular, norms are conserved if the transition matrix is unitary.

The unitary character of the transition matrix defines valid systems from the point of view of quantum mechanics. Because all unitary matrices are invertible, it follows that all quantum mechanical systems have an invertible dynamics. Furthermore, the inverted matrix is also unitary, and therefore the inverted system is also valid as a quantum dynamical system.

TODO: JP: An example as a graph is daunting. One can construct an example by combining generator matrices $e^{i\theta}$ (one coordinate rotates in complex plane), and 2-d rotations (the "cos theta/sin thetha/-sin theta/cos theta" matrix). But there wont be "mostly zeros everywhere". The form $U = e^{iH}$ is also a possibility, but then there is a whole bunch of more theory to cover.

7.5 Monadic dynamical systems

This section is not part of the intended learning outcomes of the course, but it presents a useful unified view of the previous sections which could help your understanding.

All the examples of dynamical systems we have seen in the previous section have a similar structure. They work by taking a state (which is one of the generators) and return a structure of possible future states of type G:

- deterministic: there is exactly one possible future state: we take an element of G and return an element of G. The transition function has the type $f : G \to G$, the structure of the target is just G itself.
- non-deterministic: there is a set of possible future states, which we have implemented as a characteristic function $G \to \{0, 1\}$. The transition function has the type $f : G \to (G \to \{0, 1\})$. The structure of the target is the *powerset* of G.
- stochastic: given a state, we compute a probability distribution over possible future states. The transition function has the type $f : G \to (G \to \mathit{Complex})$, the structure of the target is the probability distributions over G.
- quantum: given an observable state, we compute an (orthogonal) superposition of possible future states.

Therefore:

- deterministic: $f : G \to \mathit{Id}\ G$
- non-deterministic: $f : G \to \mathit{Powerset}\ G$, where $\mathit{Powerset}\ G = G \to \{0, 1\}$
- stochastic: $f : G \to \mathit{Prob}\ G$, where $\mathit{Prob}\ G = G \to [0, 1]$
- quantum: $f : G \to \mathit{Super}\ G$, where $\mathit{Super}\ G = G \to \mathit{Complex}$

We have represented the elements of the various structures as vectors. We also had a way of representing, as structures of possible states, those states that were known precisely: these were the canonical base vectors $e\ i$. Due to the nature of matrix-vector multiplication, what we have done was in effect:

$$
\begin{aligned}
& M * v \quad \text{-- } v \text{ represents the current possible states} \\
= & \ \{\text{- } v \text{ is a linear combination of the base vectors -}\} \\
& M * (v\ 0 \circledast e\ 0 + ... + v\ n \circledast e\ n) \\
= & \ \{\text{- homomorphism -}\} \\
& v\ 0 \circledast (M * e\ 0) + ... + v\ n \circledast (M * e\ n) \\
= & \ \{\text{- } e\ i \text{ represents the perfectly known current state } i, \text{ therefore } M * e\ i = f\ i \text{ -}\} \\
& v\ 0 \circledast f\ 0 + ... + v\ n \circledast f\ n
\end{aligned}
$$

So, we apply f to every state, as if we were starting from precisely that state, obtaining the possible future states starting from that state, and then collect all these hypothetical possible future states in some way that takes into account the initial uncertainty (represented by $v\ 0$, ..., $v\ n$) and the nature of the uncertainty (the specific $(+)$ and $(\circledast)$).

If you examine the types of the operations involved

$$e : G \to \mathit{Possible}\ G$$

and

$$\mathit{flip}\ (*) : \mathit{Possible}\ G \to (G \to \mathit{Possible}\ G) \to \mathit{Possible}\ G$$

you see that they are very similar to the monadic operations

TODO: JP: We need a proper introduction to monads somewhere

$$
\begin{aligned}
& \mathit{return} : g \to m\ g \\
& (\gg\!=) : m\ g \to (g \to m\ g') \to m\ g'
\end{aligned}
$$

which suggests that the representation of possible future states might be monadic. Indeed, that is the case.

Since we implemented all these as matrix-vector multiplications, this raises the question: is there a monad underlying matrix-vector multiplication, such that the above are instances of it (obtained by specialising the scalar type S)?

Exercise: write *Monad* instances for *Id*, *Powerset*, *Prob*, *Supp*.

7.6 The monad of linear algebra

TODO: JP: Todo: connect this with the monad of probability theory chapter.

The answer is yes, up to a point. Haskell *Monad*s, just like *Functor*s, require *return* and $\gg\!=$ to be defined for every type. This will not work, in general. Our definition will work for *finite types* only.

```
class FinFunc f where
  func :: (Finite a, Finite b) ⇒ (a → b) → f a → f b
class FinMon f where
  embed :: Finite a ⇒ a → f a
  bind  :: (Finite a, Finite b) ⇒ f a → (a → f b) → f b
```

The idea is that vectors on finite types are finite functors and monads:

```
instance Ring s ⇒ FinFunc (Vector s) where
  func f (V v) = V (λg′ → sum [v g | g ← finiteDomain, g′ == f g])
instance Ring s ⇒ FinMon (Vector s) where
  embed g      = V (is g)
  bind (V v) f = V (λg′ → sum [toF (f g) g′ ∗ v g | g ← finiteDomain])
```

Note that, if $v :: Vector\ S\ G$ and $f :: G \to Vector\ S\ G'$ then both *func f v* and *bind v f* are of type *Vector S G′*. How do these operations relate to linear algebra and matrix-vector multiplication?

Remember that *e g* is that vector whose components are zero except for the *g*th one which is one. In other words

$$e\ g = V\ (is\ g) = embed\ g$$

and thus *embed* = *e*. In order to understand how matrix-vector multiplication relates to the monadic operations, remember that matrixes are just functions of type $G \to Vector\ S\ G'$:

```
type Matrix s g g′ = g′ → Vector s g
```

According to our earlier definition, we can rewrite matrix-vector multiplication in terms of dot products

```
  mulMV m v
= {- earlier definition -}
  V (λi → sum [(m i ! j) ∗ (v ! j) | j ← finiteDomain])
= {- def. of dot -}
  V (λi → dot (m i′) v)
```

Now, with

```
toMatrix :: (g → Vector s g′) → Matrix s g g′
toMatrix f = λg′ → V (λg → toF (f g) g′)
```

we have:

```
mulMV (toMatrix f) (V v)
```

$$
\begin{aligned}
&= \{\text{- def. of } \mathit{mulMV} \text{ -}\} \\
&\quad V\ (\lambda g' \to \mathit{dot}\ ((\mathit{toMatrix}\ f)\ g')\ (V\ v)) \\
&= \{\text{- def. of } \mathit{toMatrix} \text{ -}\} \\
&\quad V\ (\lambda g' \to \mathit{dot}\ (V\ (\lambda g \to \mathit{toF}\ (f\ g)\ g'))\ (V\ v)) \\
&= \{\text{- def. of } \mathit{dot} \text{ -}\} \\
&\quad V\ (\lambda g' \to \mathit{sum}\ [\mathit{toF}\ (f\ g)\ g' * v\ g \mid g \leftarrow \mathit{finiteDomain}]) \\
&= \{\text{- def. of } \mathit{bind} \text{ -}\} \\
&\quad \mathit{bind}\ (V\ v)\ f
\end{aligned}
$$

Thus we see that $\mathit{bind}\ v\ f$ is "just" a matrix-vector multiplication.

Perhaps for extra exercises: TODO: JP: clarify status

It is worth pointing out the role of f in $\mathit{func}\ f\ v$. We can rewrite the g'th component of $\mathit{func}\ f\ v$ in terms of the dot product

$$
\begin{aligned}
&\mathit{dot}\ v\ (V\ (\lambda g \to \mathit{is}\ g'\ (f\ g))) \\
&= \\
&\mathit{dot}\ v\ (V\ (\mathit{is}\ g' \circ f))
\end{aligned}
$$

This shows that the role of f in $\mathit{func}\ f\ v$ is that of re-distributing the values of v onto the new vector.

Exercise 7.4. show that if $w = \mathit{func}\ f\ v$ then the sum of the components of w is equal to the sum of the components of v.

Exercise 7.5. 1. Prove that the functor laws hold, i.e.

$$
\begin{aligned}
&\mathit{func}\ \mathit{id} &&= \mathit{id} \\
&\mathit{func}\ (g \circ f) &&= \mathit{func}\ g \circ \mathit{func}\ f
\end{aligned}
$$

2. Prove that the monad laws hold, i.e.

$$
\begin{aligned}
&\mathit{bind}\ v\ \mathit{return} &&= v \\
&\mathit{bind}\ (\mathit{return}\ g)\ f &&= f\ g \\
&\mathit{bind}\ (\mathit{bind}\ v\ f)\ h &&= \mathit{bind}\ v\ (\lambda g' \to \mathit{bind}\ (f\ g')\ h)
\end{aligned}
$$

3. What properties of S have you used to prove these properties? Define a new type class *GoodClass* that accounts for these (and only these) properties.

7.7 Associated code

Conversions and *Show* functions so that we can actually see our vectors.

```
toL :: Finite g ⇒ Vector s g → [s]
toL (V v) = map v finiteDomain
instance (Finite g, Show s) ⇒ Show (g → s)      where show = showFun
instance (Finite g, Show s) ⇒ Show (Vector s g) where show = showVector
showVector :: (Finite g, Show s) ⇒ Vector s g → String
showVector (V v) = showFun v
showFun :: (Finite a, Show b) ⇒ (a → b) → String
showFun f = show (map f finiteDomain)
```

The scalar product of two vectors is a good building block for matrix multiplication:

```
dot' :: (Finite g, Ring s) ⇒
  (g → s) → (g → s) → s
dot' v w = sum (map (v ∗ w) finiteDomain)
```

Note that $v * w :: g \to s$ is using the *FunNumInst*.

Using it we can shorten the definition of *mulMV*

```
  mulMV m v g'
=   -- Earlier definition
  sum [m g' g ∗ v g | g ← finiteDomain]
=   -- replace list comprehension with map
  sum (map (λg → m g' g ∗ v g) finiteDomain)
=   -- use FunNumInst for (∗)
  sum (map (m g' ∗ v) finiteDomain)
=   -- Def. of dot'
  dot' (m g') v
```

Thus, we can define matrix-vector multiplication by

```
mulMV m v g' = dot' (m g') v
```

We can even go one step further:

```
  mulMV m v
=   -- Def.
  λg' → dot' (m g') v
=   -- dot' is commutative
  λg' → dot' v (m g')
=   -- Def. of (∘)
  dot' v ∘ m
```

to end up at

```
mulMV' :: (Finite g, Ring s) ⇒
          Mat s g g' → Vec s g → Vec s g'
```

```
mulMV' m v = dot' v ∘ m
type Mat s r c = c → r → s
type Vec s r = r → s
```

Similarly, we can define matrix-matrix multiplication:

```
mulMM' :: (Finite b, Ring s) ⇒
          Mat s b c → Mat s a b → Mat s a c
mulMM' m1 m2 = λr c → mulMV' m1 (getCol m2 c) r
transpose :: Mat s g g' → Mat s g' g
transpose m i j = m j i
getCol :: Mat s g g' → g → Vec s g'
getCol = transpose
getRow :: Mat s g g' → g' → Vec s g
getRow = id
```

7.8 Exercises

Search the chapter for tasks marked "Exercise".

Exercise 7.6. Compute $((M*) \circ e)\ g\ g'$.

Exercise 7.7. Matrix-matrix multiplication is defined in order to ensure a homomorphism from $(*)$ to $(\circ)$.

$$\forall\, M.\ \forall\, M'.\ ((M' * M)*) == (M'*) \circ (M*)$$

or in other words

$$H_2\ ((*), (*), (\circ))$$

Work out the types and expand the definitions to verify that this claim holds. Note that one $(*)$ is matrix-vector multiplication and the other is matrix-matrix multiplication.

Exercise 7.8. Show that matrix-matrix multiplication is associative.

Exercise 7.9. With $G = \mathbb{N}$ for the set of indices, write the (infinite-dimensional) matrix representing D for power series.

Exercise 7.10. Write the matrix I_n associated with integration of polynomials of degree n.

Exercise 7.11. In the context of Section 7.4.4: start with $v_0 = e\ 2 + e\ 3$ and iterate $M*$ a number of times, to get a feeling for the possible evolutions. What do you notice? What is the largest number of steps you can make before the result is the origin vector (just zero)?

Now change M to M' by inverting the arrow from 2 to 4 and repeat the exercise. What changes? Can you prove it?

Exercise 7.12. In the context of the example matrix M in Section 7.4.5: starting from state 0, how many steps do you need to take before the probability is concentrated in state 6?

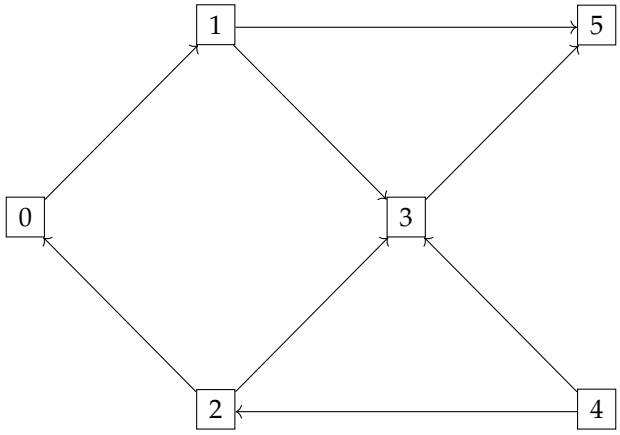
Now change M to M' by inverting the arrow from 2 to 4 and repeat the exercise. What can you say about the long-term behaviour of the system now?

Exercise 7.13. In the context of the example matrix M in Section 7.4.5: implement the example. You will need to define the transition function of type $G \to (G \to [0,1])$ returning the probability of getting from g to g', and the associated matrix.

7.8.1 Exercises from old exams

Exercise 7.14. *From exam 2017-03-14*

Consider a non-deterministic system with a transition function $f : G \to [G]$ (for $G = \{0 \,..\, 5\}$) represented in the following graph



The transition matrix can be given the type $m :: G \to (G \to \mathit{Bool})$ and the canonical vectors have type $e\ i :: G \to \mathit{Bool}$ for i in G.

1. (General questions.) What do the canonical vectors represent? What about non-canonical ones? What are the operations on *Bool* used in the matrix-vector multiplication?

2. (Specific questions.) Write the transition matrix m of the system. Compute, using matrix-vector multiplication, the result of three steps of the system starting in state 2.

Chapter 8

Exponentials and Laplace

8.1 The Exponential Function

```
{-# LANGUAGE RebindableSyntax #-}
module DSLsofMath.W08 where
import Prelude hiding (Num (..), (/), (^), Fractional (..), Floating (..), sum)
import DSLsofMath.W05
import DSLsofMath.W06
import DSLsofMath.Algebra
```

In one of the classical analysis textbooks, Rudin [1987] starts with a prologue on the exponential function. The first sentence is

> This is undoubtedly the most important function in mathematics.

Rudin goes on

> It is defined, for every complex number z, by the formula
>
> $exp\ z = \Sigma\ (z\hat{}n)\ /\ n\,!$

We, on the other hand, have defined the exponential function as the solution of a differential equation, which can be represented by a power series:

```
expx :: Fractional a ⇒ PowerSeries a
expx = integ 1 expx
```

and approximated by

$$expf :: Fractional\ a \Rightarrow a \to a$$
$$expf = eval\ 100\ expx$$

It is easy to see, using the definition of *integ* that the power series *expx* is, indeed

$$expx = [1, 1\ /\ 1, 1\ /\ (1*2), 1\ /\ (1*2*3), ..., 1\ /\ (1*2*3*...*n), ..]$$

We can compute the exponential for complex values if we can give an instance of *Fractional* for complex numbers. We could use the datatype *Data.Complex* from the Haskell standard library, but we prefer to roll our own in order to recall the basic operations on complex numbers.

TODO: JP: but it does not seem that this is done anyway?

As we saw in Chapter 1, complex values can be represented as pairs of real values.

$$\textbf{newtype}\ Complex\ r = C\ (r, r)\ \textbf{deriving}\ (Eq, Show)$$
$$i :: Ring\ a \Rightarrow Complex\ a$$
$$i = C\ (0, 1)$$

Therefore we can define, for example, the exponential of the imaginary unit:

$$ex1 :: Field\ a \Rightarrow Complex\ a$$
$$ex1 = expf\ i$$

And we have $ex1 = C\ (0.5403023058681398, 0.8414709848078965)$. Observe at the same time:

$$cosf\ 1 = 0.5403023058681398$$
$$sinf\ 1 = 0.8414709848078965$$

and therefore $expf\ i == C\ (cosf\ 1, sinf\ 1)$. This is not a coincidence, as we shall see.

Instead of evaluating the sum of the terms $a_n * z\hat{}n$, let us instead collect the terms in a series:

$$terms\ as\ z = terms1\ as\ z\ 0\ \textbf{where}$$
$$\quad terms1\ (Cons\ a\ as)\ z\ n = Cons\ (a * z\hat{}n)\ (terms1\ as\ z\ (n+1))$$

We obtain

TODO: JP: Why not use taylor series? It would be much easier to write and read. But first, properly include Taylor.hs in Chapter 6

$$ex2 ::\ Field\ a \Rightarrow PowerSeries\ (Complex\ a)$$
$$ex2 = takePoly\ 10\ (terms\ expx\ i)$$

As the code is polymorphic in the underlying number type, we can use rationals here to be able to test for equality without rounding problems.

$$
\begin{aligned}
&ex2' :: [Complex\ Rational]\\
&ex2' = [C\ (\ \ 1\ ,\ \ \ \ \ \ 0),\ \ \ \ C\ (0,\ \ \ 1\ \ \ ),\\
&\qquad C\ (-1\ /\ 2,\ \ \ \ 0),\ \ \ \ C\ (0,\ -1\ /\ 6),\\
&\qquad C\ (\ \ 1\ /\ 24,\ \ \ 0),\ \ \ \ C\ (0,\ \ \ 1\ /\ 120),\\
&\qquad C\ (-1\ /\ 720,\ \ 0),\ \ \ \ C\ (0,\ -1\ /\ 5040),\\
&\qquad C\ (\ \ 1\ /\ 40320, 0),\ \ \ \ C\ (0,\ \ \ 1\ /\ 362880)]\\
&check = toList\ ex2 == ex2'
\end{aligned}
$$

We can see that the real part of this series is the same as

$$
\begin{aligned}
&ex2R :: Poly\ Rational\\
&ex2R = takePoly\ 10\ (terms\ cosx\ 1)
\end{aligned}
$$

and the imaginary part is the same as

$$
\begin{aligned}
&ex2I :: Poly\ Rational\\
&ex2I = takePoly\ 10\ (terms\ sinx\ 1)
\end{aligned}
$$

But the terms of a series evaluated at 1 are the coefficients of the series. Therefore, the coefficients of *cosx* are

$$[1, 0, -1\ /\ 2!, 0, 1\ /\ 4!, 0, -1\ /\ 6!, ...]$$

In other words, the power series representation of the coefficients for *cos* is

$$
\begin{aligned}
&cosa\ (2 * n)\ \ \ \ \ \ = (-1)\hat{}n\ /\ (2 * n)\,!\\
&cosa\ (2 * n + 1) = 0
\end{aligned}
$$

and the terms of *sinx* are

$$[0, 1, 0, -1\ /\ 3!, 0, 1\ /\ 5!, 0, -1\ /\ 7!, ...]$$

i.e., the corresponding function for *sin* is

$$
\begin{aligned}
&sina\ (2 * n)\ \ \ \ \ \ = 0\\
&sina\ (2 * n + 1) = (-1)\hat{}n\ /\ (2 * n + 1)\,!
\end{aligned}
$$

This can be proven from the definitions of *cosx* and *sinx*. From this we obtain *Euler's formula*:

$$exp\ (i * x) = cos\ x + i * sin\ x$$

One thing which comes out of Euler's formula is the fact that the exponential is a *periodic function* on the imaginary axis. A function $f : A \to B$ is said to be periodic if there exists $T \in A$ such that

$f\ x = f\ (x + T)$ -- $\forall x \in A$

(therefore, for this definition to make sense, we need addition on A; in fact we normally assume at least a group structure, i.e., addition and subtraction together with a zero and the appropriate laws).

Since *sin* and *cos* are periodic, with period $\tau = 2 * \pi$, we have, using the standard notation $a + i * b$ for some $z = C\ (a, b)$:

$$
\begin{aligned}
&exp\ (z + i * \tau) && = \text{\{- Def. of } z \text{ -\}} \\
&exp\ ((a + i * b) + i * \tau) && = \text{\{- Rearranging -\}} \\
&exp\ (a + i * (b + \tau)) && = \text{\{- } exp \text{ is a homomorphism from } (+) \text{ to } (*) \text{ -\}} \\
&exp\ a * exp\ (i * (b + \tau)) && = \text{\{- Euler's formula -\}} \\
&exp\ a * (cos\ (b + \tau) + i * sin\ (b + \tau)) && = \text{\{- } cos \text{ and } sin \text{ are } \tau\text{-periodic -\}} \\
&exp\ a * (cos\ b + i * sin\ b) && = \text{\{- Euler's formula -\}} \\
&exp\ a * exp\ (i * b) && = \text{\{- } exp \text{ is a homomorphism -\}} \\
&exp\ (a + i * b) && = \text{\{- Def. of } z \text{ -\}} \\
&exp\ z
\end{aligned}
$$

Thus, we see that *exp* is periodic, because $exp\ z = exp\ (z + T)$ with $T = i * \tau$, for all z.

TODO: JP: Was it ever proven that exp is a homomorphism in this way?

TODO: JP: Question: are all periodic functions also periodic in the tayor series space?

8.1.1 Exponential function: Associated code

```
instance Additive r ⇒ Additive (Complex r) where
  (+) = addC
  zero = toC zero
instance AddGroup r ⇒ AddGroup (Complex r) where
  negate (C (a,b)) = C (negate a, negate b)
instance Ring r ⇒ Multiplicative (Complex r) where
  (*) = mulC
  one = toC one
  -- abs = absC – requires Floating r as context
toC :: Additive r ⇒ r → Complex r
toC x = C (x, zero)
addC :: Additive r ⇒ Complex r → Complex r → Complex r
addC (C (a,b)) (C (x,y)) = C ((a + x), (b + y))
mulC :: Ring r ⇒    Complex r → Complex r → Complex r
mulC (C (ar,a_i)) (C (br,b_i)) = C (ar * br − a_i * b_i, ar * b_i + a_i * br)
modulusSquaredC :: Ring r ⇒ Complex r → r
modulusSquaredC (C (x,y)) = x ^+ 2 + y ^+ 2
  -- TODO: usually not important (at all)
```

```
  -- absC :: Floating r => Complex r -> Complex r
  -- absC = toC . sqrt . modulusSquaredC
scaleC :: Multiplicative r ⇒ r → Complex r → Complex r
scaleC a (C (x,y)) = C (a * x,a * y)
conj :: AddGroup r ⇒ Complex r → Complex r
conj (C (x,y)) = C (x,−y)
instance Field r ⇒ MulGroup (Complex r) where
  (/) = divC
divC :: Field a ⇒ Complex a → Complex a → Complex a
divC x y = scaleC (1 / modSq) (x * conj y)
  where modSq = modulusSquaredC y
```

8.2 The Laplace transform

This material was inspired by Quinn and Rai [2008], which is highly recommended reading.

Consider the differential equation

$$f'' x - 3 * f' x + 2 * f x = exp (3 * x), f 0 = 1, f' 0 = 0$$

We can solve such equations with the machinery of power series:

```
fs :: (Eq a,Transcendental a) ⇒ PowerSeries a
fs = integ 1 fs'
  where fs' = integ 0 fs''
    fs'' = (exp (3 * x)) + 3 * fs' − 2 * fs
```

We have done this by "zooming in" on the function f and representing it by a power series, $f\ x = \Sigma\ a_n * x\hat{}n$. This allows us to reduce the problem of finding a function $f : \mathbb{R} \to \mathbb{R}$ to that of finding a list of coefficients a_n, or equivalently a function $a : \mathbb{N} \to \mathbb{R}$. Or even, if one wants an approximation only, finding a list of sufficiently many a-values for a good approximation.

TODO: JP: We're sweeping under the rug that we never defined *exp* $(k * x)$. Perhaps we should do it and simplify the exposition of exp (ix) at the same time.

Still, recursive equations are not always easy to solve (especially without a computer), so it's worth looking for alternatives.

When "zooming in" we go from f to a, but we can also look at it in the other direction: we have "zoomed out" from a to f via an infinite series:

$$a : \mathbb{N} \to \mathbb{R} \xrightarrow{\Sigma a_n * x^n} f : \mathbb{R} \to \mathbb{R}$$

We would like to go one step further

$$a : \mathbb{N} \to \mathbb{R} \xrightarrow{\Sigma a_n * x^n} f : \mathbb{R} \to \mathbb{R} \xrightarrow{??} F{:}?$$

That is, we are looking for a transformation from f to some F in a way which resembles the transformation from a to f. The analogue of "sum of an infinite series" for a continuous function is an integral:

$$a : \mathbb{N} \to \mathbb{R} \xrightarrow{\sum a_n * x^n} f : \mathbb{R} \to \mathbb{R} \xrightarrow{\int (ft) * x^t dt} F{:}?$$

We note that, for the integral $\int_0^\infty (f\ t) * x\hat{}t\ dt$ to converge for a larger class of functions (say, bounded functions), we have to limit ourselves to $|x| < 1$. Both this condition and the integral make sense for $x \in \mathbb{C}$, so we could take

TODO: JP: We never explained what a bounded function is

$$a : \mathbb{N} \to \mathbb{R} \xrightarrow{\sum a_n * x^n} f : \mathbb{R} \to \mathbb{R} \xrightarrow{\int (ft) * x^t dt} F : \{ z \mid |z| < 1 \} \to \mathbb{C}$$

but let us stick to $\mathbb{R}$ for now.

Writing, somewhat optimistically

$$\mathscr{L}\ f\ x = \int_0^\infty (f\ t) * x\hat{}t\ dt$$

we can ask ourselves what $\mathscr{L}\ f'$ looks like. After all, we want to solve *differential* equations by "zooming out". We have

$$\mathscr{L}\ f'\ x = \int_0^\infty (f'\ t) * x\hat{}t\ dt$$

Remember that $D\ (f * g) = D\ f * g + f * D\ g$, therefore

TODO: JP: Random remark: it's unfortunate that $*$ looks like the convolution operator, especially annoying in the context of laplace transforms. Is there any way to typeset this better?

$$\begin{aligned}
&\mathscr{L}\ f'\ x && = \{\text{- } g\ t = x\hat{}t;\ g'\ t = log\ x * x\hat{}t \text{ (t is the variable here, not x)}\\
&\int_0^\infty (D\ (f\ t * x\hat{}t)) - f\ t * log\ x * x\hat{}t\ dt && = \{\text{- Linearity of integration -}\}\\
&\int_0^\infty (D\ (f\ t * x\hat{}t))\ dt - \int_0^\infty f\ t * log\ x * x\hat{}t\ dt = \\
&lim_{t\to\infty}(f\ t * x\hat{}t) - (f\ 0 * x\hat{}0) - log\ x * \int_0^\infty f\ t * x\hat{}t\ dt = \{\text{- abs x < 1 -}\}\\
&-f\ 0 - log\ x * \int_0^\infty f\ t * x\hat{}t\ dt = \\
&-f\ 0 - log\ x * \mathscr{L}\ f\ x
\end{aligned}$$

The factor $log\ x$ is somewhat awkward. Let us therefore return to the definition of $\mathscr{L}$ and operate a change of variables:

$$\begin{aligned}
&\mathscr{L}\ f\ x = \int_0^\infty (f\ t) * x\hat{}t\ dt && \Leftrightarrow \{\text{- } x = exp\ (log\ x) \text{ -}\}\\
&\mathscr{L}\ f\ x = \int_0^\infty (f\ t) * (exp\ (log\ x))\hat{}t\ dt && \Leftrightarrow \{\text{- } (a\hat{}b)\hat{}c = a\hat{}(b * c) \text{ -}\}\\
&\mathscr{L}\ f\ x = \int_0^\infty (f\ t) * exp\ (log\ x * t)\ dt
\end{aligned}$$

Since $log\ x < 0$ for $|x| < 1$, we make the substitution $-s = log\ x$. The condition $|x| < 1$ becomes $s > 0$ (or, in $\mathbb{C}$, *real* $s > 0$), and we have

$$\mathscr{L}\ f\ s = \int_0^\infty (f\ t) * exp\ (-s * t)\ dt$$

This is the definition of the Laplace transform of the function f. Going back to the problem of computing $\mathscr{L}\ f'$, we now have

$$\begin{array}{ll} \mathscr{L}\ f'\ s & = \{\text{- The computation above with } s = -log\ x. \text{ -}\} \\ -f\ 0 + s * \mathscr{L}\ f\ s & \end{array}$$

We have obtained

$$\mathscr{L}\ f'\ s = s * \mathscr{L}\ f\ s - f\ 0 \quad \text{-- The "Laplace-D" law}$$

From this, we can deduce

$$\begin{array}{ll} \mathscr{L}\ f''\ s & = \{\text{- Laplace-D for } f' \text{ -}\} \\ s * \mathscr{L}\ f'\ s - f'\ 0 & = \{\text{- Laplace-D for } f \text{ -}\} \\ s * (s * \mathscr{L}\ f\ s - f\ 0) - f'\ 0 & = \{\text{- Simplification -}\} \\ s\hat{}2 * \mathscr{L}\ f\ s - s * f\ 0 - f'\ 0 & \end{array}$$

Exercise 8.1: what is the general formula for $\mathscr{L}\ f^{(k)}s$?

Returning to our differential equation, we have

$$\begin{array}{l} f''\ x - 3 * f'\ x + 2 * f\ x = exp\ (3 * x), f\ 0 = 1, f'\ 0 = 0 \\ \Leftrightarrow \{\text{- point-free form -}\} \\ f'' - 3 * f' + 2 * f = exp \circ (3*), f\ 0 = 1, f'\ 0 = 0 \\ \Rightarrow \{\text{- applying } \mathscr{L} \text{ to both sides -}\} \\ \mathscr{L}\ (f'' - 3 * f' + 2 * f) = \mathscr{L}\ (exp \circ (3*)), f\ 0 = 1, f'\ 0 = 0 \quad \text{-- Eq. (1)} \end{array}$$

Remark: Note that this is a necessary condition, but not a sufficient one. The Laplace transform is not injective. For one thing, it does not take into account the behaviour of f for negative arguments. Because of this, we often assume that the domain of definition for functions to which we apply the Laplace transform is $\mathbb{R}_{\geq 0}$. For another, it is known that changing the values of f for a countable number of its arguments does not change the value of the integral.

According to the definition of $\mathscr{L}$ and because of the linearity of the integral, we have that, for any f and g for which the transformation is defined, and for any constants α and β

$$\mathscr{L}\ (\alpha \circledast f + \beta \circledast g) = \alpha \circledast \mathscr{L}\ f + \beta \circledast \mathscr{L}\ g$$

Note that this is an equality between functions. Indeed, recalling Chapter 7, we are working here with the vector space of functions (f and g are elements of it). The operator $(\circledast)$ refers to scaling in a vector space — here scaling functions. The above equation says that $\mathscr{L}$ is a linear transformation in that space.

TODO: JP: So what is its "matrix" representation?

Applying this linearity property to the left-hand side of (1), we have for any s:

$$
\begin{aligned}
&\mathscr{L}\ (f'' - 3 * f' + 2 * f)\ s \\
&= \{\text{- } \mathscr{L} \text{ is linear -}\} \\
&\mathscr{L}\ f''\ s - 3 * \mathscr{L}\ f'\ s + 2 * \mathscr{L}\ f\ s \\
&= \{\text{- re-writing } \mathscr{L}\ f'' \text{ and } \mathscr{L}\ f' \text{ in terms of } \mathscr{L}\ f \text{ -}\} \\
&s\hat{}2 * \mathscr{L}\ f\ s - s * f\ 0 - f'\ 0 - 3 * (s * \mathscr{L}\ f\ s - f\ 0) + 2 * \mathscr{L}\ f\ s \\
&= \{\text{-} f\ 0 = 1, f'\ 0 = 0 \text{ -}\} \\
&(s\hat{}2 - 3 * s + 2) * \mathscr{L}\ f\ s - s + 3
\end{aligned}
$$

For the right-hand side, we apply the definition:

$$
\begin{aligned}
&\mathscr{L}\ (exp \circ (3*))\ s && = \{\text{- Def. of } \mathscr{L} \text{ -}\} \\
&\int_0^\infty\ exp\ (3 * t) * exp\ (-s * t)\ dt && = \\
&\int_0^\infty\ exp\ ((3 - s) * t))\ dt && = \\
&lim_{t \to \infty} \frac{exp\ ((3-s)*t)}{3-s} - \frac{exp\ ((3-s)*0)}{3-s} && = \{\text{- for } s > 3 \text{ -}\} \\
&\frac{1}{s-3}
\end{aligned}
$$

Therefore, we have, writing F for $\mathscr{L}\ f$

$$(s\hat{}2 - 3 * s + 2) * F\ s - s + 3 = \frac{1}{s-3}$$

and therefore

$$
\begin{aligned}
&F\ s && = \{\text{- Solve for } F\ s \text{ -}\} \\
&\frac{\frac{1}{s-3} + s - 3}{s\hat{}2 - 3 * s + 2} && = \{\text{- } s\hat{}2 - 3 * s + 2 = (s-1) * (s-2) \text{ -}\} \\
&\frac{10 - 6 * s + s\hat{}2}{(s-1) * (s-2) * (s-3)}
\end{aligned}
$$

We now have the problem of "recovering" the function f from its Laplace transform. The standard approach is to use the linearity of $\mathscr{L}$ to write F as a sum of functions with known inverse transforms. We know one such function:

$$exp\ (\alpha * t)\ \{\text{- is the inverse Laplace transform of -}\}\ 1\ /\ (s - \alpha)$$

In fact, in our case, this is all we need.

The idea is to write $F\ s$ as a sum of three fractions with denominators $s - 1$, $s - 2$, and $s - 3$ respectively, i.e., to find A, B, and C such that

$$
\begin{aligned}
&A\ /\ (s-1) + B\ /\ (s-2) + C\ /\ (s-3) = (10 - 6 * s + s\hat{}2)\ /\ ((s-1) * (s-2) * (s-3)) \\
&\Rightarrow \\
&A * (s-2) * (s-3) + B * (s-1) * (s-3) + C * (s-1) * (s-2) = 10 - 6 * s + s\hat{}2 \quad \text{-- (2)}
\end{aligned}
$$

We need this equality (2) to hold for values $s > 3$. A *sufficient* condition for this is for (2) to hold for *all* s. A *necessary* condition for this is for (2) to hold for the specific values 1, 2, and 3.

$$\begin{array}{ll} \text{For } s = 1 : A * (-1) * (-2) & = 10 - 6 + 1 \Rightarrow A = 2.5 \\ \text{For } s = 2 : B * 1 * (-1) & = 10 - 12 + 4 \Rightarrow B = -2 \\ \text{For } s = 3 : C * 2 * 1 & = 10 - 18 + 9 \Rightarrow C = 0.5 \end{array}$$

It is now easy to check that, with these values, (2) does indeed hold, and therefore that we have

$$F\ s = 2.5 * (1\ /\ (s-1)) - 2 * (1\ /\ (s-2)) + 0.5 * (1\ /\ (s-3))$$

The inverse transform is now easy:

$$f\ t = 2.5 * exp\ t - 2 * exp\ (2 * t) + 0.5 * exp\ (3 * t)$$

Our mix of necessary and sufficient conditions makes it necessary to check that we have, indeed, a solution for the differential equation. The verification is in this case trivial.

8.3 Laplace and other transforms

To sum up, we have defined the Laplace transform and shown that it can be used to solve differential equations. It can be seen as a continuous version of the transform between the infinite sequence of coeeficients $a : \mathbb{N} \to \mathbb{R}$ and the functions behind formal power series.

Laplace is also closely related to Fourier series, which is a way of expressing functions on a closed interval as a linear combination of dicrete frequency components rather than as a function of time. Finally, Laplace is also a close relative of the Fourier transform. Both transforms are used to express functions as a sum of "complex frequencies", but Laplace allows a wider range of functions to be transformed. A nice local overview and comparison is B. Berndtsson's "Fourier and Laplace Transforms"[1] Fourier analysis is a common tool in courses on Transforms, Signals and Systems.

[1] Available from http://www.math.chalmers.se/Math/Grundutb/CTH/mve025/1516/Dokument/F-analys.pdf.

8.4 Exercises

Exercise 8.1. Starting from the "Laplace-D" law

$$\mathscr{L}\, f'\, s = s * \mathscr{L}\, f\, s - f\, 0$$

Derive a general formula for $\mathscr{L}\ f^{(k)} s$.

Exercise 8.2. Find the Laplace transforms of the following functions:

1. $\lambda t.\, 3 * e^{5*t}$
2. $\lambda t.\, e^{\alpha * t} - \beta$
3. $\lambda t.\, e^{(t + \frac{\pi}{6})}$

Exercise 8.3.

1. Show that:
   - $\sin t = \frac{1}{2*i}\left(e^{i*t} - e^{-i*t}\right)$
   - $\cos t = \frac{1}{2}\left(e^{i*t} + e^{-i*t}\right)$
2. Find the Laplace transforms $\mathscr{L}$ *sin* and $\mathscr{L}$ *cos* using the transform for the exponentials and the result from 1.

8.4.1 Exercises from old exams

Exercise 8.4. *From exam 2016-03-15*

Consider the following differential equation:

$$f''\, t - 2 * f'\, t + f\, t = e^{2*t}, \quad f\, 0 = 2, \quad f'\, 0 = 3$$

Solve the equation using the Laplace transform. You should need only one formula (and linearity):

$$\mathcal{L}\left(\lambda t.\, e^{\alpha * t}\right) s = 1/(s - \alpha)$$

Exercise 8.5. *From exam 2016-08-23*

Consider the following differential equation:

$$f''\, t - 5 * f'\, t + 6 * f\, t = e^{t}, \quad f\, 0 = 1, \quad f'\, 0 = 4$$

Solve the equation using the Laplace transform. You should need only one formula (and linearity):

$$\mathcal{L}\left(\lambda t.\, e^{\alpha * t}\right) s = 1/(s - \alpha)$$

Exercise 8.6. *From exam 2016-Practice*

Consider the following differential equation:

$$f'' t - 2 * f' t + f t - 2 = 3 * e^{2*t}, \quad f 0 = 5, \quad f' 0 = 6$$

Solve the equation using the Laplace transform. You should need only one formula (and linearity):

$$\mathcal{L}\,(\lambda t.\, e^{\alpha * t})\, s = 1/(s - \alpha)$$

Exercise 8.7. *From exam 2017-03-14*

Consider the following differential equation:

$$f'' t + 4 * f t = 6 * \cos t, \quad f 0 = 0, \quad f' 0 = 0$$

Solve the equation using the Laplace transform. You should need only two formulas (and linearity):

$$\mathcal{L}\,(\lambda t.\, e^{\alpha * t})\, s = 1/(s - \alpha)$$

$$2 * \cos t = e^{i*t} + e^{-i*t}$$

Exercise 8.8. *From exam 2017-08-22*

Consider the following differential equation:

$$f'' t - 3\sqrt{2} * f' t + 4 * f t = 0, \quad f 0 = 2, \quad f' 0 = 3\sqrt{2}$$

Solve the equation using the Laplace transform. You should need only one formula (and linearity):

$$\mathcal{L}\,(\lambda t.\, e^{\alpha * t})\, s = 1/(s - \alpha)$$

Chapter 9

Probability Theory

```
{-# LANGUAGE GADTs #-}
module DSLsofMath.W09 where
import Control.Monad (ap)
import Data.List ((\\))
type ℝ = Double   -- pretend...
```

In this chapter, we define a DSL to describe and reason about problems such as those of the following list, and sometimes we can even compute the probabilites involved by evaluating the DSL expressions.

1. Assume you throw two 6-faced dice, what is the probability that the product is greater than 10 if their sum is greater than 7?
2. Suppose that a test for using a particular drug is 99% sensitive and 99% specific. That is, the test will produce 99% true positive results for drug users and 99% true negative results for non-drug users. Suppose that 0.5% of people are users of the drug. What is the probability that a randomly selected individual with a positive test is a drug user? [1]
3. Suppose you're on Monty Hall's *Let's Make a Deal!* You are given the choice of three doors, behind one door is a car, the others, goats. You pick a door, say 1, Monty opens another door, say 3, which has a goat. Monty says to you "Do you want to pick door 2?" Is it to your advantage to switch your choice of doors?

Our method will be to:

[1] Example found in Wikipedia article on Bayes' theorem

- Describe the space of possible situations, or outcomes.
- Define the events whose probabilities we will consider.
- Evaluate such probabilities.

Depending on the context, we use the word "situation" or "outcome" for the same mathematical objects. The word "outcome" evokes some experiment, explicitly performed; and the "outcome" is the situation after the experiment is over. When we use the word "situation" there is not necessarily an explict experiment, but something happens according to a specific scenario. We consider the situation at the end of the scenario in question.

9.1 Sample spaces

Generally textbook problems involving probability involve the description of some scenario or experiment, with an explicit uncertainty, including the outcome of certain measures, then the student is asked to compute the probability of some event.

It is common to refer to a sample space by the labels S, Ω, or U, but in this chapter we will define many such spaces, and therefore we will use descriptive names instead. While the concept of a space of events underpins modern understandings of probability theory, textbooks sometimes give a couple of examples involving coin tosses and promptly forget the concept of sample space in the body of the text. Here we will instead develop this concept using our DSL methodology. Once this is done, we will be able to see that an accurate model of the sample space is an essential tool to solve probability problems.

Our first task is to describe the possible structure of sample spaces, and model them as a DSL. To this end we will use a data type to represent spaces. This type is indexed by the underlying Haskell type of possible outcomes.

$$Space :: Type \to Type$$

Finite space In Example 1., we consider dice with 6 faces. For such a purpose we define a constructor *Finite* embedding a list of possible outcomes into a space:

$$Finite :: [a] \to Space\ a$$

The scenario (or "experiment") corresponding to throwing a die is then represented by the following space:

$$
\begin{aligned}
&\mathit{die} :: \mathit{Int} \to \mathit{Space}\ \mathit{Int} \\
&\mathit{die}\ n = \mathit{Finite}\ [1\,..\,n] \\
&\mathit{d6} = \mathit{die}\ 6
\end{aligned}
$$

In particular the space $\mathit{point}\ x$ is the space with a single point — only trivial probabilities (zero or one) are involved here:

$$
\begin{aligned}
&\mathit{point} :: a \to \mathit{Space}\ a \\
&\mathit{point}\ x = \mathit{Finite}\ [x]
\end{aligned}
$$

Scaling space If the die is well-balanced, then all cases have the same probability (or probability mass) in the space, and this is what we have modelled above. But this is not always the case. Hence we need a way to represent this. We use the following combinator: [2]

$$\mathit{Factor} :: \mathbb{R} \to \mathit{Space}\ ()$$

Its underlying type is the unit type $()$, but its mass (or density) is given by a real number.

On its own, *Factor* may appear useless, but we can setup the *Space* type so that this mass or density can depend on (previously introduced) spaces.

Product of spaces As a first instance of a dependency, we introduce the product of two spaces, as follows:

$$\mathit{prod} :: \mathit{Space}\ a \to \mathit{Space}\ b \to \mathit{Space}\ (a, b)$$

For example, the "experiment" of throwing two 6-faced dice is represented as follows:[3]

$$
\begin{aligned}
&\mathit{twoDice} :: \mathit{Space}\ (\mathit{Int}, \mathit{Int}) \\
&\mathit{twoDice} = \mathit{prod}\ \mathit{d6}\ \mathit{d6}
\end{aligned}
$$

But let's say now that we know that the sum is greater than 7. We can then define the following parametric space, which has a mass 1 if the condition is satisfied and 0 otherwise. (This space is trivial in the sense that no uncertainty is involved.)

[2] This is in fact scaling, as defined in the LinAlg Chapter — the spaces over a given domain form a vector space. However, we choose not to use this point of view, because we are generally not interested in the vector space structure of probability spaces. Furthermore, using this terminology risks adding more confusion than otherwise. In particular, the word "scale" could be misunderstood as scaling the *value* of a numerical variable. Instead here we scale densites, and use *Factor* for this purpose.

[3] The use of a pair corresponds to the fact that the two dice can be identified individually.

$$sumAbove7 :: (Int, Int) \to Space\ ()$$
$$sumAbove7\ (x, y) = Factor\ (\textbf{if}\ x + y > 7\ \textbf{then}\ 1\ \textbf{else}\ 0)$$

We now want to take the product of the *twoDice* space and the *sumAbove7* space; but the issue is that *sumAbove7 depends* on the outcome of *twoDice*. To support this dependency we need a generalisation of the product which we call "Sigma" (Σ) because of its similarity of structure with the summation operation.

$$\Sigma :: Space\ a \to (a \to Space\ b) \to Space\ (a, b)$$

Hence:

$$problem1 :: Space\ ((Int, Int), ())$$
$$problem1 = \Sigma\ twoDice\ sumAbove7$$

The values of the dice are the same as in *twoDice*, but the density of any sum less than 7 is brought down to zero.

We can check that the product of spaces is a special case of Σ:

$$prod\ a\ b = \Sigma\ a\ (const\ b)$$

If we compare the above to the usual sum notation, the right-hand-side would be $\sum_{i \in a} b$ which is the sum of *card a* copies of b, thus a kind of "product" of a and b.

Projections In the end we may not be interested in all values and hide some of them. For this purpose we use the following combinator:

$$Project :: (a \to b) \to Space\ a \to Space\ b$$

A typical use is $Project\ fst :: Space\ (a, b) \to Space\ a$.

Real line Before we continue, we may also add a way to represent real-valued spaces, with the same probability for every real number.

$$RealLine :: Space\ \mathbb{R}$$

Summary In sum, we have a datatype for the abstract syntax of "space expressions":

```
data Space a where
  Finite   :: [a] → Space a
  Factor   :: ℝ → Space ()
  Σ        :: Space a → (a → Space b) → Space (a, b)
  Project  :: (a → b) → Space a → Space b
  RealLine :: Space ℝ
```

9.2 Bind and return

Very often, one will be interested only in a particular projection only. Hence, it is useful to combine Σ and *Project* in a single combinator, as follows:

```
bind :: Space a → (a → Space b) → Space b
bind a f = Project snd (Σ a f)
```

This means that *Space* has a monadic structure:

TODO: explain this somewhere in the book

```
instance Functor Space where
  fmap f = (pure f <*>)
instance Applicative Space where
  pure x = Finite [x]
  (<*>) = ap
instance Monad Space where
  (>>=) = bind
```

9.3 Distributions

So far we have defined several spaces, not used them to compute any probability. We set out to do this in this section. In section 9.4 we will see how to compute the *measure* :: *Space a* → $\mathbb{R}$ of a space but before that we will talk about another important notion in probability theory: that of a distribution. A distribution is a space whose total mass (or measure) is equal to 1.

```
isDistribution :: Space a → Bool
isDistribution s = measure s == 1
```

We may use the following type synonym to indicate distributions:

```
type Distr a = Space a
```

Let us define a few useful distributions. First, we present the uniform distribution among a finite set of elements. It is essentially the same as the *Finite* space, but we scale every element so that the total measure comes down to 1.

```
factorWith :: (a → ℝ) → Space a → Space a
factorWith f s = do {x ← s; Factor (f x); return x}
scale :: ℝ → Space a → Space a
scale c = factorWith (const c)

uniformDiscrete :: [a] → Distr a
uniformDiscrete xs = scale (1.0 / fromIntegral (length xs))
                           (Finite xs)
```

The distribution of the balanced die can then be represented as follows:

$$
\begin{aligned}
&\mathit{dieDistr} :: \mathit{Distr}\ \mathit{Integer}\\
&\mathit{dieDistr} = \mathit{uniformDiscrete}\ [1\,..\,6]
\end{aligned}
$$

Another useful discrete distribution is the Bernoulli distribution of parameter p. It is a distribution whose value is *True* with probability p and and *False* with probability $1 - p$. Hence it can be used to represent a biased coin toss.

$$
\begin{aligned}
&\mathit{bernoulli} :: \mathbb{R} \to \mathit{Distr}\ \mathit{Bool}\\
&\mathit{bernoulli}\ p = \mathit{factorWith}\ (\lambda b \to \textbf{if}\ b\ \textbf{then}\ p\ \textbf{else}\ 1 - p)\\
&\qquad\qquad\qquad\qquad (\mathit{Finite}\ [\mathit{False}, \mathit{True}])
\end{aligned}
$$

Finally we can define the normal distribution with average μ and standard deviation σ.

$$
\begin{aligned}
&\mathit{normal} :: \mathbb{R} \to \mathbb{R} \to \mathit{Distr}\ \mathbb{R}\\
&\mathit{normal}\ \mu\ \sigma = \mathit{factorWith}\ (\mathit{normalMass}\ \mu\ \sigma)\ \mathit{RealLine}\\
&\mathit{normalMass} :: \mathit{Floating}\ r \Rightarrow r \to r \to r \to r\\
&\mathit{normalMass}\ \mu\ \sigma\ x = \mathit{exp}\ (-(((x - \mu)\ /\ \sigma)\hat{}2\ /\ 2))\ /\ (\sigma * \sqrt{2 * \pi})
\end{aligned}
$$

In some textbooks, distributions are sometimes called "random variables". However we consider this terminology to be misleading — random variables will be defined precisely later on.

We could try to define the probabilities or densities of possible values of a distribution, say $\mathit{distributionDensity} :: \mathit{Space}\ a \to (a \to \mathbb{R})$, and from there define the expected value (and other statistical moments), but we'll take another route.

9.4 Semantics of spaces

First, we come back to general probability spaces without restriction one their *measure*: it does not need to be equal to one. We define a function *integrator*, which generalises the notions of weighted sum, and weighted integral. When encountering *Finite* spaces, we sum; when encountering *RealLine* we integrate. When encountering *Factor* we will adjust the weights. The integrator of a product (in general Σ) is the nested integration of spaces. The weight is given as a second parameter to *integrator*, as a function mapping elements of the space to a real value.

$$
\begin{aligned}
&\mathit{integrator} :: \mathit{Space}\ a \to (a \to \mathbb{R}) \to \mathbb{R}\\
&\mathit{integrator}\ (\mathit{Finite}\ a) \quad g = \mathit{bigsum}\ a\ g\\
&\mathit{integrator}\ (\mathit{RealLine}) \quad g = \mathit{integral}\ g
\end{aligned}
$$

$$
\begin{array}{ll}
\mathit{integrator}\ (\mathit{Factor}\ f) & g = f * g\ () \\
\mathit{integrator}\ (\Sigma\ a\ f) & g = \mathit{integrator}\ a\ \$\ \lambda x \to \mathit{integrator}\ (f\ x)\ \$\ \lambda y \to g\ (x, y) \\
\mathit{integrator}\ (\mathit{Project}\ f\ a)\ g & = \mathit{integrator}\ a\ (g \circ f)
\end{array}
$$

The above definition relies on the usual notions of sum (*bigsum*) and *integral*. We can define the sum of some terms, for finite lists, as follows:

$$
\begin{array}{l}
\mathit{bigsum} :: [a] \to (a \to \mathbb{R}) \to \mathbb{R} \\
\mathit{bigsum}\ xs\ f = \mathit{sum}\ (\mathit{map}\ f\ xs)
\end{array}
$$

We use also the definite integral over the whole real line. However we will leave this concept undefined at the Haskell level — thus whenever using real-valued spaces, our defintions are not usable for numerical computations, but for symbolic computations only.

$$
\begin{array}{l}
\mathit{integral} :: (\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \\
\mathit{integral} = \mathit{undefined}
\end{array}
$$

The simplest quantity that we can compute using the integrator is the measure of the space — it total "mass" or "volume". To compute the measure of a space, we can simply integrate the constant 1 (so only mass matters).

$$
\begin{array}{l}
\mathit{measure} :: \mathit{Space}\ a \to \mathbb{R} \\
\mathit{measure}\ d = \mathit{integrator}\ d\ (\mathit{const}\ 1)
\end{array}
$$

As a sanity check, we can compute the measure of a Bernoulli distribution (we use the $>>>$ notation to indicate an expression being computed) and find that it is indeed 1.

$$
\begin{array}{l}
\text{-- } >>> \mathit{measure}\ (\mathit{bernoulli}\ 0.2) \\
\text{-- } 1.0
\end{array}
$$

The integration of *id* over a real-valued distribution yields its expected value.[4]

$$
\begin{array}{l}
\mathit{expectedValueOfDistr} :: \mathit{Distr}\ \mathbb{R} \to \mathbb{R} \\
\mathit{expectedValueOfDistr}\ d = \mathit{integrator}\ d\ \mathit{id} \\
\quad \text{-- } >>> \mathit{expectedValueOfDistr}\ \mathit{dieDistr} \\
\quad \text{-- } 3.5
\end{array}
$$

Exercise: compute symbolically the expected value of the *bernoulli* distribution.

Properties of *integrator* We can use the definitions to show some useful calculational properties of spaces.

TODO: Format the lemmas

TODO: Name the lemmas

[4] We reserve the name *expectedValue* for the expected value of a random variable, defined later.

integrator/bind lemma: $integrator\ (s \gg\!= f)\ g == integrator\ s\ \$\ \lambda x \to integrator\ (f\ x)\ g$

$$
\begin{aligned}
&integrator\ (s \gg\!= f)\ g \\
=\ &\{\text{- by Def. of bind } (\gg\!=) \text{ -}\} \\
&integrator\ (Project\ snd\ (\Sigma\ s\ f))\ g \\
=\ &\{\text{- by Def. of integrator/project -}\} \\
&integrator\ (\Sigma\ s\ f)\ (g \circ snd) \\
=\ &\{\text{- by Def. of integrator/Sigma -}\} \\
&integrator\ s\ \$\ \lambda x \to integrator\ (f\ x)\ \$\ \lambda y \to (g \circ snd)\ (x, y) \\
=\ &\{\text{- snd/pair -}\} \\
&integrator\ s\ \$\ \lambda x \to integrator\ (f\ x)\ \$\ \lambda y \to g\ y \\
=\ &\{\text{- composition -}\} \\
&integrator\ s\ \$\ \lambda x \to integrator\ (f\ x)\ g
\end{aligned}
$$

The same in **do** notation: $integrator\ (\mathbf{do}\ x \leftarrow s; t)\ g == integrator\ s\ \$\ \lambda x \to integrator\ t\ g$

integrator/return lemma: $integrator\ (return\ x)\ g == g\ x$

$$
\begin{aligned}
&integrator\ (return\ x)\ g \\
=\ &\{\text{- by Def. of } return \text{ -}\} \\
&integrator\ (Finite\ [x])\ g \\
=\ &\{\text{- integrator/Finite -}\} \\
&sum\ (map\ g)\ [x] \\
=\ &\{\text{- by Def. of sum/map -}\} \\
&g\ x
\end{aligned}
$$

integrator/fmap lemma: $integrator\ (fmap\ f\ s)\ g == integrator\ s\ (g \circ f)$

$$
\begin{aligned}
&integrator\ (fmap\ f\ s)\ g \\
=\ &\{\text{- by Def. of } fmap \text{ -}\} \\
&integrator\ (s \gg\!= (return \circ f))\ g \\
=\ &\{\text{- by integrator/bind lemma -}\} \\
&integrator\ s\ \$\ \lambda x \to integrator\ (return\ (f\ x))\ g \\
=\ &\{\text{- by integrator/return lemma -}\} \\
&integrator\ s\ \$\ \lambda x \to g\ (f\ x) \\
=\ &\{\text{- definition of } (\circ) \text{ -}\} \\
&integrator\ s\ (g \circ f)
\end{aligned}
$$

Linearity of integrator:

If g is a linear function (addition or multiplication by a constant), then:

$$integrator\ s\ (g \circ f) == g\ (integrator\ s\ f)$$

(or, equivalently, $integrator\ s\ (\lambda x \to g\ (f\ x)) = g\ (integrator\ s\ f)$)

The proof proceeds by structural induction over s. The hypothesis of linearity is used in the base cases. Notably the linearity property of *Finite* and *RealLine* hinge on the linearity of sums and integrals. The case of *Project* is immediate by definition. The case for Σ is proven as follows:

$$
\begin{aligned}
&\mathit{integrator}\ (\Sigma\ a\ f)\ (g \circ h) \\
&= \{\text{- By definition -}\} \\
&\mathit{integrator}\ a\ \$\ \lambda x \to \mathit{integrator}\ (f\ x)\ \$\ \lambda y \to g\ (h\ (x, y)) \\
&= \{\text{- By induction -}\} \\
&\mathit{integrator}\ a\ \$\ \lambda x \to g\ (\mathit{integrator}\ (f\ x)\ \$\ \lambda y \to h\ (x, y) \\
&= \{\text{- By induction -}\} \\
&g\ (\mathit{integrator}\ a\ \$\ \lambda x \to (\mathit{integrator}\ (f\ x)\ \$\ \lambda y \to h\ (x, y)) \\
&= \{\text{- By definition -}\} \\
&g\ (\mathit{integrator}\ (\Sigma\ a\ f)\ h)
\end{aligned}
$$

As a corrolary, *fmap* is linear as well: if f is linear, then $\mathit{integrator}\ (\mathit{fmap}\ f\ s)\ g = f\ (\mathit{integrator}\ s\ g)$

Properties of *measure*

- $\mathit{measure}\ (\mathit{pure}\ x) == 1$
- If f is a constant function such that $f\ x = t$, then $\mathit{measure}\ (\Sigma\ s\ f) == \mathit{measure}\ s * \mathit{measure}\ t$.
- $\mathit{measure}\ (s \gg\!= f) == \mathit{measure}\ s * \mathit{measure}\ (f\ k)$, under the same condition. TODO: k is unbound

$\gg\!=$/distribution If s is a distribution and $f\ x$ is a distribution for every x, then $s \gg\!= f$ is a distribution

fmap/distribution Corollary: s is a distribution iff. $\mathit{fmap}\ f\ s$ is a distribution.

The proof of the second item is as follows:

$$
\begin{aligned}
&\mathit{measure}\ (\Sigma\ s\ f) \\
&== \{\text{- by definition of measure -}\} \\
&\mathit{integrator}\ (\Sigma\ s\ f)\ (\mathit{const}\ 1) \\
&== \{\text{- by definition of integrator for Sigma -}\} \\
&\mathit{integrator}\ s\ \$\ \lambda x \to \mathit{integrator}\ (f\ x)\ (\mathit{const}\ 1) \\
&== \{\text{- by definition of of measure -}\} \\
&\mathit{integrator}\ s\ \$\ \lambda x \to (\mathit{measure}\ (f\ x)) \\
&== \{\text{- by assumption -}\} \\
&\mathit{integrator}\ s\ \$\ \lambda x \to (\mathit{measure}\ (f\ k)) \\
&== \{\text{- By linearity of integrator -}\} \\
&\mathit{measure}\ (f\ k) * (\mathit{integrator}\ s\ \$\ \lambda x \to 1) \\
&== \{\text{- By definition of measure -}\} \\
&\mathit{measure}\ (f\ k) * \mathit{measure}\ s
\end{aligned}
$$

Exercise: using the above lemmas, prove $integrator\ (bernoulli\ p)\ f == p * integrator\ f + (1 - p) * integrator\ f$.

9.5 Random Variables

Even though we have already studied variables in detail (in Chapter 3), it is good to come back to them for a moment before returning to *random* variables proper.

According to Wikipedia, a variable has a different meaning in computer science and in mathematics:

> Variable may refer to:
>
> Variable (computer science) , a symbolic name associated with a value and whose associated value may be changed
>
> Variable (mathematics) , a symbol that represents a quantity in a mathematical expression, as used in many sciences

By now we have a pretty good grip on variables in computer science. In Chapter 3. we have described a way to reduce mathematical variables (position q and velocity v in lagrangian mechanics) to computer science variables. This was done by expressing variables as *functions* of a truly free variable, time (t). Time is a "computer science" variable in this context: it can be substituted by any value without "side effects" on other variables (unlike positions (q) and velocities (v)).

In this light, let us return to random variables. Wikipedia is not very helpful here, so we can turn ourselves to Grinstead and Snell [2003], who give the following definition:

> A random variable is simply an expression whose value is the outcome of a particular experiment.

This may be quite confusing at this stage. What are those expressions, and where do the experiment occur in the variable? Our answer is to using spaces to represent the "experiments" that Grinstead and Snell mention. More specifically, if $s : Space\ a$, each possible situation at the end of the experiment is representable in the type a and the space will specify the mass of each of them (via the integrator).

Then, a b-valued random variable f related to an experiment represented by a space $s : Space\ a$ is a function f of type $a \to b$. Then $f\ x$ is the "expression" that Grinstead and Snell refer to. The variable x is the outcome, and s is represents the experiment — which is most often implicit in a random variable expressions as written in a math book.

We finally can define the expected value (and other statistical moments) of random variable.

In textbooks, one will often find the notation $E[t]$ for the expected value of a real-valued random variable t. As just mentioned, from our point of view, this notation can be confusing because it leaves the space of situations completely implicit. That is, it is not clear how t depends on the outcome of experiments.

With our DSL approach, we make this dependency completely explicit. For example, the expected value of real-valued random variable takes the space of outcomes as its first argument:

```
expectedValue :: Space a → (a → ℝ) → ℝ
expectedValue s f = integrator s f / measure s
```

For instance, we can use the above to compute the expected value of the sum of two dice throws:

```
-- >>> expectedValue twoDice (λ(x, y) → fromIntegral (x + y))
-- 7.0
```

Essentially, what the above does is computing the weighted sum/integral of $f\ (x)$ for every point x in the space. Because s is a space (not a distribution), we must normalise the result by dividing by its measure.

Exercise: define various statistical moments (variance, skew, curtosis, etc.)

Theorem: Linearity of expected value Furthermore, if f is linear, then $expectedValue\ (f <\$> s) = f\ (expectedValue\ s)$

```
expectedValue (f <$> s) g
  == {- By definition of expected value -}
integrator (f <$> s) g / measure (f <$> s)
  == {- By linearity of integrator over fmap -}
integrator s (g ∘ f) / measure s
  == {- By definition of expected value -}
expectedValue s (g ∘ f)
```

9.6 Events and probability

In textbooks, one typically finds the notation $P\ (e)$ for the probability of an *event* e. Again, the space of situations s is implicit as well as the dependency between e and s.

Here, we define events as *boolean-valued* random variables.

Thus an event e can be defined as a boolean-valued function $e : a \to Bool$ over a space $s : Space\ a$. Assuming that the space s accurately represents the relative mass of all possible situations, there are two ways to define the probability of e.

The first defintion is as the expected value of $indicator \circ e$, where *indicator* maps boolean to reals as follows:

$$
\begin{aligned}
&indicator :: Bool \to \mathbb{R} \\
&indicator\ True\ = 1 \\
&indicator\ False = 0 \\
&probability1 :: Space\ a \to (a \to Bool) \to \mathbb{R} \\
&probability1\ d\ e = expectedValue\ d\ (indicator \circ e)
\end{aligned}
$$

The second definition of probability is as the ratio of the measure of the subspace where e holds, the measure and the complete space.

$$
\begin{aligned}
&probability2 :: Space\ a \to (a \to Bool) \to \mathbb{R} \\
&probability2\ s\ e = measure\ (\Sigma\ s\ (isTrue \circ e))\ /\ measure\ s \\
&isTrue :: Bool \to Space\ () \\
&isTrue = Factor \circ indicator
\end{aligned}
$$

where *isTrue c* is the subspace which has measure 1 if c is true and 0 otherwise. The subspace of s where e holds is then the first projection of $\Sigma\ s\ (isTrue \circ e)$.

$$
\begin{aligned}
&subspace :: (a \to Bool) \to Space\ a \to Space\ a \\
&subspace\ e\ s = Project\ fst\ (\Sigma\ s\ (isTrue \circ e))
\end{aligned}
$$

We can show that if s has a non-zero measure, then the two definitions are equivalent:

Lemma: $measure\ s * probability\ s\ e = measure\ (\Sigma\ s\ (isTrue \circ e))$ Proof:

$$
\begin{aligned}
&\quad probability1\ s\ e \\
&= \{\text{- Def. of } probability1 \text{ -}\} \\
&\quad expectedValue\ s\ (indicator \circ e) \\
&= \{\text{- Def. of } expectedValue \text{ -}\} \\
&\quad integrator\ s\ (indicator \circ e)\ /\ measure\ s \\
&= \{\text{- Def. of } (\circ) \text{ -}\} \\
&\quad integrator\ s\ (\lambda x \to indicator\ (e\ x))\ /\ measure\ s \\
&= \{\text{- mutiplication by 1 -}\} \\
&\quad integrator\ s\ (\lambda x \to indicator\ (e\ x) * const\ 1\ (x, ()))\ /\ measure\ s \\
&= \{\text{- Def. of } integrator \text{ -}\} \\
&\quad integrator\ s\ (\lambda x \to integrator\ (Factor\ (indicator\ (e\ x)))\ (\lambda y \to const\ 1\ (x, y)))\ /\ measure\ s \\
&= \{\text{- Def. of } isTrue \text{ -}\} \\
&\quad integrator\ s\ (\lambda x \to integrator\ (isTrue\ (e\ x))\ (\lambda y \to const\ 1\ (x, y)))\ /\ measure\ s
\end{aligned}
$$

$= \{\text{- Def. of } \textit{integrator} \text{ -}\}$
$\quad \textit{measure}\ (\Sigma\ s\ (\textit{isTrue} \circ e))\ /\ \textit{measure}\ s$

It will be often convenient to define a space whose underlying set is a boolean value and compute the probability of the identity event:

$\textit{probability} :: \textit{Space Bool} \to \mathbb{R}$
$\textit{probability}\ d = \textit{expectedValue}\ d\ \textit{indicator}$

Sometimes one even finds in the literature and folklore the notation $P\ (v)$, where v is a value, which stands for $P\ (t = v)$, for an implicit random variable t. Here even more creativity is required from the reader, who must not only infer the space of outcomes, but also which random variable the author means.

9.7 Conditional probability

Another important notion in probability theory is conditional probability, written $P(F \mid G)$ and read "probability of f given g".

We can define the conditional probability of f given g by taking the probability of f in the sub space where g holds:

$\textit{condProb} :: \textit{Space}\ a \to (a \to \textit{Bool}) \to (a \to \textit{Bool}) \to \mathbb{R}$
$\textit{condProb}\ s\ f\ g = \textit{probability1}\ (\textit{subspace}\ g\ s)\ f$

We find the above defintion more intuitive than the more usual definition $P(F \mid G) = P(F \cap G)/P(G)$. However, this last equality can be proven, by calculation:

Lemma: $\textit{condProb}\ s\ f\ g == \textit{probability}\ s\ (\lambda y \to f\ y \wedge g\ y)\ /\ \textit{probability}\ s\ g$

Proof:

TODO: Possibly split into helper lemma(s) to avoid diving "too deep".

$\quad \textit{condProb}\ s\ f\ g$
$= \{\text{- Def of condProb -}\}$
$\quad \textit{probability1}\ (\textit{subspace}\ g\ s)\ f$
$= \{\text{- Def of subspace -}\}$
$\quad \textit{probability1}\ (\Sigma\ s\ (\textit{isTrue} \circ g))\ (f \circ \textit{fst})$
$= \{\text{- Def of probability1 -}\}$
$\quad \textit{expectedValue}\ (\Sigma\ s\ (\textit{isTrue} \circ g))\ (\textit{indicator} \circ f \circ \textit{fst})$
$= \{\text{- Def of expectedValue -}\}$
$\quad (1\ /\ \textit{measure}\ (\Sigma\ s\ (\textit{isTrue} \circ g))) * (\textit{integrator}\ (\Sigma\ s\ (\textit{isTrue} \circ g))\ (\textit{indicator} \circ f \circ \textit{fst}))$
$= \{\text{- Def of } \textit{integrator} \text{ (Sigma) -}\}$
$\quad (1\ /\ \textit{measure}\ s\ /\ \textit{probability}\ s\ g) * (\textit{integrator}\ s\ \$\ \lambda x \to \textit{integrator}\ (\textit{isTrue} \circ g)\ \$\ \lambda y \to \textit{indicator} \circ f \circ \textit{fst}\ \$\ (x$

$= \{\text{- Def of } fst \text{ -}\}$
$\quad (1 \,/\, measure\ s \,/\, probability\ s\ g) * (integrator\ s\ \$\ \lambda x \to integrator\ (isTrue \circ g)\ \$\ \lambda y \to indicator \circ f\ \$\ x)$
$= \{\text{- Def of } isTrue \text{ -}\}$
$\quad (1 \,/\, measure\ s \,/\, probability\ s\ g) * (integrator\ s\ \$\ \lambda x \to indicator\ (g\ x) * indicator\ (f\ x))$
$= \{\text{- Property of } indicator \text{ -}\}$
$\quad (1 \,/\, measure\ s \,/\, probability\ s\ g) * (integrator\ s\ \$\ \lambda x \to indicator\ (\lambda y \to g\ y \wedge f\ y))$
$= \{\text{- associativity of multiplication -}\}$
$\quad (1 \,/\, probability\ s\ g) * (integrator\ s\ \$\ \lambda x \to indicator\ (\lambda y \to g\ y \wedge f\ y)) \,/\, measure\ s$
$= \{\text{- Definition of } probability1 \text{ -}\}$
$\quad (1 \,/\, probability\ s\ g) * probability1\ s\ (\lambda y \to g\ y \wedge f\ y)$

9.8 Examples

9.8.1 Dice problem

We will use the monadic interface to define the experiment, hiding all random variables except the outcome that we care about (is the product greater than 10?):

```
diceSpace :: Space Bool
diceSpace = do
  x ← d6   -- balanced die 1
  y ← d6   -- balanced die 2
  isTrue (x + y ⩾ 7)   -- observe that the sum is >= 7
  return (x * y ⩾ 10)   -- consider only the event "product >= 10"
```

Then we can compute its probability:

```
diceProblem :: ℝ
diceProblem = probability diceSpace
  -- >>> diceProblem
  -- 0.9047619047619047
```

Some helpers:

TODO: Merge with text above to explain more

```
p1 (x,y) = x + y ⩾ 7
p2 (x,y) = x * y ⩾ 10
test1    = measure (prod d6 d6 >>= isTrue ∘ p1)              -- 21
test2    = measure (prod d6 d6 >>= isTrue ∘ p2)              -- 19
testBoth = measure (prod d6 d6 >>= isTrue ∘ λxy → p1 xy ∧ p2 xy)  -- 19
prob21   = condProb (prod d6 d6) p2 p1                       -- 19/21
```

9.9 Drug test

The above drug test problem 2. is often used as an illustration for the Bayes theorem. We can solve it in exactly the same fashion as the Dice problem.

We begin by describing the space of situations. To do so we make heavy use of the *bernoulli* distribution. First we model the distribution of drug users. Then we model the distribution of test outcomes — depending on whether we have a user or not. Finally, we project out all variables, caring only about *isUser*.

```
drugSpace :: Space Bool
drugSpace = do
  isUser ← bernoulli 0.005    -- model the distribution of drug users
  testPositive ← bernoulli (if isUser then 0.99 else 0.01)
      -- model test accuracy
  isTrue testPositive         -- we have "a positive test" by assumption
  return isUser               -- we're interested in isUser.
```

The probability is computed as usual:

```
userProb :: ℝ
userProb = probability drugSpace
  -- >>> userProb
  -- 0.33221476510067116
```

Perhaps surprisingly, we never needed the Bayes theorem to solve the problem. Indeed, the Bayes theorem is already incorporated in our defintion of *probability*.

9.10 Monty Hall

We can model the Monty Hall problem as follows: A correct model is the following:

```
doors :: [Int]
doors = [1,2,3]
montySpace :: Bool → Space Bool
montySpace changing = do
  winningDoor     ← uniformDiscrete doors    -- any door can be the winning one
  pickedDoor      ← uniformDiscrete doors    -- player picks door blindly
  montyPickedDoor ← uniformDiscrete          -- Monty cannot pick the same door, nor a winning door.
                      (doors \\ [pickedDoor, winningDoor])
  let newPickedDoor =
    if changing
    then head (doors \\ [pickedDoor, montyPickedDoor])
```

```
    -- player takes a door which is NOT the previously picked, nor the showed door.
  else pickedDoor
return (newPickedDoor == winningDoor)
-- >>> probability (montySpace False)
-- 0.3333333333333333
-- >>> probability (montySpace True)
-- 0.6666666666666666
```

The Monty Hall is sometimes considered paradoxical: it is strange that changing one's mind can change the outcome. The crucial point is that Monty can never show a door which contains the prize. To illustrate, an *incorrect* way to model the Monty Hall problem, which still appears to follow the example point by point, is the following:

```
montySpaceIncorrect :: Bool → Space Bool
montySpaceIncorrect changing = do
  winningDoor ← uniformDiscrete [1 :: Int, 2, 3]   -- any door can be the winning one
  let pickedDoor = 1                               -- player picks door 1.
  let montyPickedDoor = 3                          -- monty opens door 3.
  isTrue (montyPickedDoor ≠ winningDoor)           -- door 3 is not winning
  let newPickedDoor = if changing then 2 else 1    -- we can change or not
  return (newPickedDoor == winningDoor)            -- has the player won?
-- >>> probability (montySpace1 False)
-- 0.5
-- >>> probability (montySpace1 True)
-- 0.5
```

The above is incorrect, because everything happens as if Monty chooses a door before the player made its first choice.

9.11 Advanced problem

Consider the following problem: how many times must one throw a coin before one obtains 3 heads in a row.

We can model the problem as follows:

```
coin = bernoulli 0.5
coins = do
  x ← coin
  xs ← coins
  return (x : xs)
threeHeads :: [Bool] → Int
threeHeads (True : True : True: _) = 3
```

$$threeHeads\ (\_ : xs) = 1 + threeHeads\ xs$$
$$example' = threeHeads <\$> coins$$

Attempting to evaluate $probability1\ threeHeads'\ (<5)$ does not terminate. Indeed, we have an infinite list, which translates in infinitely many cases to consider. So the evaluator cannot solve this problem. We have to resort to a symbolic method. First, we can unfold the definitions of *coins* in *threeHeads*, and obtain:

```
threeHeads' = helper 3
helper 0 = return 0
helper m = do
  h ← coin
  (1+) <$> if h
    then helper (m − 1)
    else helper 3    -- when we have a tail, we start from scratch
```

Evaluating the probability still does not terminate: we no longer have an infinite list, but we still have infinitely many possibilities to consider: however small, there is always a probability to get a "tail" at the wrong moment, and the evaluation must continue.

We can start by showing that *helper m* is a distribution (its measure is 1). The proof is by induction on m:

- for the base case $measure\ (helper\ 0) = measure\ (pure\ 0) = 1$
- induction: assume that *helper m* is a distribution. Then **if** h **then** *helper m* **else** *helper* 3 for every h. The result is obtained by using the $\gg=$/distribution lemma.

Then, we can symbolically compute the integrator of *helper*. We won't be using the base case. In the recursive case, we have:

```
integrator (helper (m + 1)) f
 == {- By def. of helper -}
integrator (coin >>= λh → (1+) <$> if h then helper m else helper 3) f
 == {- By integrator/bind -}
integrator coin >>= λh → integrator ((1+) <$> if h then helper m else helper 3) f
 == {- By integrator of linear function (fmap case) -}
integrator coin >>= λh → 1 + integrator (if h then helper m else helper 3) f
 == {- By case analysis -}
integrator coin >>= λh → 1 + (if h then integrator (helper m) f else integrator (helper 3) f)
 == {- By linearity of integrator -}
1 + (integrator coin >>= λh → if h then integrator (helper m) f else integrator (helper 3) f)
 == {- By integrator/bernouilli (exercise) -}
1 + 0.5 * integrator (helper m) f + 0.5 * integrator (helper 3) f
```

If we let $h\ m = \mathit{expectedValueOfDistr}\ (\mathit{helper}\ m)$, and using the above lemma, then we find:

$$h\ (m+1) == 1 + 0.5 * h\ m + 0.5 * h\ 3$$

which we can rewrite as:

$$2 * h\ (m+1) = 2 + h\ m + h\ 3$$

and expand for $m = 0$ to $m = 2$:

$$2 * h\ 1 = 2 + h\ 3$$
$$2 * h\ 2 = 2 + h\ 1 + h\ 3$$
$$2 * h\ 3 = 2 + h\ 2 + h\ 3$$

This leaves us with a system of linear equations with three unknowns $h\ 1$, $h\ 2$ and $h\ 3$, which admits a single solution with $h\ 3 = 14$, giving the final solution to the initial problem.

9.12 Independent events

One possible way to define for independent events is as follows. E is independent from F iff $P(E \mid F) = P(E)$. We can express this definition in our DSL as follows:

$$\mathit{independentEvents} :: \mathit{Space}\ a \to (a \to \mathit{Bool}) \to (a \to \mathit{Bool}) \to \mathit{Bool}$$
$$\mathit{independentEvents}\ s\ e\ f = \mathit{probability1}\ s\ e == \mathit{condProb}\ s\ e\ f$$

According to Grinstead and Snell [2003], two events independent iff. $P(E \cap F) = P(E) \cdot P(F)$.

Using our language, we would write instead:

$$\mathit{probability1}\ s\ (\lambda x \to e\ x \wedge f\ e) == \mathit{probability1}\ s\ e * \mathit{probability1}\ s\ f$$

Proof.

In the left to right direction:

$$\begin{aligned} & P\ (E \cap F) \\ = & \text{ \{- by def. of cond. prob -\}} \\ & P\ (E \mid F) \cdot P\ (F) \\ = & \text{ \{- by def. of independent events -\}} \\ & P\ (E) \cdot P\ (F) \end{aligned}$$

This part of the proof is written like so using our DSL:

$$
\begin{aligned}
&\mathit{probability1}\ s\ (\lambda x \to e\ x \wedge f\ e) \\
&= \mathit{condProb}\ s\ e\ f * \mathit{probability1}\ s\ f \\
&= \mathit{probability1}\ s\ e * \mathit{probability1}\ s\ f
\end{aligned}
$$

We note that at this level of abstraction, the proofs follow the same structure as the textbook proofs — the underlying space s is constant.

In the right to left direction:

$$
\begin{aligned}
&P\ (E \mid F) \\
&= \{\text{- by def. of cond. prob -}\} \\
&P\ (E \cap F)\ /\ P\ (F) \\
&= \{\text{- by assumption -}\} \\
&P\ (E) \cdot P\ (F)\ /\ P\ (F) \\
&= \{\text{- by computation -}\} \\
&P\ (E)
\end{aligned}
$$

Exercise: express the rest of the proof using our DSL

TODO: In addition to a semantics based on integrators, one can program a semantics based on monte carlo sampling

TODO: Some related work.

9.13 End

TODO: sum up and close

Some guiding ideas, as bullet points: - textbooks are often written with the goal of taking you directly to an intuitive understanding of the topic. This often means skipping through a description of the domain from-the-ground up This may work for you, but if it does not, we encourage using the skills learned in this book. - use an typeable/executable/testable (formal) language (for example Haskell) to check your understanding. Fill any gap left open. - algebraic structures, and morphisms between them, is key to drawing generic patterns between various fields of math. When you figure out such generic patterns you can transpose knowledge from one domain to the other. Sometimes the existence of a morphism completely fixes the kind of definitions that one can use (e.g. derivatives, vector spaces). - syntax and semantics are often different but related objects for a given domain. Pay attention to which is which. - When checking proofs, it is often fruitful to imagine them as programs taking you from hypothesis to conclusions. Make sure that you can follow the proof threads, as if it were dataflow path.

Chapter 1: Haskell-intro, Types, Functions, Complex numbers, $\mathit{eval} : \mathit{syntax} \to \mathit{semantics}$

Chapter 2: Logic, proofs, specifications, laws, predicate logic, FOL, $\forall\ x.\ \exists\ y.\ \ldots$

Chapter 3: Types in Mathematics, derivatives, Lagrange equations (case study)

Chapter 4: $\mathit{eval} : \mathit{FunExp} \to (\mathbb{R} \to \mathbb{R})$, eval', evalD

Chapter 5: Polynomial functions, Homomorphism / Algebra / Monoid / Ring

Chapter 6: Homomorphisms / Formal Power Series

Chapter 7: Linear algebra, vector spaces, matrices, bases

Chapter 8: *exp*, *Laplace*

You have reached the end — rejoice!

Appendix A

Parameterised Complex Numbers

```
module DSLsofMath.CSem where
newtype ComplexSem r = CS (r, r) deriving Eq
```

Lifting operations to a parameterised type When we define addition on complex numbers (represented as pairs of real and imaginary components) we can do that for any underlying type r which supports addition.

```
type CS = ComplexSem    -- for shorter type expressions below
liftCS :: (     r →      r →      r) →
          (CS r → CS r → CS r)
liftCS (+) (CS (x, y)) (CS (x', y')) = CS (x + x', y + y')
```

Note that *liftCS* takes $(+)$ as its first parameter and uses it twice on the RHS.

```
re :: ComplexSem r  → r
re z@(CS (x, y))  = x
im :: ComplexSem r → r
im z@(CS (x, y)) = y
(.+.) :: Num r ⇒ ComplexSem r → ComplexSem r → ComplexSem r
(CS (a, b)) .+. (CS (x, y)) = CS ((a + x), (b + y))
(.*.) :: Num r ⇒  ComplexSem r → ComplexSem r → ComplexSem r
CS (ar, a_i) .*. CS (br, b_i) = CS (ar * br − a_i * b_i, ar * b_i + a_i * br)
instance Show r ⇒ Show (ComplexSem r) where
  show = showCS
```

```
showCS :: Show r ⇒ ComplexSem r → String
showCS (CS (x,y)) = show x ++ " + " ++ show y ++ "i"
```

A corresponding syntax type: the second parameter *r* makes is possible to express "complex numbers over" different base types (like *Double*, *Float*, *Integer*, etc.).

```
data ComplexSy v r = Var v
                   | FromCart r r
                   | ComplexSy v r :++ ComplexSy v r
                   | ComplexSy v r :** ComplexSy v r
```

Bibliography

R. A. Adams and C. Essex. *Calculus: a complete course*. Pearson Canada, 7th edition, 2010.

J.-P. Bernardy and S. Chatzikyriakidis. A computational treatment of anaphora and its algorithmic implementation. *Journal of Logic, Language and Information*, 2020.

N. Botta, P. Jansson, and C. Ionescu. Contributions to a computational theory of policy advice and avoidability. *Journal of Functional Programming*, 27:1–52, 2017a. ISSN 0956-7968. doi: 10.1017/S0956796817000156.

N. Botta, P. Jansson, C. Ionescu, D. R. Christiansen, and E. Brady. Sequential decision problems, dependent types and generic solutions. *Logical Methods in Computer Science*, 13(1), 2017b. doi: 10.23638/LMCS-13(1:7)2017. URL https://doi.org/10.23638/LMCS-13(1:7)2017.

R. Boute. The decibel done right: a matter of engineering the math. *Antennas and Propagation Magazine, IEEE*, 51(6):177–184, 2009. doi: 10.1109/MAP.2009.5433137.

K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc. of the fifth ACM SIGPLAN international conference on Funct. Prog.*, pages 268–279. ACM, 2000.

K. Doets and J. van Eijck. *The Haskell Road to Logic, Maths and Programming*. Texts in computing. King's College Publications, London, 2004. ISBN 978-0-9543006-9-2. URL https://fldit-www.cs.uni-dortmund.de/~peter/PS07/HR.pdf.

C. H. Edwards, D. E. Penney, and D. Calvis. *Elementary Differential Equations*. Pearson Prentice Hall Upper Saddle River, NJ, 6h edition, 2008.

D. Gries and F. B. Schneider. *A logical approach to discrete math*. Springer, 1993. doi: 10.1007/978-1-4757-3837-7.

D. Gries and F. B. Schneider. Teaching math more effectively, through calculational proofs. *American Mathematical Monthly*, pages 691–697, 1995. doi: 10.2307/2974638.

C. M. Grinstead and J. L. Snell. *Introduction to Probability*. AMS, 2003. URL http://www.dartmouth.edu/~chance/teaching_aids/books_articles/probability_book/book.html.

C. Ionescu and P. Jansson. Dependently-typed programming in scientific computing: Examples from economic modelling. In R. Hinze, editor, *24th Symposium on Implementation and Application of Functional Languages (IFL 2012)*, volume 8241 of *LNCS*, pages 140–156. Springer-Verlag, 2013a. doi: 10.1007/978-3-642-41582-1_9.

C. Ionescu and P. Jansson. Dependently-typed programming in scientific computing. In *Implementation and Application of Functional Languages*, pages 140–156. Springer Berlin Heidelberg, 2013b. doi: 10.1007/978-3-642-41582-1_9.

C. Ionescu and P. Jansson. Domain-specific languages of mathematics: Presenting mathematical analysis using functional programming. In J. Jeuring and J. McCarthy, editors, *Proceedings of the 4th and 5th International Workshop on Trends in Functional Programming in Education, Sophia-Antipolis, France and University of Maryland College Park, USA, 2nd June 2015 and 7th June 2016*, volume 230 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–15. Open Publishing Association, 2016. doi: 10.4204/EPTCS.230.1.

C. Jaeger, P. Jansson, S. van der Leeuw, M. Resch, and J. D. Tabara. GSS: Towards a research program for Global Systems Science. http://blog.global-systems-science.eu/?p=1512, 2013. ISBN 978.3.94.1663-12-1. Conference Version, prepared for the Second Open Global Systems Science Conference June 10-12, 2013, Brussels.

P. Jansson, S. H. Einarsdóttir, and C. Ionescu. Examples and results from a bsc-level course on domain-specific languages of mathematics. In *Proc. 7th Int. Workshop on Trends in Functional Programming in Education*, volume 295 of *EPTCS*, page 79–90. Open Publishing Association, 2019. doi: 10.4204/eptcs.295.6. URL http://dx.doi.org/10.4204/EPTCS.295.6. Presented at TFPIE 2018.

R. Kraft. Functions and parameterizations as objects to think with. In *Maple Summer Workshop, July 2004, Wilfrid Laurier University, Waterloo, Ontario, Canada*, 2004.

E. Landau. *Einführung in die Differentialrechnung und Integralrechnung*. Noordhoff, 1934.

E. Landau. *Differential and Integral Calculus*. AMS/Chelsea Publication Series. AMS Chelsea Pub., 2001.

D. Lincke, P. Jansson, M. Zalewski, and C. Ionescu. Generic libraries in C++ with concepts from high-level domain descriptions in Haskell: A DSL for computational vulnerability assessment. In *IFIP Working Conf. on Domain-Specific Languages*, volume 5658/2009 of *LNCS*, pages 236–261, 2009. doi: 10.1007/978-3-642-03034-5_12.

S. Mac Lane. *Mathematics: Form and function*. Springer New York, 1986.

M. D. McIlroy. Functional pearl: Power series, power serious. *J. of Functional Programming*, 9:323–335, 1999. doi: 10.1017/S0956796899003299.

T. J. Quinn and S. Rai. Discovering the laplace transform in undergraduate differential equations. *PRIMUS*, 18(4):309–324, 2008.

J. J. Rotman. *A first course in abstract algebra*. Pearson Prentice Hall, 2006.

W. Rudin. *Principles of mathematical analysis*, volume 3. McGraw-Hill New York, 1964.

W. Rudin. *Real and complex analysis*. Tata McGraw-Hill Education, 1987.

D. Stirzaker. *Elementary Probability*. Cambridge University Press, 2 edition, 2003. doi: 10.1017/CBO9780511755309.

J. Tolvanen. Industrial experiences on using DSLs in embedded software development. In *Proceedings of Embedded Software Engineering Kongress (Tagungsband), December 2011*, 2011. doi: 10.1.1.700.1924.

C. Wells. Communicating mathematics: Useful ideas from computer science. *American Mathematical Monthly*, pages 397–408, 1995. doi: 10.2307/2975030.