Domain Specific Languages of Mathematics: Lecture Notes

Patrik Jansson Cezar Ionescu

January 20, 2018

Abstract

These notes aim to cover the lectures and exercises of the recently introduced course "Domain-Specific Languages of Mathematics" (at Chalmers and University of Gothenburg). The course was developed in response to difficulties faced by third-year computer science students in learning and applying classical mathematics (mainly real and complex analysis). The main idea 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 Introduction

These lecture notes aim to cover the lectures and exercises of the recently introduced BSc-level course "Domain Specific Languages of Mathematics" (at Chalmers Univeristy of Technology and University of Gothenburg). The immediate aim of the course is to improve the mathematical education of computer scientists and the computer science education of mathematicians. We believe the course can be the starting point for far-reaching changes, leading to a restructuring of the mathematical training especially for engineers, 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, this 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 increasingly see the 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. Since, moreover, discrete mathematics tends to be immediately useful to them, this makes the added difficulty of continuous mathematics 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. For example, a course in "Modeling of sustainable energy systems" for Chalmers' CSE[1] students has to be tailored around this limitation, meaning that the models, methods, and tools that can be presented need to be drastically simplified, to the point where contact with mainstream research becomes impossible.

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 (see, for example, 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 combine this with 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"

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

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.

The course on *DSLs of Mathematics (DSLM)* allows us to present 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, used to a more informal style, the increased formality is justified by the need to implement (fragments of) the language. 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.

The course has 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 (Domain Specific Language) 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 DSLsofMath

In the first instances, the course is an elective course for the second year within programmes such as CSE, 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 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 plan 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 example, for CS and CSE students we will measure the percentage of students who, having taken DSLM, pass the third-year courses *Transforms, signals and systems* and *Control Theory (Reglerteknik)*, which are current major stumbling blocks. For math students, we would like to measure their performance in ulterior scientific computing courses.

Since the course is, at least initially, an elective one, we also have the possibility of comparing the results with those of a control group (students who have not taken the course).

The work that lead up to the current course is as follows:

2014: in interaction with our colleagues from the various study programmes, we performed an assessment of the current status of potential students for the course in terms of their training (what prerequisites we can reasonably assume) and future path (what mathematical fields they are likely to encounter in later studies), and we worked out a course plan (which we submitted in February 2015, so that the first instance of the course could start in January 2016). We also make a survey of similar courses being offered at other universities, but did not find any close matches.

2015: we developed course materials for use within the first instance, wrote a paper [Ionescu and Jansson, 2016] about the course and pressented the pedagogical ideas at several events (TFPIE'15, DSLDI'15, IFIP WG 2.1 #73 in Göteborg, LiVe4CS in Glasgow).

2016: we ran the first instance of DSLM (partly paid by the regular course budget, partly by this project) with Cezar Ionescu as main lecturer.

2017: we ran the second instance of DSLM (paid fully by the regular course budget), now with Patrik Jansson as main lecturer.

2016 and 2017: we used the feedback from students following the standard Chalmers course evaluation in order to improve and further develop the course material.

Future work includes involving faculty from CSE and mathematics in the development of other mathematics courses (prel. Linear Algebra, Analysis) 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 Mathematical Analysis 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.

0.1 About this course

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 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 topic

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).

0.2 Who should read these lecture notes?

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.

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.

0.3 Notation and code convention

Each chapter ends with exercises to help the reader practice the concepts just taught. Most exam questions from the first five exams of the DSLsofMath course have been included as exercises, so for those of you taking the course, you can check your progress towards the final examination. Sometimes the chapter text contains short, inlined questions, like "Exercice 1.13: 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.

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.

1 A DSL for arithmetic expressions and complex numbers

This chapter is partly based on the paper [Ionescu and Jansson, 2016] from the International Workshop on Trends in Functional Programming in Education 2015. 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:

```
module DSLsofMath.W01 where
import qualified DSLsofMath.CSem as CSem
import DSLsofMath.CSem (ComplexSem (CS))
import Numeric.Natural (Natural)
import Data.Ratio (Rational, Ratio, (%))
import Data.List (find)
```

1.1 Intro: Pitfalls with traditional mathematical notation

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 or as a function of x.

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

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.) 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? 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". Mathematics text books usually avoid the risk of confusion by (silently) renaming variables when needed, but we believe this renaming is a sufficiently important operation to be more explicitly mentioned.

Variable names as type hints In mathematical texts there are often conventions about the names used for variables of certain types. Typical examples include 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

> **type** $T = \mathit{Real}$
> **type** $S = \mathbb{C}$
> $\mathcal{L} : (T \to \mathbb{C}) \to (S \to \mathbb{C})$

1.2 Types of data

Dividing up the world (or problem domain) into values of different types is one of the guiding principles of this course. We will see that keeping track of types can guide the development of theories, languages, programs and proofs. To start out we introduce some of the ways types are defined in Haskell, the language we use for implementation (and often also specification) of mathematical concepts.

1.2.1 type / newtype / 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.

> **type** $\mathit{Heltal} = \mathit{Integer}$
> **type** $\mathit{Foo} = (\mathit{Maybe}\ [\mathit{String}], [[\mathit{Heltal}]])$
> **type** $\mathit{BinOp} = \mathit{Heltal} \to \mathit{Heltal} \to \mathit{Heltal}$
> **type** $\mathit{Env}\ v\ s = [(v, s)]$

The new name for the type on the RHS does not add type safety, just readability (if used wisely). The *Env* example shows that a type synonym can have type parameters.

newtype – more protection A simple example of the use of **newtype** in Haskell is to distinguish values which should be kept apart. A simple example is

> **newtype** $\mathit{Age} = \mathit{Ag}\ \mathit{Int}$ -- Age in years
> **newtype** $\mathit{Shoe} = \mathit{Sh}\ \mathit{Int}$ -- Shoe size (EU)

Which introduces two new types, *Age* and *Shoe*, which both are internally represented by an *Int* but which are good to keep apart.

The constructor functions $Ag :: Int \to Age$ and $Sh :: Int \to Shoe$ are used to translate from plain integers to ages and shoe sizes.

In the lecture notes we used a newtype for the semantics of complex numbers as a pair of numbers in the cartesian representation but 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.

Examples values: $zero = Z$, $one = S\ Z$, $three = S\ (S\ one)$

The **data** keyword will be used throughout the course to define datatypes of syntax trees for different kinds of expressions: simple arithmetic expresssions, complex number expresssions, etc. But it can also be used for non-recursive datatypes, like **data** $Bool = False \mid True$, or **data** $Person = P\ String\ Age\ Shoe$. 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 *Person* type is an example of a *product type*, where each value uses the same constructor P and records values for the name, age, and shoe size of the person modelled. (See exercise 1.11 for the intuition behind the terms "sum" and "product" used here.)

Maybe **and parameterised types.** It is very often possible describe a family of types of the same "shape". One simple example is the type constructor *Maybe*:

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

This declaration introduces

- 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 a function may, or may not, return a value.

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).

> **data** $Either\ p\ q = Left\ p \mid Right\ q$

1.2.2 *Env* and variable *lookup*.

The type synonym

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

is one way of expressing a partial function from v to s. As an example value of this type we can take:

```
env1 :: Env String Int
env1 = [("hej", 17), ("du", 38)]
```

We can see the type *Env v s* as a syntactic representation of a partial function from v to s. We can convert to a total function *Maybe* returning an s using *evalEnv*:

```
evalEnv :: Eq v => Env v s -> (v -> Maybe s)
evalEnv vss var = findFst vss
  where findFst ((v, s) : vss)
            | var == v    = Just s
            | otherwise = findFst vss
        findFst []        = Nothing
```

Or we can use the Haskell prelude function *lookup* = *flip evalEnv*:

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

We will use *Env* and *lookup* below (in section 1.3) when we introduce abstract syntax trees containing variables.

1.3 A syntax for simple arithmetical expressions

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

This declaration introduces

- 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: $x = V$ `"x"`, $e_1 = P\ x\ x$, $e_2 = T\ e_1\ e_1$

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

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

Example values: $y = V$ `"y"`, $e_1 = y :+ y$, $e_2 = x :* e_1$

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

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

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: 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 primes.

Together with a datatype for the syntax of arithmetic expressions we also want to define an evaluator of the expressions. The concept of "an evaluator", a function from the syntax to the semantics, is something we will return to many times in this book. We have already seen one example: the function *evalEnv* which translates from a list of key-value-pairs (the abstract syntax of the environment) to a function (the semantics).

In the evaluator for $AE'\ v$ we take this one step further: given an environment *env* and the syntax of an arithmetic expression e we compute the semantics of that expression.

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

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

1.4 A case study: complex numbers

We will start by an analytic reading of the introduction of complex numbers in 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. For this section, we bracket our previous knowledge and approach the text as we would a completely new domain, even if that leads to a somewhat exaggerated attention to detail.

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

> 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 value i, and, since we know nothing about other values, we make i the only member of this type:

$$\begin{aligned}&\textbf{data}\ \mathit{ImagUnits} = I\\ &i :: \mathit{ImagUnits}\\ &i = I\end{aligned}$$

We use a capital I in the **data** declaration because a lowercase constructor name would cause a syntax error in Haskell.

Next, we have 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:

$$\begin{aligned}\textbf{data}\ \mathit{ComplexA} = {} & \mathit{CPlus}_1\ \mathbb{R}\ \mathbb{R}\ \mathit{ImagUnits}\\ \mid {} & \mathit{CPlus}_2\ \mathbb{R}\ \mathit{ImagUnits}\ \mathbb{R}\end{aligned}$$

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

$$\begin{aligned}&\mathit{showCA} :: \mathit{ComplexA} \to \quad \mathit{String}\\ &\mathit{showCA}\quad (\mathit{CPlus}_1\ x\ y\ i) = \mathit{show}\ x +\!\!+ \texttt{" + "} +\!\!+ \mathit{show}\ y +\!\!+ \texttt{"i"}\\ &\mathit{showCA}\quad (\mathit{CPlus}_2\ x\ i\ y) = \mathit{show}\ x +\!\!+ \texttt{" + "} +\!\!+ \texttt{"i"} +\!\!+ \mathit{show}\ y\end{aligned}$$

Notice that the type $\mathbb{R}$ is not implemented yet and it is not really even exactly implementable but we want to focus on complex numbers so we will approximate $\mathbb{R}$ by double precision floating point numbers for now.

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

The text continues 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.

Mathematics		Haskell
$3 + 2i$		$CPlus_1\ 3\ 2\ I$
$\frac{7}{2} - \frac{2}{3}i$	$= \frac{7}{2} + \frac{-2}{3}i$	$CPlus_1\ (7\ /\ 2)\ (-2\ /\ 3)\ I$
$i\pi$	$= 0 + i\pi$	$CPlus_2\ 0\ I\ \pi$
-3	$= -3 + 0i$	$CPlus_1\ (-3)\ 0\ I$

Table 1: Examples of notation and abstract syntax for some complex numbers.

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.

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

$$toComplex :: \mathbb{R} \to ComplexA$$
$$toComplex\ x = CPlus_1\ x\ 0\ I$$

Again, at this stage there are many open questions. For example, we can assume that $i\ 1$ stands for the complex number $CPlus_2\ 0\ I\ 1$, but what about 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 $i\ 1$ and therefore $CPlus_2\ 0\ I\ 1$. But what about, say, $2\ i$? 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 by definition.

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:

$$\textbf{data}\ ComplexB = CPlusB\ \mathbb{R}\ \mathbb{R}\ ImagUnits$$

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

$$\textbf{data}\ ComplexC = CPlusC\ \mathbb{R}\ \mathbb{R}$$

(The renaming of the constructor to *CPlusC* serves as a guard against the case 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.) The equality on complex numbers is what we would obtain in Haskell by using **deriving** *Eq*.

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**:

> **newtype** $ComplexD = CD\ (\mathbb{R}, \mathbb{R})$ **deriving** Eq

The point of the somewhat confusing discussion of using "letters" to stand for complex numbers is to introduce a substitute for *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 $Re\ (z)$. We call y the **imaginary part** of z and denote it $Im\ (z)$:
>
> $$Re\ (z) = Re\ (x + yi) = x$$
> $$Im\ (z) = Im\ (x + yi) = y$$

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

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

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.

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
>
> $$w + z = (a + x) + (b + y)\ i$$
> $$w - z = (a - x) + (b - y)\ i$$

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:

$$\begin{aligned} &(+.) :: ComplexD \to ComplexD \to ComplexD \\ &(CD\ (a, b)) +. (CD\ (x, y)) = CD\ ((a + x), (b + y)) \end{aligned}$$

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.

Exercises:

- implement $(*.)$ for *ComplexD*

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. 2). (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.)

```
data      ImagUnits      = I
data      ComplexA       = CPlus1  ℝ ℝ ImagUnits
                         | CPlus2  ℝ ImagUnits ℝ
data      ComplexB       = CPlusB ℝ ℝ ImagUnits
data      ComplexC       = CPlusC ℝ ℝ
newtype   ComplexD       = CD (ℝ, ℝ) deriving Eq
newtype   ComplexSem r = CS (r, r)  deriving Eq
```

Figure 2: Complex number datatype refinement (semantics).

1.5 A syntax for (complex) arithmetical expressions

So far we have tried to find a datatype to represent the intended *semantics* of complex numbers. That approach is called "shallow embedding". Now we turn to 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:

```
evalE :: ComplexE → ComplexD
```

The datatype *ComplexE* should collect ways of building syntactic expression 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:

```
data ComplexE = ImagUnit
              | ToComplex ℝ
              | Plus    ComplexE ComplexE
              | Times ComplexE ComplexE
deriving (Eq, Show)
```

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

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

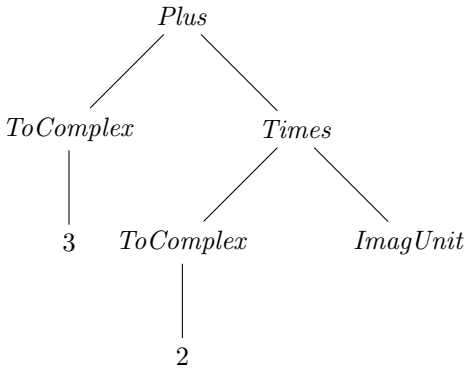
For example, when implementing the *evalE* (*Plus* c_1 c_2) case, you can assume that you already know the values $s_1, s_2 ::$ *ComplexD* corresponding to the subtrees c_1 and c_2. The only thing left is to add them up componentwise and we can assume there is a function $(+.) ::$ *ComplexD* $\rightarrow$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

```
evalE ImagUnit        = CD (0, 1)
evalE (ToComplex r) = CD (r, 0)
evalE (Plus c1 c2)    = evalE c1 +. evalE c2
evalE (Times c1 c2)   = evalE c1 *. evalE c2
```

We also define a function to embed a semantic complex number in the syntax:

```
fromCD :: ComplexD → ComplexE
fromCD (CD (x, y)) = Plus (ToComplex x) (Times (ToComplex y) ImagUnit)
testE1 = Plus (ToComplex 3) (Times (ToComplex 2) ImagUnit)
testE2 = Times ImagUnit ImagUnit
```

This is *testE1* as an abstract syntax tree:



1.6 Laws, properties and testing

There are certain laws 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 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 the lecture,

```
propImagUnit :: Bool
propImagUnit = Times ImagUnit ImagUnit === ToComplex (−1)
(===) :: ComplexE → ComplexE → Bool
z === w = evalE z == evalE w
```

and that *fromCD* is an embedding:

```
propFromCD :: ComplexD → Bool
propFromCD c = evalE (fromCD c) == c
```

but we also have that *Plus* and *Times* should be associative and commutative and *Times* should distribute over *Plus*:

```
propAssocPlus x y z       = Plus (Plus x y) z       === Plus x (Plus y z)
propAssocTimes x y z      = Times (Times x y) z     === Times x (Times y z)
propDistTimesPlus x y z = Times x (Plus y z)    === Plus (Times x y) (Times x z)
```

These three laws actually fail, but not because of the implementation of *evalE*. We will get back to that later but let us first generalise the properties a bit by making the operator a parameter:

```
propAssocA :: Eq a ⇒ (a → a → a) → a → a → a → Bool
propAssocA (+?) x y z = (x +? y) +? z == x +? (y +? z)
```

Note that *propAssocA* is a higher order function: it takes a function (a binary operator) as its first parameter. It 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 other binary operators. In Haskell there is a type class *Num* for different types of "numbers" (with operations (+), (∗), etc.). We can try out *propAssocA* for a few of them.

```
propAssocAInt = propAssocA (+) :: Int → Int → Int → Bool
propAssocADouble = propAssocA (+) :: Double → Double → Double → Bool
```

The first is fine, but the second fails due to rounding errors. QuickCheck can be used to find small examples - I like this one best:

```
notAssocEvidence :: (Double, Double, Double, Bool)
notAssocEvidence = (lhs, rhs, lhs − rhs, lhs == rhs)
  where lhs = (1 + 1) + 1 / 3
        rhs = 1 + (1 + 1 / 3)
```

For completeness: this is the answer:

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

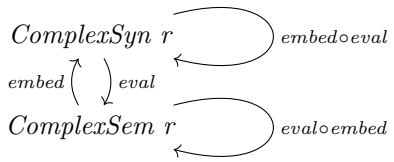
This is actually the underlying reason why some of the laws failed for complex numbers: the approximative nature of *Double*. But to be sure there is no other bug hiding we need to make one more version of the complex number type: parameterise on the underlying type for $\mathbb{R}$. At the same time we generalise *ToComplex* to *FromCartesian*:

```
data ComplexSyn r = FromCartesian r r
                  | ComplexSyn r :+: ComplexSyn r
                  | ComplexSyn r :∗: ComplexSyn r
toComplexSyn :: Num a ⇒ a → ComplexSyn a
toComplexSyn x = FromCartesian x 0
  -- From CSem in appendix A: newtype ComplexSem r = CS (r, r) deriving Eq
evalCSyn :: Num r ⇒ ComplexSyn r → CSem.ComplexSem r
evalCSyn (FromCartesian x y) = CS (x, y)
evalCSyn (l :+: r) = evalCSyn l +.^CSem evalCSyn r
evalCSyn (l :∗: r) = evalCSyn l ∗.^CSem evalCSyn r
instance Num a ⇒ Num (ComplexSyn a) where
  (+) = (:+:)
  (∗) = (:∗:)
  fromInteger = fromIntegerCS
    -- Exercise: add a few more operations (hint: extend ComplexSyn as well)
    -- Exercise: also extend eval
fromIntegerCS :: Num r ⇒ Integer → ComplexSyn r
fromIntegerCS = toComplexSyn ∘ fromInteger
```

From syntax to semantics and back We have seen evaluation functions from abstract syntax to semantics ($eval :: Syn \to Sem$). Often an inverse is also available: $embed :: Sem \to Syn$. For our complex numbers we have

$$embed :: CSem.ComplexSem\ r \to ComplexSyn\ r$$
$$embed\ (CS\ (x, y)) = FromCartesian\ x\ y$$

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



Exercise 1.14: What about the opposite direction? When is $embed\ (eval\ e) == e$?

More about laws Some laws appear over and over again in different mathematical contexts. Binary operators are often as 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:

$Associative\ (+) = \forall\ a, b, c.\ \ (a + b) + c = a + (b + c)$

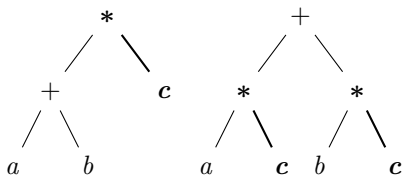
$Commutative\ (+) = \forall\ a, b.\ \ a + b = b + a$

Non-examples: division is not commutative, average is commutative but not associative.

$Distributive\ (*)\ (+) = \forall\ a, b, c.\ \ (a + b) * c = (a * c) + (b * c)$

We saw implementations of some of these laws as *propAssocA* and *propDistTimesPlus* earlier, and learnt that the underlying set matters: $(+)$ for $\mathbb{R}$ has some properties, but $(+)$ for *Double* has other. When implementing, approximation is often necessary, 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.).

To get a feeling for the distribution law, it can be helpful to study the syntax trees of the left and right hand sides. Note that $(*c)$ is pushed down (distributed) to both a and b:



(In the language of section 4.2.1, distributivity means that $(*c)$ is a $(+)$-homomorphism.)

Exercise: Find some operator $(\#)$ which satisfies $Distributive\ (+)\ (\#)$

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

1.7 More about functions

Function composition. The infix operator . (period) in Haskell is an implementation of the mathematical operation of function composition. The period is an ASCII approximation of the

composition symbol $\circ$ typically used in mathematics. (The symbol $\circ$ is encoded as `U+2218` and called RING OPERATOR in Unicode, `&#8728` in HTML, `\circ` in TEX, etc.) Its implementation is:

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

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

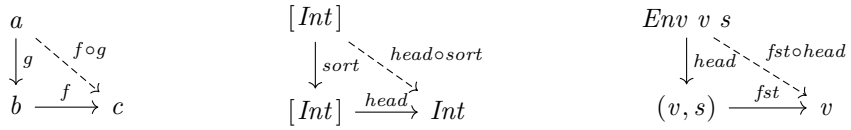


Figure 3: Function composition diagrams: in general, and two examples

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.

1.7.1 Notation and abstract syntax for (infinite) sequences

As a bit of preparation for the language of sequences and limits in later lectures we here spend a few lines on the notation and abstract syntax of sequences.

Common math book notation: $\{a_i\}_{i=0}^{\infty}$ 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 $a_i : X$. If we interpret "subscript" as function application we can see that $a : \mathbb{N} \to X$ is a useful typing of a sequence. Some examples:

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

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"):

$$\begin{array}{l}
\mathit{addSeq} :: \mathit{Num}\ a \Rightarrow \mathit{Seq}\ a \to \mathit{Seq}\ a \to \mathit{Seq}\ a \\
\mathit{addSeq}\ f\ g\ i = f\ i + g\ i
\end{array}$$

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

$$
\begin{aligned}
&\mathit{liftSeq2} :: (a \to b \to c) \to \mathit{Seq}\ a \to \mathit{Seq}\ b \to \mathit{Seq}\ c \\
&\mathit{liftSeq2}\ \mathit{op}\ f\ g\ i = \mathit{op}\ (f\ i)\ (g\ i) \quad \text{-- } \{\mathit{op}\ (f\ 0)\ (g\ 0), \mathit{op}\ (f\ 1)\ (g\ 1), ...\}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{liftSeq}_1 :: (a \to b) \to \mathit{Seq}\ a \to \mathit{Seq}\ b \\
&\mathit{liftSeq}_1\ h\ f\ i = h\ (f\ i) \quad \text{-- } \{h\ (f\ 0), h\ (f\ 1), h\ (f\ 2), ...\} \\
&\mathit{liftSeq0} :: a \to \mathit{Seq}\ a \\
&\mathit{liftSeq0}\ c\ i = c
\end{aligned}
$$

Exercice 1.13: what does function composition do to a sequence?

Another common mathematical operator on sequences is the limit. We will get back to limits in later chapters (2.11, 2.13), but here 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 x, may or may not have a limit. Thus the customary use of $L =$ is a bit of abuse of notation, because the right hand side may not be well defined. One way to capture that is to give *lim* the type $(\mathbb{N} \to X) \to \mathit{Maybe}\ X$. Then $L = \lim_{i\to\infty} x_i$ would mean $\mathit{Just}\ L = \mathit{lim}\ x$ We will return to limits and their proofs in Sec. 2.12 after we have reviewed some logic. Here we just define one more common operation: the sum of a sequence (like $\sigma = \sum_{i=0}^{\infty} 1/i!$). 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:

$$
\begin{aligned}
&\mathit{sums} :: \mathit{Num}\ a \Rightarrow \mathit{Seq}\ a \to \mathit{Seq}\ a \\
&\mathit{sums} = \mathit{scan}\ 0\ (+) \\
&\mathit{scan} :: a \to (a \to a \to a) \to \mathit{Seq}\ a \to \mathit{Seq}\ a \\
&\mathit{scan}\ z\ (+)\ f = s \\
&\quad \textbf{where}\ s\ 0 = z \\
&\qquad\quad s\ i = s\ (i-1) + f\ i
\end{aligned}
$$

And by combining this with limits we can state formally that the sum of a sequence a exists and is S iff the limit of *sums* a exists and is S. As a formula we get $\mathit{Just}\ S = \mathit{lim}\ (\mathit{sums}\ a)$, and for our example it turns out that it converges and that $\sigma = \sum_{i=0}^{\infty} 1/i! = e$ but we will not get to that until Sec. 8.1.

We will also return to limits in Sec. 3.3 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 ($\mathit{Seq}\ a$), some operations (*conSeq*, *addSeq*, *fmap*, *sums*, ...) and some "run functions" or predicates (like *lim* and *sum*).

1.8 Exercises: complex numbers and DSLs

Exercise 1.1. Consider the following data type for simple arithmetic expressions:

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

(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 $\mathit{eval} :: \mathit{Exp} \to \mathit{Integer}$ 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:

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

   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 *Var 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 $\mathit{varVal} :: \mathit{String} \to \mathit{Integer}$ 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* `"x"` 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 1.2. We will now look at a slightly more generalized version of the *Exp* type from the previous exercise:

$$
\begin{aligned}
&\textbf{data } \mathit{E2}\ a = \mathit{Con}\ a \\
&\qquad\quad \mid \mathit{Var}\ \mathit{String} \\
&\qquad\quad \mid \mathit{E2}\ a\ \text{`}\mathit{Plus}\text{`}\ \ \mathit{E2}\ a \\
&\qquad\quad \mid \mathit{E2}\ a\ \text{`}\mathit{Minus}\text{`}\ \mathit{E2}\ a \\
&\qquad\quad \mid \mathit{E2}\ a\ \text{`}\mathit{Times}\text{`}\ \mathit{E2}\ a \\
&\quad \textbf{deriving } (\mathit{Eq}, \mathit{Show})
\end{aligned}
$$

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 *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 row in the table.

   (a) Express the table above in Haskell by creating *vars* :: *Env String Double*.

   (b) Create a function *varVal* :: *Env a* → *String* → *a* that returns the value of a variable, given an *Env* and a variable name. For instance, *varVal vars* `"d"` should return 7.4

   (c) Create a function *eval* :: *Num a* ⇒ *Env a* → *E2* → *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 1.3. *From exam 2017-08-22*

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 $(R, +, 0)$:

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

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. Give a type signature for, and define, a general evaluator for *SR v* expressions on the basis of an assignment function.

Exercise 1.4. *From exam 2016-03-15*

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 1.5. *From exam 2016-08-23*

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.

Exercise 1.6. 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* `"z"`) :∗: *toComplex* 1.

Exercise 1.7. 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 1.8. 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 1.9. 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 1.10. 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 1.11. Counting values. Now assume we have *f2p*, *s2f*, etc used with three finite types 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 $Either\ a\ b$? (a, b)? $a \to b$? etc. These rules for computing the cardinality suggests that *Either* is similar to sum, $(,)$ is similar to product and $(\to)$ to (flipped) power. These rules show that we can use many intuitions from high-school algebra when working with types.

Exercise 1.12. 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 1.13. From section 1.7.1:

- What does function composition do to a sequence? (composition on the left?, on the right?)
- How is $liftSeq_1$ related to *fmap*? *liftSeq0* to *conSeq*?

Exercise 1.14. 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?

2 Logic and calculational proofs

The learning outcomes of this chapter is "develop adequate notation for mathematical concepts" and "perform calculational proofs" (still in the context of "organize areas of mathematics in DSL terms").

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.

```
module DSLsofMath.W02 where
import qualified DSLsofMath.AbstractFOL as FOL
import DSLsofMath.AbstractFOL (andIntro, andElimR, andElimL, notIntro, notElim)
```

2.1 Propositional Calculus

(Swedish: Satslogik[2])

Now we turn to the main topic of this chapter: logic and proofs. Our first DSL for this chapter is the language of *propositional calculus* (or logic), modelling simple propositions with the usual combinators for and, or, implies, etc. The syntactic constructs are collected in Table 2.

a, b, c, ...	names of propositions
False, *True*	Constants
And	$\wedge$ &
Or	$\vee$ |
Implies	$\Rightarrow$
Not	$\neg$

Table 2: Syntax for propositions

Some example propositions: $p_1 = a \wedge (\neg a)$, $p_2 = a \Rightarrow b$, $p_3 = a \vee (\neg a)$, $p_4 = (a \wedge b) \Rightarrow (b \wedge a)$. If we assign all combinations of truth values for the names, we can compute a truth value of the whole proposition. In our examples, p_1 is always false, p_2 is mixed and p_3 and p_4 are always true.

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 PropCalc = Con     Bool
              | Name    String
              | And     PropCalc PropCalc
              | Or      PropCalc PropCalc
              | Implies PropCalc PropCalc
              | Not     PropCalc
```

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"
```

From this datatype we can write an evaluator to *Bool* which computes the truth value of a term given an environment:

[2]Some Swe-Eng translations are collected here: `https://github.com/DSLsofMath/DSLsofMath/wiki/Translations-for-mathematical-terms`.

```
type Name = String
evalPC :: (Name → Bool) → PropCalc → Bool
evalPC = error "Exercise"   -- see 2.14 for a similar function
```

The function *evalPC* translates from the syntactic to the semantic domain. (The evaluation function for a DSL describing a logic is often called *check* instead of *eval* but here we stick to *eval*.) Here *PropCalc* is the (abstract) *syntax* of the language of propositional calculus and *Bool* is the *semantic domain*.

Alternatively, we can view $(Name \to Bool) \to Bool$ as the semantic domain. A value of this type is a mapping from a truth table (for the names) to *Bool*. This mapping is often also tabulated as a truth table with one more "output" column.

As a first example of a truth table, consider the proposition $t = Implies\ (Con\ False)\ a$. We will use the shorter notation with just T for true and F for false. The truth table semantics of t is usually drawn as follows: one column for the name a listing all combinations of T and F, and one column for the result of evaluating the expression.

a	t
F	T
T	T

This table shows that no matter what value assignment we try for the only variable a, the semantic value is $T = True$. Thus the whole expression could be simplified to just T without changing the semantics.

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. The & columns become $F\ F\ F\ T$ and finally the $\Rightarrow$ column (the output) becomes true everywhere.

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

A proposition whose truth table output is constantly true is called a *tautology*. Thus both t and p_4 are tautologies. 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 cases for n names. (There are very good heuristic algorithms to look for tautologies even for thousands of names — but that is not part of this course.)

What we call "names" are often called "(propositional) variables" but we will soon add another kind of variables (and quantification over them) to the calculus.

2.2 First Order Logic (predicate logic)

(Swedish: Första ordningens logik = predikatlogik)

Our second DSL is that of *First Order Logic (FOL)*. This language has two datatypes: propositions, and *terms* (new). 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 actual function symbols are usually domain specific — we can use rational number expressions as an example. In this case we can model the terms as a datatype:

```
data RatT = RV String | FromI Integer | RPlus RatT RatT | RDiv RatT RatT
  deriving Show
```

This introduces variables and three function symbols: *FromI* of arity 1, *RPlus*, *RDiv* of arity 2.

The propositions from *PropCalc* are extended so that they can refer to terms. We will normally refer to a *FOL* proposition as a *formula*. 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 N of arity 0, P of arity 1 and Q of arity 2 we can form *formulas* like N, $P\ (x)$, $Q\ (f\ (x, x), y)$, etc. Note that we have two separate layers: formulas normally refer to terms, but terms cannot refer to formulas.

The formulas introduced so far are all *atomic formulas* but 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" ($\exists$).

An example FOL formula:

$$\forall\ x.\ \ P\ (x) \Rightarrow (\exists\ y.\ \ Q\ (f\ (x, x), y))$$

Note that FOL can only quantify over *term* variables, not over predicates. (Second order logic and higher order logic allow quantification over predicates.)

Another example: a formula stating that function symbol *plus* is commutative:

$$\forall\ x.\ \ \forall\ y.\ \ Eq\ (plus\ (x, y), plus\ (y, x))$$

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 *P* `"Eq"` but is often included.

```
data FOL = P String [RatT]
         | Equal RatT RatT
         | And     FOL FOL
         | Or      FOL FOL
         | Implies FOL FOL
         | Not     FOL
         | FORALL String FOL
         | EXISTS  String FOL
  deriving Show
commPlus :: FOL
commPlus = FORALL "x" (FORALL "y" (Equal (RPlus (RV "x") (RV "y"))
                                         (RPlus (RV "y") (RV "x"))))
```

Quantifiers: meaning, proof and syntax. "Forall"-quantification can be seen as a generalisation of *And*. First we can generalise the binary operator to an n-ary version: And_n. To prove $And_n\ (A_1, A_2, ..., A_n)$ we need a proof of each A_i. Thus we could define $And_n\ (A_1, A_2, ..., A_n) = A_1\ \&\ A_2\ \&\ ...\ \&\ A_n$ where & is the infix version of binary *And*. The next step is to note that the formulas A_i can be generalised to $A\ (i)$ where i is a term variable and A is a unary predicate symbol. We can think of i ranging over an infinite collection of constant terms i_0, i_1, ... 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. The standard procedure is to introduce a fresh 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 terms to proofs such that $f\ t$ is a proof of $P\ t$ for each term t.

Note that the syntactic 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). The scoping of x in $\exists\ x.\ \ b$ is the same as in $\forall\ x.\ \ b$.

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

2.3 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). In this section we keep the predicate part of the version of *FOL* from the previous section, but we replace the term language *RatT* with pure (untyped) set theory.

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.

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. (Yes, this can make your head spin.)

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:

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

If we use conventional set notation we get $\mathit{vN}\ 0 = \{\,\}$, $\mathit{vN}\ 1 = \{\{\,\}\}$, $\mathit{vN}\ 2 = \{\{\,\},\{\{\,\}\}\}$, $\mathit{vN}\ 3 = \{\{\,\},\{\{\,\}\},\{\{\,\},\{\{\,\}\}\}\}$, etc. If we use the shorthand $\overline{n}$ for $\mathit{vN}\ n$ 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). The function vN is explored in more detail in the first assignment of the DSLsofMath course.

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: $\{a, b\} \stackrel{\text{def}}{=} \mathit{Union}\ (S\ a)\ (S\ b)$ 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].

2.4 Back to quantifiers

After this detour through untyped set land, let us get back to the most powerful concept of FOL: the quantifiers. We have already seen how the "forall" quantifier can be seen as a generalisation of *And* and in the same way we can see the "exists" quantifier as a generalisation of *Or*.

First we generalise the binary *Or* to an n-ary Or_n. To prove $Or_n\ A_1\ A_2\ ...\ A_n$ 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 one unary predicate A expressing the formulas $A\ (i)$ for the term variable i. Then the final step is to "or" all these formulas to obtain $\exists\ i.\ \ A\ i$.

At this point it is good to sum up and compare the two quantifiers and how to prove them:

(t, b_t) is a proof of $\exists\ x.\ \ P\ (x)$ if b_t is a proof of $P\ (t)$.

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

Curry-Howard If we abbreviate "is a proof" as : and use the Haskell convention for function application we get

$$\begin{array}{lll} (t, b_t) : (\exists\ x.\ \ P\ x) & \textbf{if} & b_t\ : P\ t \\ f\ \ \ \ \ \ : (\forall\ x.\ \ P\ x) & \textbf{if} & f\ t : P\ t\ \text{ for all } t \end{array}$$

This now very much looks like type rules, and that is not a coincidence. The *Curry-Howard correspondence* says that we can think of propositions as types and proofs as "programs". These typing judgements are not part of FOL, but the correspondence is used quite a bit in this course to keep track of proofs.

We can also interpret the simpler binary connectives using the Curry-Howard correspondence. A proof of $\mathit{And}\ P\ Q$ is a pair of a proof of P and a proof of Q. Or, as terms: if $p : P$ and $q : Q$ then $(p, q) : \mathit{And}\ P\ Q$. Similarly, a proof of $\mathit{Or}\ P\ Q$ is either a proof of P or a proof of Q: we can pick the left (P) or the right (Q) using the Haskell datatype *Either*: if $p : P$ then $\mathit{Left}\ p : \mathit{Or}\ P\ Q$ and if $q : Q$ then $\mathit{Right}\ q : \mathit{Or}\ P\ Q$. In this way we can build up what is called "proof terms" for a large fragment of logic. It turns out that each such proof term is basically a program in a functional programming language, and that the formula a certain term proves is the type for the program.

Typed quantification In each instance of FOL, quantification is always over the full set of terms, but it is often convenient to quantify over a subset with a certain property (like all even numbers, or all non-empty sets). We will use a notation we can call "typed quantification" as a short-hand notation for the full quantification in combination with a restriction to the subset. For existential and universal quantification these are the definitions (assuming T is a property of terms, that is a unary predicate on terms):

$$(\exists\ x : T.\ \ P\ x) \stackrel{\text{def}}{=} (\exists\ x.\ \ T\ x\ \&\ P\ x)$$
$$(\forall\ x : T.\ \ P\ x) \stackrel{\text{def}}{=} (\forall\ x.\ \ T\ x \Rightarrow P\ x)$$

A good exercise is to work out the rules for "pushing negation through" typed quantification, from the corresponding rules for full quantification.

2.5 Proof by contradiction

Let's try to 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 (for positive real numbers) can be specified by $r = \sqrt{s}$ iff $r\hat{}2 == s$ and $r : \mathbb{R}$. The formula "x is irrational" is just $\neg\ (R\ x)$ where R is the predicate "is rational".

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

The classical way to prove a negation $\neg\ P$ is to assume P and derive something absurd (some Q and $\neg\ Q$, for example). Lets 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 $GCD\ (a, b) \geqslant 2$ which implies $\neg\ Q$.

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

2.6 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? We can reason about two possible cases, one of which has to hold: $R\ x$ or $\neg\ (R\ x)$.

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!

2.7 Functions as proofs

To prove a formula $P \Rightarrow Q$ we assume a proof $p : P$ and derive a proof $q : Q$. Such a proof can be expressed as $(\lambda p \rightarrow q) : (P \Rightarrow Q)$: a proof of an implication is a function from proofs to proofs.

As we saw earlier, a similar rule holds for the "forall" quantifier: a function f from terms t to proofs of $P\ t$ is a proof of $\forall\ x.\ \ P\ x$.

```
module DSLsofMath.AbstractFOL where
data And  p q;                    data Or  p q
data Impl p q;                    data Not p
andIntro  :: p → q → And p q;     orElim  :: Or p q → (p → r) → (q → r) → r
andElimL :: And p q → p;          orIntroL :: p → Or p q
andElimR :: And p q → q;          orIntroR :: q → Or p q
implIntro :: (p → q) → Impl p q;  notIntro :: (p → And q (Not q)) → Not p
implElim  :: Impl p q → p → q;    notElim  :: Not (Not p) → p
andIntro = u; orElim = u; andElimR = u; orIntroL = u; andElimL = u; orIntroR = u;
implIntro = u; notElim = u; notIntro = u; implElim = u; u = undefined;
```

Figure 4: The Haskell module *AbstractFOL*.

As we saw in section 2.4, a very common kind of formula is "typed quantification": if a type (a set) S of terms can be described as those that satisfy the unary predicate T we can introduce the short-hand notation

$$(\forall\ x : T.\ \ P\ x) = (\forall\ x.\ \ T\ x \Rightarrow P\ x)$$

A proof of this is a two-argument function p which takes a term t and a proof of $T\ t$ to a proof of $P\ t$.

In pseudo-Haskell we can express the implication laws as follows:

$$impIntro : (A \to B) \to (A \Rightarrow B)$$
$$impElim : (A \Rightarrow B) \to (A \to B)$$

It should come as no surprise that this "API" can be implemented by $(\Rightarrow) = (\to)$, which means that both *impIntro* and *impElim* can be implemented as *id*.

Similarly we can express the universal quantification laws as:

$$\forall\text{-}Intro : ((a : Term) \to P\ a) \to (\forall\ x.\ \ P\ x)$$
$$\forall\text{-}Elim : (\forall\ x.\ \ P\ x) \to ((a : Term) \to P\ a)$$

To actually implement this we need a *dependent* function type, which Haskell does not provide. But we can still use it as a tool for understanding and working with logic formulas and mathematical proofs.

Haskell supports limited forms of dependent types and more is coming every year but for proper dependently typed programming I recommend the language Agda.

2.8 Proofs for *And* and *Or*

When formally proving properties in FOL we should use the introduction and elimination rules. The propositional fragment of FOL is given by the rules for $\wedge$, $\to$, $\longleftrightarrow$, $\neg$, $\vee$. We can use the Haskell type checker to check proofs in this fragment, using the functional models for introduction and elimination rules. Examine Fig. 4 (also available in the file `AbstractFOL.lhs`), which introduces an empty datatype for every connective (except $\longleftrightarrow$), and corresponding types for the introduction and elimination rules. The introduction and elimination rules are explicitly left undefined, but we can still combine them and type check the results. For example:

```
example0 :: FOL.And p q → FOL.And q p
example0 evApq = andIntro (andElimR evApq) (andElimL evApq)
```

(The variable name *evApq* is a mnemonic for "evidence of *And p q*".)

Notice that Haskell will not accept

```
example0 evApq = andIntro (andElimL evApq) (andElimR evApq)
```

unless we change the type.

Another example:

```
example1 :: FOL.And q (FOL.Not q) → p
example1 evAqnq = notElim (notIntro (λhyp → evAqnq))
```

To sum up the *And* case we have one introduction and two elimination rules:

```
andIntro  :: p → q → And p q
andElimL :: And p q → p
andElimR :: And p q → q
```

If we see these introduction and elimination rules as an API, what would be a reasonable implementation of the datatype *And p q*? A type of pairs! Then we see that the corresponding Haskell functions would be

```
pair :: p → q → (p, q)   -- andIntro
fst  :: (p, q) → p       -- andElimL
snd  :: (p, q) → q       -- andElimR
```

Revisiting the tupling transform In exercise 1.9, the "tupling transform" was introduced, relating a pair of functions to a function returning a pair. (Please revisit that exercise if you skipped it before.) There is a logic formula corresponding to the type of the tupling transform:

$$(a \Rightarrow (b \,\&\, c)) \Leftrightarrow (a \Rightarrow b) \,\&\, (a \Rightarrow c)$$

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 :: Impl (Impl a (And b c)) (And (Impl a b) (Impl a c))
test1 = implIntro (λa2bc →
          andIntro (implIntro (λa → andElimL (implElim a2bc a)))
                   (implIntro (λa → andElimR (implElim a2bc a))))
test2 :: Impl (And (Impl a b) (Impl a c)) (Impl a (And b c))
test2 = implIntro (λfg →
          implIntro (λa →
            andIntro
              (implElim (andElimL fg) a)
              (implElim (andElimR fg) a)))
```

Or **is the dual of** *And*. Most of the properties of *And* have corresponding properties for *Or*. Often it is enough to simply swap the direction of the "arrows" (implications) and swap the role between introduction and elimination.

$$\begin{aligned} &\mathit{orIntroL} : P \to (P \mid Q) \\ &\mathit{orIntroR} : Q \to (P \mid Q) \\ &\mathit{orElim} \quad : (P \Rightarrow R) \to (Q \Rightarrow R) \to ((P \mid Q) \Rightarrow R) \end{aligned}$$

Here the implementation type can be a labelled sum type, also called disjoint union and in Haskell: *Either*.

2.9 Case study: there is always another prime

As an example of combining forall, exists and implication let us turn to one statement of the fact that there are infinitely many primes. If we assume 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:

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

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 first translate from logic to programs as described above. We can translate step by step, starting from the top level. The forall-quantifier translates to a (dependent) function type $(n : \mathit{Term}) \to$ and the implication to a normal function type $\mathit{Prime}\ n \to$. 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:

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

Now we can start filling in the definition of *proof* as a two-argument function returning a nested pair:

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

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

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

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? 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 $> n$. QED.

Note that the proof can be used to define a somewhat 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)
$7 \mapsto 71$ ...

2.10 Existential quantification as a pair type

We mentioned before that existential quantification can be seen as as a "big *Or*" of a family of formulas $P\ a$ for all terms a. This means that to prove the quantification, we only need exhibit one witness and one proof for that member of the family.

$$\exists\text{-}Intro : (a : Term) \to P\ a \to (\exists\ x.\ \ P\ x)$$

For binary *Or* the "family" only had two members, one labelled L for *Left* and one R for *Right*, and we used 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*.

$$orElim : (P \Rightarrow R) \to (Q \Rightarrow R) \to ((P \mid Q) \Rightarrow R)$$

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:

$$\exists\text{-}Elim : ((a : Term) \to P\ a \Rightarrow R) \to (\exists\ x.\ \ P\ x) \Rightarrow R$$

The datatype corresponding to $\exists\ x.\ \ P\ x$ is a pair of a witness a and a proof of $P\ a$. We sometimes write this type $(a : Term, P\ a)$.

2.11 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 we 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 $X : \mathscr{P}\ \mathbb{R}$. In general, the operator $\mathscr{P}$ takes a set (here $\mathbb{R}$) to the set of all its subsets.

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

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

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

Next: introduce the "open ball" function B.

$$B : \mathbb{R} \to \mathbb{R}_{>0} \to \mathscr{P}\ \mathbb{R}$$
$$B\ c\ r = \{\, x \mid |x - c| < r \,\}$$

$B\ c\ r$ 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

$$\begin{array}{l} getq\ \epsilon \mid \epsilon < 2 \qquad = 1 - \epsilon\ /\ 2 \\ \qquad\quad \mid otherwise = 17\ /\ 38 \end{array}$$

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:

$$\begin{array}{l} getq\ \epsilon = 1\ /\ getn\ \epsilon \\ getn\ \epsilon = 1 + ceiling\ (1\ /\ \epsilon) \end{array}$$

Exercise: 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 exercises in quantifier negation!

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

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 **let** $\epsilon = noLim\ getq$ **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 $p \notin xs$).

Exercise: 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 $Limp\ p\ X \Rightarrow Infinite\ X$.

2.12 The limit of a sequence

Now we can move from limit points to the more familiar limit of a sequence. At the core of DSLsofMath is the ability to analyse definitions from mathematical texts, and here we will use the definition of the limit of a sequence from 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 $a\_n - L < \varepsilon$.

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:

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

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

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

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

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

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: Prove that the limit of a sequence is unique.

Exercise: prove that $(a_1 \; haslim \; L_1) \;\&\; (a_2 \; haslim \; L_2)$ implies $(a_1 + a_2) \; haslim \; (L_1 + L_2)$.

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

2.13 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 + 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:

$$lim\ f\ a\ L = \forall\ \epsilon > 0.\ \ \exists\ \delta > 0.\ \ P\ \epsilon\ \delta$$

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:

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

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.)

2.14 Recap of syntax tres with variables, *Env* and *lookup*

This was frequently a source of confusion already the first week so there is already a question + answers earlier in this text. But here is an additional example to help clarify the matter.

$$\textbf{data}\ Rat\ v = RV\ v \mid FromI\ Integer \mid RPlus\ (Rat\ v)\ (Rat\ v) \mid RDiv\ (Rat\ v)\ (Rat\ v)$$
$$\textbf{deriving}\ (Eq, Show)$$

```
newtype RatSem = RSem (Integer, Integer)
```

We have a type *Rat v* for the syntax trees of rational number expressions (with variables of type *v*) and a type *RatSem* for the semantics of those rational number expressions as pairs of integers. The constructor $RV :: v \to Rat\ v$ is used to embed variables with names of type *v* in *Rat v*. We could use *String* instead of *v* but with a type parameter *v* we get more flexibility at the same time as we get better feedback from the type checker. Note that this type parameter serves a different purpose from the type parameter in 1.6.

To evaluate some $e :: Rat\ v$ we need to know how to evaluate the variables we encounter. What does "evaluate" mean for a variable? Well, it just means that we must be able to translate a variable name (of type *v*) to a semantic value (a rational number in this case). To "translate a name to a value" we can use a function (of type $v \to RatSem$) so we can give the following implementation of the evaluator:

```
evalRat1 :: (v -> RatSem) -> (Rat v -> RatSem)
evalRat1 ev (RV v)      = ev v
evalRat1 ev (FromI i)   = fromISem i
evalRat1 ev (RPlus l r) = plusSem (evalRat1 ev l) (evalRat1 ev r)
evalRat1 ev (RDiv l r)  = divSem  (evalRat1 ev l) (evalRat1 ev r)
```

Notice that we simply added a parameter *ev* for "evaluate variable" to the evaluator. The rest of the definition follows a common pattern: recursively translate each subexpression and apply the corresponding semantic operation to combine the results: *RPlus* is replaced by *plusSem*, etc.

```
fromISem :: Integer -> RatSem
fromISem i = RSem (i, 1)
plusSem :: RatSem -> RatSem -> RatSem
plusSem = undefined   -- TODO: exercise
  -- Division of rational numbers
divSem :: RatSem -> RatSem -> RatSem
divSem (RSem (a, b)) (RSem (c, d)) = RSem (a * d, b * c)
```

Often the first argument *ev* to the eval function is constructed from a list of pairs:

```
type Env v s = [(v, s)]
envToFun :: (Show v, Eq v) => Env v s -> (v -> s)
envToFun [] v = error ("envToFun: variable " ++ show v ++ " not found")
envToFun ((w, s) : env) v
  | w == v      = s
  | otherwise = envToFun env v
```

Thus, *Env v s* can be seen as an implementation of a "lookup table". It could also be implemented using hash tables or binary search trees, but efficiency is not the point here. Finally, with *envToFun* in our hands we can implement a second version of the evaluator:

```
evalRat2 :: (Show v, Eq v) => (Env v RatSem) -> (Rat v -> RatSem)
evalRat2 env e = evalRat1 (envToFun env) e
```

SET and PRED Several groups have had trouble grasping the difference between *SET* and *PRED*. This is understandable, because we have so far in the lectures mostly talked about term syntax + semantics, and not so much about predicate syntax and semantics. The one example of terms + predicates covered in the lectures is Predicate Logic and I never actually showed how eval (for the expressions) and check (for the predicates) is implemented.

As an example we can we take our terms to be the rational number expressions defined above and define a type of predicates over those terms:

$$
\begin{aligned}
&\textbf{type } \mathit{Term}\ v = \mathit{Rat}\ v \\
&\textbf{data } \mathit{RPred}\ v = \mathit{Equal}\ \ (\mathit{Term}\ v)\ (\mathit{Term}\ v) \\
&\qquad\qquad\ \ |\ \mathit{LessThan}\ (\mathit{Term}\ v)\ (\mathit{Term}\ v) \\
&\qquad\qquad\ \ |\ \mathit{Positive}\ \ (\mathit{Term}\ v) \\
&\qquad\qquad\ \ |\ \mathit{AND}\ (\mathit{RPred}\ v)\ (\mathit{RPred}\ v) \\
&\qquad\qquad\ \ |\ \mathit{NOT}\ (\mathit{RPred}\ v) \\
&\quad \textbf{deriving } (\mathit{Eq}, \mathit{Show})
\end{aligned}
$$

Note that the first three constructors, *Eq*, *LessThan*, and *Positive*, describe predicates or relations between terms (which can contain term variables) while the two last constructors, *AND* and *NOT*, just combine such relations together. (Terminology: I often mix the words "predicate" and "relation".)

We have already defined the evaluator for the *Term v* type but we need to add a corresponding "evaluator" (called *check*) for the *RPred v* type. Given values for all term variables the predicate checker should just determine if the predicate is true or false.

$$
\begin{aligned}
&\mathit{checkRP} :: (\mathit{Eq}\ v, \mathit{Show}\ v) \Rightarrow \mathit{Env}\ v\ \mathit{RatSem} \to \mathit{RPred}\ v \to \mathit{Bool} \\
&\mathit{checkRP}\ \mathit{env}\ (\mathit{Equal}\ \ t_1\ t_2) = \mathit{eqSem}\ \ (\mathit{evalRat2}\ \mathit{env}\ t_1)\ (\mathit{evalRat2}\ \mathit{env}\ t_2) \\
&\mathit{checkRP}\ \mathit{env}\ (\mathit{LessThan}\ t_1\ t_2) = \mathit{lessThanSem}\ (\mathit{evalRat2}\ \mathit{env}\ t_1)\ (\mathit{evalRat2}\ \mathit{env}\ t_2) \\
&\mathit{checkRP}\ \mathit{env}\ (\mathit{Positive}\ \ t_1) = \mathit{positiveSem}\ \ (\mathit{evalRat2}\ \mathit{env}\ t_1) \\
&\mathit{checkRP}\ \mathit{env}\ (\mathit{AND}\ p\ q) = (\mathit{checkRP}\ \mathit{env}\ p) \wedge (\mathit{checkRP}\ \mathit{env}\ q) \\
&\mathit{checkRP}\ \mathit{env}\ (\mathit{NOT}\ p) = \neg\ (\mathit{checkRP}\ \mathit{env}\ p)
\end{aligned}
$$

Given this recursive definition of *checkRP*, the semantic functions *eqSem*, *lessThanSem*, and *positiveSem* can be defined by just working with the rational number representation:

$$
\begin{aligned}
&\mathit{eqSem} :: \mathit{RatSem} \to \mathit{RatSem} \to \mathit{Bool} \\
&\mathit{lessThanSem} :: \mathit{RatSem} \to \mathit{RatSem} \to \mathit{Bool} \\
&\mathit{positiveSem} :: \mathit{RatSem} \to \mathit{Bool} \\
&\mathit{eqSem} = \mathit{error}\ \texttt{"TODO"} \\
&\mathit{lessThanSem} = \mathit{error}\ \texttt{"TODO"} \\
&\mathit{positiveSem} = \mathit{error}\ \texttt{"TODO"}
\end{aligned}
$$

2.15 More general code for first order languages

This subsection contains some extra material which is not a required part of the course.

It is possible to make one generic implementation of *FOL* which can be specialised to any first order language.

- *Term* = Syntactic terms
- n = names (of atomic terms)
- f = function names
- v = variable names
- *WFF* = Well Formed Formulas
- p = predicate names

```
data Term n f v = N n | F f [Term n f v] | V v
  deriving Show
data WFF n f v p =
    P p    [Term n f v]
  | Equal (Term n f v)    (Term n f v)
  | And   (WFF n f v p) (WFF n f v p)
  | Or    (WFF n f v p) (WFF n f v p)
  | Equiv (WFF n f v p) (WFF n f v p)
  | Impl  (WFF n f v p) (WFF n f v p)
  | Not   (WFF n f v p)
  | FORALL v (WFF n f v p)
  | EXISTS  v (WFF n f v p)
  deriving Show
```

2.16 Exercises: abstract FOL

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

2.16.1 Exercises

Short technical note For these first exercises on the propositional fragment of FOL (introduced in section 2.8), 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 2.1. Prove these three theorems (for arbitrary p and q):

```
Impl (And p q) q
Or p q → Or q p
Or p (Not p)
```

Exercise 2.2. Translate to Haskell and prove the De Morgan laws:

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

(translate equivalence to conjunction of two implications).

Exercise 2.3. So far, the implementation of the datatypes has played no role. To make this clearer: define the types for connectives in *AbstractFol* in any way you wish, e.g.:

$$\begin{aligned} And\ p\ q &= A\ () \\ Not\ p\ \ &= B\ p \end{aligned}$$

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 2.4. The introduction and elimination rules suggest that some implementations of the datatypes for connectives might be more reasonable than others. We have seen that the type of evidence for $p \to q$ is very similar to the type of functions $p \to q$, so it would make sense to define

type $Impl\ p\ q = (p \to q)$

Similarly, $\wedge{-}ElimL$ and $\wedge{-}ElimR$ behave like the functions *fst* and *snd* on pairs, so we can take

type $And\ p\ q = (p, q)$

while the notion of proof by cases is very similar to that of writing functions by pattern-matching on the various clauses, making $p \vee q$ similar to *Either*:

type $Or\ p\ q = Either\ p\ q$

1. Define and implement the corresponding introduction and implementation rules as functions.
2. Compare proving the distributivity laws

$$\begin{aligned} &(p \wedge q) \vee r \longleftrightarrow (p \vee r) \wedge (q \vee r) \\ &(p \vee q) \wedge r \longleftrightarrow (p \wedge r) \vee (q \wedge r) \end{aligned}$$

using the "undefined" introduction and elimination rules, with writing the corresponding functions with the given implementations of the datatypes. The first law, for example, requires a pair of functions:

$$\begin{aligned} &(Either\ (p, q)\ r \to (Either\ p\ r, Either\ q\ r) \\ &, (Either\ p\ r, Either\ q\ r) \to Either\ (p, q)\ r \\ &) \end{aligned}$$

Moral: The natural question is: is it true that every time we find an implementation using the "pairs, $\to$, *Either*" translation of sentences, we can also find one using the "undefined" introduction and elimination rules? The answer, perhaps surprisingly, is *yes*, as long as the functions we write are total. This result is known as *the Curry-Howard isomorphism.*

Exercise 2.5. Can we extend the Curry-Howard isomorphism to formulas with $\neg$? In other words, is there a type that we could use to define $Not\ p$, which would work together with pairs, $\to$, and *Either* to give a full translation of sentential logic?

Unfortunately, we cannot. The best that can be done is to define an empty type

data *Empty*

and define *Not* as

type $Not\ p = p \to Empty$

The reason for this definition is: when p is *Empty*, the type $Not\ p$ is not empty: it contains the identity

$idEmpty :: Empty \to Empty$
$isEmpty\ evE = evE$

When p is not *Empty* (and therefore is true), there is no (total, defined) function of type $p \to Empty$, and therefore $Not\ p$ is false.

Moreover, mathematically, an empty set acts as a contradiction: there is exactly one function from the empty set to any other set, namely the empty function. Thus, if we had an element of the empty set, we could obtain an element of any other set.

Now to the exercise:

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

$notIntro :: (p \to (q, q \to Empty)) \to (p \to Empty)$

Using

$contraHey :: Empty \to p$
$contraHey\ evE =$ **case** evE **of** $\{\}$

prove

$q \wedge \neg q \to p$

You will, however, not be able to prove $p \vee \neg p$ (try it!).

Prove

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

but you will not be able to prove the converse.

Exercise 2.6. The implementation $Not\ p = p \to Empty$ is not adequate for representing all the closed formulas in FOL, but it is adequate for *constructive logic* (also known as *intuitionistic*). In constructive logic, the $\neg p$ is *defined* as $p \to \bot$, and the following elimination rule is given for $\bot$: $\bot{-}Elim : \bot \to a$, corresponding to the principle that everything follows from a contradiction ("if you believe $\bot$, you believe everything").

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:

$noContra :: (Either\ p\ (p \to Empty) \to Empty) \to Empty$

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

Exercise 2.7. *From exam 2016-08-23*

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 2.8. *From exam 2017-08-22*

Adequate notation for mathematical concepts and proofs (or "50 shades of continuity").

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):

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

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 $new\delta$.

2.16.2 More exercises

Preliminary remarks

- 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 2.12) by

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

Exercise 2.9. Consider the statement:

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

a. Define the sequence $\{\, a_n \,\}$ as a function $a : \mathbb{N} \to \mathbb{R}$.

b. 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

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

The resulting formula should have no $\neg$ in it (that's possible because the negation of $<$ is $\geqslant$).

c. Give a functional interpretation of the resulting formula.

d. Sketch an implementation of the function, considering two cases: $L \neq 0$ and $L = 0$.

Exercise 2.10. Consider the statement:

The limit of a convergent sequence is unique.

a. 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

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

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.

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

3 Types in Mathematics

```
{-# LANGUAGE FlexibleInstances #-}
module DSLsofMath.W03 where
import Data.Char (toUpper)
```

3.1 Types in mathematics

Types are sometimes mentioned explicitly in mathematical texts:

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

The types of "higher-order" operators are usually not given explicitly:

- $lim : (\mathbb{N} \to \mathbb{R}) \to \mathbb{R}$ 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 $D\,f$
- $\partial f/\partial x_i : (\mathbb{R}^n \to \mathbb{R}) \to (\mathbb{R}^n \to \mathbb{R})$
- 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 $f(x, y, z) = \ldots$
- a better notation (by Landau) which doesn't rely on the names given to the arguments was popularised in Landau [1934] (English edition Landau [2001]): D_1 for the partial derivative with respect to x_1, etc.
- Exercise: for $f : \mathbb{R}^2 \to \mathbb{R}$ define D_1 and D_2 using only D.

3.2 Typing Mathematics: derivative of a function

Let's start simple with the classical definition of the derivative from 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 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 $X = Y$ 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 it is useful to briefly look back to the definition of "limit of a function" in Section 2.13.) 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, $D\ f\ x = lim\ (\varphi\ x)\ 0$. [3]

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.

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

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 $f' : \mathbb{R} \to \mathbb{R}$, only numerical approximations. But 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 3.7.

3.3 Typing Mathematics: partial derivative

Continuing on our quest of typing the elements of mathematical textbook definitions we now turn to a functions of more than one argument. Our example here will is from page 169 of Mac Lane [1986], 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

$$
\begin{aligned}
&U \subseteq \mathbb{R} \times \mathbb{R} \quad \text{-- cartesian plane}\\
&f \ : U \to \mathbb{R}\\
&z \ : U \to \mathbb{R} \quad \text{-- but see below}
\end{aligned}
$$

[3] 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: $D\ f = limAt\ 0 \circ \psi\ f$.

$$f'_x \ : \ U \to \mathbb{R}$$

The x in the subscript of f' is *not* a real number, but a symbol (a *Char*).

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 appears to be a non-zero real number, 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 $lim\ (\varphi\ (x, y))\ 0$. 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 $z \in \mathbb{R}$, although the phrase "the quantity z" (on line 2) suggests this. A possible interpretation is that z is used to abbreviate the expression $f\ (x, y)$; thus, everywhere we can replace z with $f\ (x, y)$. In particular, $\partial z\ /\ \partial x$ becomes $\partial f\ (x, y)\ /\ \partial x$, which we can interpret as $\partial f\ /\ \partial x$ applied to (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.

3.4 Type inference and understanding: Lagrangian case study

From (Sussman and Wisdom 2013):

> 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, we start typing the elements involved:

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

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

   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, 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 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 $\mathbb{R}$.)

b. 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

$$\begin{array}{l} L : (T, Q, V) \to \mathbb{R} \\ L\ \ (t, q, v) \quad = \ \ldots \end{array}$$

c. 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 is *not* the real number 0, but rather the constant function *const* 0:

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

d. 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!

e. 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.

f. 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 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. We will use $n = 1$ for the rest of this example.

g. Now that we have a path, the coordinates at any time are given by the path. And as 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.

$$\begin{array}{ll} q : T \to Q & \\ q\ t = w\ t & \text{-- or, equivalently, } q = w \end{array}$$

$\dot{q} : T \to V$
$\dot{q}\ t = dw\ /\ dt$ -- or, equivalently, $\dot{q} = D\ w$

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

$expand : (T \to Q) \to (T \to (T, Q, V))$
$expand\ w\ t = (t, w\ t, D\ w\ t)$

h. 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

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

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

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

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

i. 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:

$D\ ((D_2\ L) \circ (expand\ w)) : T \to \mathbb{R}$
$(D_3\ L)\ \circ (expand\ w)\ : T \to \mathbb{R}$

The equation becomes

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

or, after simplification:

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

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

j. "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):

$Lagrange\ (L, w) = D\ (D_3\ L \circ expand\ w) == D_2\ L \circ expand\ w$

where we use (==) to avoid confusion with the equlity sign (=) used for the definition of the predicate.

So, we have figured out what the equation "means", in terms of operators 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 "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.

3.5 Type in Mathematics (Part II)

3.5.1 Type classes

The kind of type inference we presented in the last lecture becomes automatic with experience in a domain, but is very useful in the beginning.

The "trick" of looking for an appropriate combinator with which to pre- or post-compose a function in order to makes types match is often useful. It is similar to the casts one does automatically in expressions such as $4 + 2.5$.

One way to understand such casts from the point of view of functional programming is via *type classes*. As a reminder, the reason $4 + 2.5$ works is because floating point values are members of the class *Num*, which includes the member function

$$\mathit{fromInteger} :: \mathit{Integer} \to a$$

which converts integers to the actual type a.

Type classes are related to mathematical structures which, in turn, are related to DSLs. The structuralist point of view in mathematics is that each mathematical domain has its own fundamental structures. Once these have been identified, one tries to push their study as far as possible *on their own terms*, i.e., without introducing other structures. For example, in group theory, one starts by exploring the consequences of just the group structure, before one introduces, say, an order structure and monotonicity.

The type classes of Haskell seem to have been introduced without relation to their mathematical counterparts, perhaps because of pragmatic considerations. For now, we examine the numerical type classes *Num*, *Fractional*, and *Floating*.

$$\begin{array}{l}\textbf{class}\ (\mathit{Eq}\ a, \mathit{Show}\ a) \Rightarrow \mathit{Num}\ a\ \textbf{where}\\ \quad (+), (-), (*) :: a \to a \to a\\ \quad \mathit{negate} \quad :: a \to a\\ \quad \mathit{abs}, \mathit{signum} :: a \to a\\ \quad \mathit{fromInteger} :: \mathit{Integer} \to a\end{array}$$

This is taken from the Haskell documentation[4] but it appears that *Eq* and *Show* are not necessary, because there are meaningful instances of *Num* which don't support them:

$$\begin{array}{l}\textbf{instance}\ \mathit{Num}\ a \Rightarrow \mathit{Num}\ (x \to a)\ \textbf{where}\\ \quad f + g \quad = \lambda x \to f\ x + g\ x\\ \quad f - g \quad = \lambda x \to f\ x - g\ x\\ \quad f * g \quad = \lambda x \to f\ x * g\ x\\ \quad \mathit{negate}\ f \quad = \mathit{negate} \circ f\\ \quad \mathit{abs}\ f \quad = \mathit{abs} \circ f\\ \quad \mathit{signum}\ f \quad = \mathit{signum} \circ f\\ \quad \mathit{fromInteger} = \mathit{const} \circ \mathit{fromInteger}\end{array}$$

This instance for functions allows us to write expressions like $\mathit{sin} + \mathit{cos} :: \mathit{Double} \to \mathit{Double}$ or $\mathit{sq} * \mathit{double} :: \mathit{Integer} \to \mathit{Integer}$. As another example:

$$\mathit{sin\hat{}2} = \lambda x \to (\mathit{sin}\ x)\hat{}(\mathit{const}\ 2\ x) = \lambda x \to (\mathit{sin}\ x)\hat{}2$$

thus the typical math notation $\sin^2$ works fine in Haskell. (Note that there is a clash with another use of superscript for functions: sometimes $f\hat{}n$ means *composition* of f with itself n times. With that reading sin^2 would mean $\lambda x \to \mathit{sin}\ (\mathit{sin}\ x)$.)

Exercise: play around with this a bit in ghci.

[4] Fig. 6.2 in section 6.4 of the Haskell 2010 report: [Marlow , ed., Sect. 6.4].

3.5.2 Overloaded integers literals

As an aside, we will spend some time explaining a convenient 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. This means that the same program text can have different meaning depending on the type of the context. The literal $three = 3$, for example, can be used as an integer, a real number, a complex number, or even as a (constant) function (by the instance $Num\ (x \to a)$).

The instance declaration of the method *fromInteger* above looks recursive, but is not. The same pattern appeared already in section 1.6, which near the end included roughly the following lines:

instance $Num\ r \Rightarrow Num\ (ComplexSyn\ r)$ **where**
  -- ... several other methods and then
  $fromInteger = toComplexSyn \circ fromInteger$

To see why this is not a recursive definition we need to expand the type and to do this I will introduce a name for the right hand side (RHS): *fromIntC*.

```
--          ComplexSyn r <---------- r <---------- Integer
fromIntC =              toComplexSyn . fromInteger
```

I have placed the types in the comment, with "backwards-pointing" arrows indicating that *fromInteger*:: $Integer \to r$ and $toComplexSyn :: r \to ComplexSyn\ r$ while the resulting function is $fromIntC :: Integer \to ComplexSyn\ r$. The use of *fromInteger* at type r means that the full type of *fromIntC* must refer to the *Num* class. Thus we arrive at the full type:

$fromIntC :: Num\ r \Rightarrow Integer \to ComplexSyn\ r$

As an example we have that

$3 :: ComplexSyn\ Double$	== {- *Integer* literals have an implicit *fromInteger* -}
$(fromInteger\ 3) :: ComplexSyn\ Double$	== {- *Num* instance for *ComplexSyn* -}
$toComplexSyn\ \ (fromInteger\ 3)$	== {- *Num* instance for *Double* -}
$toComplexSyn\ \ 3.0$	== {- Def. of *toComplexSyn* from Section 1.6 -}
$FromCartesian\ 3.0\ 0$	== {- *Integer* literals have an implicit *fromInteger* -}
$FromCartesian\ 3.0\ (fromInteger\ 0)$	== {- *Num* instance for *Double*, again -}
$FromCartesian\ 3.0\ 0.0$	

3.5.3 Back to the numeric hierarchy instances for functions

Back to the main track: defining numeric operations on functions. We have already defined the operations of the *Num* class, but we can move on to the neighbouring classes *Fractional* and *Floating*.

The class *Fractional* is for types which in addition to the *Num* operations also supports division:

class $Num\ a \Rightarrow Fractional\ a$ **where**
  $(/)\ :: a \to a \to a$
  $recip\ :: a \to a$ -- $\lambda x \to 1\ /\ x$
  $fromRational :: Rational \to a$ -- similar to *fromInteger*

and the *Floating* class collects the "standard" functions from calculus:

```
class Fractional a ⇒ Floating a where
  π                    :: a
  exp, log, √·         :: a → a
  (**), logBase        :: a → a → a
  sin, cos, tan        :: a → a
  asin, acos, atan     :: a → a
  sinh, cosh, tanh     :: a → a
  asinh, acosh, atanh :: a → a
```

We can instantiate these type classes for functions in the same way we did for *Num*:

```
instance Fractional a ⇒ Fractional (x → a) where
  recip f        = recip ∘ f
  fromRational = const ∘ fromRational


instance Floating a ⇒ Floating (x → a) where
  π      = const π
  exp f  = exp ∘ f
  f ** g = λx → (f x) ** (g x)
    -- and so on
```

Exercise: complete the instance declarations.

These type classes represent an abstract language of algebraic 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.

3.6 Type classes in Haskell

We now abstract from *Num* and look at what a type class is and how it is used. One view of a type class is as a set of types. For *Num* that is the set of "numeric types", for *Eq* the set of "types with computable equality", etc. The types in this set are called instances and are declared by **instance** declarations. When a class C is defined, there are no types in this set (no instances). In each Haskell module where C is in scope there is a certain collection of instance declarations. Here is an example of a class with just two instances:

```
class C a where
  foo :: a → a
instance C Integer where
  foo = (1+)
instance C Char where
  foo = toUpper
```

Here we see the second view of a type class: as a collection of overloaded methods (here just *foo*). Overloaded here means that the same symbol can be used with different meaning at different types. If we use *foo* with an integer it will add one, but if we use it with a character it will convert it to upper case. The full type of *foo* is $C\ a \Rightarrow a \to a$ and this means that it can be used at any type a for which there is an instance of C in scope.

Instance declarations can also be parameterised:

```
instance C a ⇒ C [a] where
  foo xs = map foo xs
```

This means that for any type a which is already an instance of C we also make the type $[a]$ an instance (recursively). Thus, we now have an infinite collection of instances of C: *Char*, $[Char]$, $[[Char]]$, etc. Similarly, with the function instance for *Num* above, we immediately make the types $x \to Double$, $x \to (y \to Double)$, etc. into instances (for all x, y, ...).

3.7 Computing derivatives

An important part of calculus is the collection of laws, or rules, for computing derivatives. Using the notation $D\ f$ for the derivative of f and lifting the numeric operations to functions we can fill in a nice table of examples which can be followed to compute derivatives of many functions:

$$
\begin{array}{ll}
D\ (f + g) & = D\ f + D\ g \\
D\ (f * g) & = D\ f * g + f * D\ g \\
D\ (f \circ g)\ x & = D\ f\ (g\ x) * D\ g\ x \quad \text{-- the chain rule} \\
D\ (const\ a) & = const\ 0 \\
D\ id & = const\ 1 \\
D\ (\hat{}n)\ x & = (n-1) * (x\hat{}(n-1)) \\
D\ sin\ x & = cos\ x \\
D\ cos\ x & = -(sin\ x) \\
D\ exp\ x & = exp\ x
\end{array}
$$

and so on.

If we want to get a bit closer to actually implementing D we quickly notice a problem: if D has type $(\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$ we have no way of telling which of these rules we should apply. Given a real (semantic) function f as an argument, D cannot know if this function was written using a $+$, or *sin* or *exp* as outermost operation. The only thing D 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. All the rules in the table seem to work on *syntactic* functions: abstract syntax trees *representing* the real (semantic) functions.

We observe that we can compute derivatives for any expressions made out of arithmetical functions, standard functions, and their compositions. In other words, the computation of derivatives is based on a domain specific langauge (a DSL) of expressions (representing functions in one variable). Here is the start of a grammar for this little language:

$$
\begin{array}{rl}
expression ::= & const\ \mathbb{R} \\
| & id \\
| & expression + expression \\
| & expression * expression \\
| & exp\ expression \\
| & \ldots
\end{array}
$$

We can implement this in a datatype:

```
data FunExp = Const Double
            | Id
            | FunExp :+: FunExp
            | FunExp :*: FunExp
            | Exp FunExp
              -- and so on
            deriving Show
```

The intended meaning of elements of the *FunExp* type is functions:

$$
\begin{array}{lll}
eval :: FunExp & \to (Double \to Double) & \\
eval\ \ (Const\ \alpha) & = const\ \alpha & \\
eval\ \ Id & = id & \\
eval\ \ (e_1 :+: e_2) & = eval\ e_1 + eval\ e_2 & \text{-- note the use of ``lifted +'',} \\
eval\ \ (e_1 :*: e_2) & = eval\ e_1 * eval\ e_2 & \text{-- ``lifted *'',} \\
eval\ \ (Exp\ e_1) & = exp\ (eval\ e_1) & \text{-- and ``lifted } exp\text{''.} \\
\text{-- and so on} & &
\end{array}
$$

An example:

$$
\begin{array}{l}
f_1 :: Double \to Double \\
f_1\ x = exp\ (x\hat{}2) \\
e_1 :: FunExp \\
e_1 = Exp\ (Id :*: Id)
\end{array}
$$

We can implement the derivative of *FunExp* expressions using the rules of derivatives. We want to implement a function $derive :: FunExp \to FunExp$ which makes the following diagram commute:

$$
\begin{array}{ccc}
FunExp & \xrightarrow{eval} & Func \\
\Big\downarrow{\scriptstyle derive} & & \Big\downarrow{\scriptstyle D} \\
FunExp & \xrightarrow{eval} & Func
\end{array}
$$

In other words we want

$$eval \circ derive\ e = D \circ eval$$

or, in other words, for any expression e, we want

$$eval\ (derive\ e) = D\ (eval\ e)$$

For example, let us derive the *derive* function for *Exp e*:

$$
\begin{array}{ll}
eval\ (derive\ (Exp\ e)) & = \{\text{- specification of } derive \text{ above -}\} \\
D\ (eval\ (Exp\ e)) & = \{\text{- def. } eval \text{ -}\} \\
D\ (exp\ (eval\ e)) & = \{\text{- def. } exp \text{ for functions -}\} \\
D\ (exp \circ eval\ e) & = \{\text{- chain rule -}\} \\
(D\ exp \circ eval\ e) * D\ (eval\ e) & = \{\text{- } D \text{ rule for } exp \text{ -}\} \\
(exp \circ eval\ e) * D\ (eval\ e) & = \{\text{- specification of } derive \text{ -}\} \\
(exp \circ eval\ e) * (eval\ (derive\ e)) & = \{\text{- def. of } eval \text{ for } Exp \text{ -}\} \\
(eval\ (Exp\ e)) * (eval\ (derive\ e)) & = \{\text{- def. of } eval \text{ for } :*: \text{ -}\} \\
eval\ (Exp\ e :*: derive\ e) &
\end{array}
$$

Therefore, the specification is fulfilled by taking

$$derive\ (Exp\ e) = Exp\ e :*: derive\ e$$

Similarly, we obtain

$$
\begin{array}{ll}
derive\ (Const\ \alpha) & = Const\ 0 \\
derive\ Id & = Const\ 1 \\
derive\ (e_1 :+: e_2) & = derive\ e_1 :+: derive\ e_2 \\
derive\ (e_1 :*: e_2) & = (derive\ e_1 :*: e_2) :+: (e_1 :*: derive\ e_2) \\
derive\ (Exp\ e) & = Exp\ e :*: derive\ e
\end{array}
$$

Exercise: complete the *FunExp* type and the *eval* and *derive* functions.

3.8 Shallow embeddings

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 the first part of this lecture. 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.

The second approach is called "shallow embedding" or "almost abstract syntax". It can be more economical, since it needs no *eval*. The question is: can we implement *derive* in the shallow embedding?

Note that the reason the shallow embedding is possible 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. This is sometimes referred to as "compositionality".

We check whether the semantics of derivatives is compositional. The evaluation function for derivatives is

$$
\begin{aligned}
&eval' :: FunExp \to Double \to Double \\
&eval' = eval \circ derive
\end{aligned}
$$

For example:

$$
\begin{aligned}
&eval'\ (Exp\ e) && = \{\text{- def. } eval', \text{ function composition -}\} \\
&eval\ (derive\ (Exp\ e)) && = \{\text{- def. } derive \text{ for } Exp \text{ -}\} \\
&eval\ (Exp\ e :*: derive\ e) && = \{\text{- def. } eval \text{ for } :*: \text{ -}\} \\
&eval\ (Exp\ e) * eval\ (derive\ e) && = \{\text{- def. } eval \text{ for } Exp \text{ -}\} \\
&exp\ (eval\ e) * eval\ (derive\ e) && = \{\text{- def. } eval' \text{ -}\} \\
&exp\ (eval\ e) * eval'\ e && = \{\text{- let } f = eval\ e,\ f' = eval'\ e \text{ -}\} \\
&exp\ f * f'
\end{aligned}
$$

Thus, given only the derivative $f' = eval'\ e$, it is impossible to compute $eval'\ (Exp\ e)$. (There is no way to implement $eval' Exp :: (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$.) Thus, it is not possible to directly implement *derive* using shallow embedding; the semantics of derivatives is not compositional. Or rather, *this* semantics is not compositional. It is quite clear that the derivatives cannot be evaluated without, at the same time, being able to evaluate the functions. So we can try to do both evaluations simultaneously:

$$
\begin{aligned}
&\textbf{type } FD\ a = (a \to a, a \to a) \\
&evalD :: FunExp \to FD\ Double \\
&evalD\quad e \qquad = (eval\ e, eval'\ e)
\end{aligned}
$$

(Note: At this point, you are adviced to look up and solve exercise 1.9 on the "tupling transform" in case you have not done so already.)

Is *evalD* compositional?

We compute, for example:

$$
\begin{aligned}
&evalD\ (Exp\ e) && = \{\text{- specification of } evalD \text{ -}\} \\
&(eval\ (Exp\ e), eval'\ (Exp\ e)) && = \{\text{- def. } eval \text{ for } Exp \text{ and reusing the computation above -}\} \\
&(exp\ (eval\ e), exp\ (eval\ e) * eval'\ e) && = \{\text{- introduce names for subexpressions -}\} \\
&\textbf{let } f\ = eval\ e
\end{aligned}
$$

$$
\begin{array}{ll}
\quad f' = eval'\ e & \\
\textbf{in}\ (exp\ f, exp\ f * f') & = \{\text{- def. } evalD \text{ -}\} \\
\textbf{let}\ (f, f') = evalD\ e & \\
\textbf{in}\ (exp\ f, exp\ f * f') &
\end{array}
$$

This semantics *is* compositional and the *Exp* case is:

$$
\begin{array}{l}
evalDExp :: FD\ Double \to FD\ Double \\
evalDExp \quad (f, f') = (exp\ f, exp\ f * f')
\end{array}
$$

We can now define a shallow embedding for the computation of derivatives, using the numerical type classes.

$$
\begin{array}{l}
\textbf{instance}\ Num\ a \Rightarrow Num\ (a \to a, a \to a)\ \textbf{where} \\
\quad (f, f') + (g, g') = (f + g, f' \quad\ + g' \quad) \\
\quad (f, f') * (g, g') = (f\ *\ g, f' * g + f * g') \\
\quad fromInteger\ n\ \ = (fromInteger\ n, const\ 0)
\end{array}
$$

Exercise: implement the rest

3.9 Exercises

Exercise 3.1. 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

- $w1 = id$ or
- $w2 = sin$ or
- $w3 = (q0-) \circ (g*) \circ (/2) \circ (\hat{}2)$

3.10 Exercises from old exams

Exercise 3.2. *From exam 2016-Practice*

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 3.3. *From exam 2016-03-16*

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 3.4. *From exam 2016-08-23*

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

- $P\ \emptyset = 0$, $P\ \Omega = 1$
- A and B are disjoint (i.e., $A \cap B = \emptyset$), then: $P\ A + P\ B = P\ (A \cup B)$.

Conditional probabilities are defined as follows (*Elementary Probability 2nd Edition*, 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
> $P\ (A \mid B) = P\ (A \cap B)\ /\ P\ B$

a. What are the types of the elements involved in the definition of conditional probability? (P, $\cap$, $/$, $|$)

b. In the 1933 monograph that set the foundations of contemporary probability theory, Kolmogorov used, instead of $P\ (A \mid B)$, the expression $P_A\ B$. Type this expression. Which notation do you prefer (provide a *brief* explanation).

Exercise 3.5. *From exam 2017-03*

(Note that this exam question was later included as one of the examples in this chapter, see 3.3. 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 3.6. *From exam 2017-08-22*

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.

a. 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 $\{0, 1, ..., n-1\}$.

b. 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 contraints (if any) are needed on the type of the matrix entries in the two cases?

c. 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 3.7. (Extra material outside the course.) In the same direction as the Lagrangian case study in section 3.4 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.

4 Compositional Semantics and Algebraic Structures

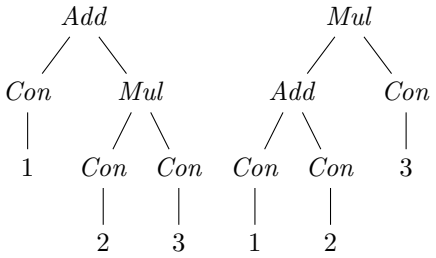
```
{-# LANGUAGE FlexibleInstances, GeneralizedNewtypeDeriving #-}
module DSLsofMath.W04 where
import Prelude hiding (Monoid)
```

4.1 Compositional semantics

4.1.1 A simpler example of a non-compositional function

Consider a very simple datatype of 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
```



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 → 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:

```
pretty (Add x y) = prettyAdd (pretty x) (pretty y)
pretty (Mul x y) = prettyMul (pretty x) (pretty y)
pretty (Con c)   = prettyCon c

prettyAdd :: String → String → String
prettyMul :: String → String → String
prettyCon :: Integer → String
```

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 both e_1 and e_2 are different but they pretty-print to the same string. There are many ways to fix this, some more "pretty" than others, but the main problem is that some information is lost in the translation.

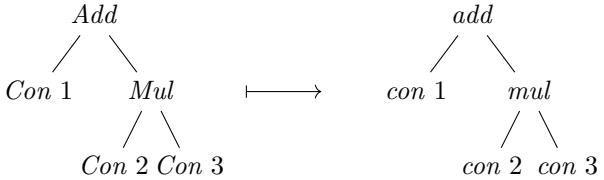
TODO(perhaps): Explain using three pretty printers for the three "contexts": at top level, inside *Add*, inside *Mul*, ... then combine them with the tupling transform just as with *evalD*. The result is the following:

```
prTop :: E → String
prTop e = let (pTop, _, _) = prVersions e
          in pTop
prVersions = foldE prVerAdd prVerMul prVerCon
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 (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 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.

4.1.2 Compositional semantics in general

In general, for a syntax *Syn*, and a possible semantics (a type *Sem* and an *eval* function of type *Syn* → *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 a "semantic constructor" *ci*. Thus, in our datatype E, a compositional semantics means that *Add* maps to *add*, *Mul* ↦ *mul*, and *Con* ↦ *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:

```
evalE1 :: E → Integer
evalE1 = foldE (+) (*) id
```

and with a minimal modification we can also make it work for other numeric types:

```
evalE2 :: Num a ⇒ E → a
evalE2 = foldE (+) (*) fromInteger
```

Another thing worth noting is that if we replace each abstract syntax constructor with itself we get the identity function (a "deep copy"):

```
idE :: E → E
idE = foldE Add Mul Con
```

Finally, it is often 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 make "hide" the arguments to the fold:

```
foldIE :: IntExp t ⇒ E → t
foldIE = foldE add mul con
instance IntExp E where
  add = Add
  mul = Mul
  con = Con
instance IntExp Integer where
  add = (+)
  mul = (*)
  con = id
idE' :: E → E
idE' = foldIE
evalE' :: E → Integer
evalE' = foldIE
```

And we can also see *pretty* as instance:

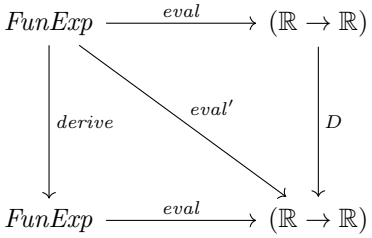
```
instance IntExp String where
  add = prettyAdd
  mul = prettyMul
  con = prettyCon
pretty' :: E → String
pretty' = foldIE
```

4.1.3 Back to derivatives and evaluation

TODO: perhaps not include this here. The background is that this material did not quite fit in the previous lecture. Also some repition was needed.

Review section 3.8 again with the definition of *eval′* being non-compositional (just like *pretty*) and *evalD* a more complex, but compositional, semantics.

We want to implement $eval' = eval \circ derive$ in the following diagram:



As we saw in section 3.8 this does not work. Instead we need to use the "tupling transform" to compute a pair of f and $D\,f$ at once.

4.2 Algebraic Structures and DSLs

In this lecture, we continue exploring the relationship between type classes, mathematical structures, and DSLs.

4.2.1 Algebras, homomorphisms

TODO: rewrite to make it smoother

From Wikipedia:

> In universal algebra, an algebra (or algebraic structure) is a set A together with a collection of operations on A.
>
> Example:
>
> > **class** *Monoid a* **where**
> > *unit* :: *a*
> > *op* :: *a* → *a* → *a*
>
> After the operations have been specified, the nature of the algebra can be further limited by axioms, which in universal algebra often take the form of identities, or *equational laws*.
>
> Example: Monoid equations
>
> A monoid is an algebra which has an associative operation 'op' and a unit. The laws can be formulated as the following equations:
>
> $$\forall\, x : a.\ (unit\ `op`\ x == x \wedge x\ `op`\ unit == x)$$
> $$\forall\, x, y, z : a.\ (x\ `op`\ (y\ `op`\ z) == (x\ `op`\ y)\ `op`\ z)$$

Examples of monoids include numbers with additions, $(\mathbb{R}, 0, (+))$, numbers with multiplication $(\mathbb{R}_{>0}, 1, (*))$, and even endofunctions with composition $(a \to a, id, (\circ))$. It is a good exercise to check that the laws are satisfied.

In mathematics, as soon as there are several examples of a structure, the question of what "translation" between them comes up. An important class of such "translations" are "structure preserving maps" called *homomorphisms*. As two examples, we have the homomorphisms *log* and *exp*:

$$
\begin{array}{ll}
exp & : \mathbb{R} \to \mathbb{R}_{>0} \\
exp\ 0 & = 1 \\
exp\ (a+b) & = exp\ a * exp\ b \\
log & : \mathbb{R}_{>0} \to \mathbb{R} \\
log\ 1 & = 0 \\
log\ (a*b) & = log\ a + log\ b
\end{array}
$$

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 fA of A and corresponding fB of B (of arity, say, n), $h\ (fA\ (x_1, ..., x_n)) = fB\ (h\ (x_1), ..., h\ (x_n))$.

Example: Monoid homomorphism

```
h unit       = unit           -- h takes units to units
h (x `op` y) = h x `op` h y   -- and distributes over op
```

```
newtype ANat     = A Int  deriving (Show, Num, Eq)
instance Monoid ANat where
  unit           = A 0
  op (A m) (A n) = A (m + n)
newtype MNat     = M Int deriving (Show, Num, Eq)
instance Monoid MNat where
  unit           = M 1
  op (M m) (M n) = M (m * n)
```

Exercise: characterise the homomorphisms from *ANat* to *MNat*.

Solution:

Let $h : ANat \to MNat$ be a homomorphism. Then

$$
\begin{array}{ll}
h\ 0 & = 1 \\
h\ (x+y) & = h\ x * h\ y
\end{array}
$$

For example $h\ (x+x) = h\ x * h\ x = (h\ x)\hat{}2$ which for $x = 1$ means that $h\ 2 = (h\ 1)\hat{}2$.

More generally, every n in *ANat* is equal to the sum of n ones: $1 + 1 + ... + 1$. Therefore

$$h\ n = (h\ 1)\hat{}n$$

Every choice of $h\ 1$ "induces a homomorphism". This means that the value of the function h is fully determined by its value for 1.

Exercise: show that *const* is a homomorphism. The distribution law can be shown as follows:

```
h a + h b                          = {- h = const in this case -}
const a + const b                  = {- By def. of (+) on functions -}
(λx → const a x + const b x)       = {- By def. of const, twice -}
(λx → a + b)                       = {- By def. of const -}
const (a + b)                      = {- h = const -}
h (a + b)
```

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: Show that *apply c* is a homomorphism for all c, where $apply\ x\ f = f\ x$.

4.2.2 Homomorphism and compositional semantics

Last time, we saw that *eval* is compositional, while *eval′* is not. Another way of phrasing that is to say that *eval* is a homomorphism, while *eval′* is not. To see this, we need to make explicit the structure of *FunExp*:

```
instance Num FunExp where
  (+)         = (:+:)
  (*)         = (:*:)
  fromInteger = Const ∘ fromInteger
instance Fractional FunExp where
instance Floating FunExp where
  exp         = Exp
```

and so on.

Exercise: complete the type instances for *FunExp*.

For instance, we have

```
eval (e1 :*: e2) = eval e1 * eval e2
eval (Exp e)     = exp (eval e)
```

These properties do not hold for *eval′*, but do hold for *evalD*.

The numerical classes in Haskell do not fully do justice to the structure of expressions, for example, they do not contain an identity operation, which is needed to translate *Id*, nor an embedding of doubles, etc. If they did, then we could have evaluated expressions more abstractly:

```
eval :: GoodClass a ⇒ FunExp → a
```

where *GoodClass* gives exactly the structure we need for the translation.

Exercise: define *GoodClass* and instantiate *FunExp* and *Double* → *Double* as instances of it. Find another instance of *GoodClass*.

```
class GoodClass t where
  addF :: t → t → t
  mulF :: t → t → t
  expF :: t → t
    -- ... Exercise: continue to mimic the FunExp datatype as a class
newtype FD a = FD (a → a, a → a)
instance Num a ⇒ GoodClass (FD a) where
  addF = evalDApp
  mulF = evalDMul
  expF = evalDExp
    -- ... Exercise: fill in the rest
evalDApp = error "Exercise"
evalDMul = error "Exercise"
evalDExp = error "Exercise"
```

Therefore, we can always define a homomorphism from *FunExp* to *any* instance of *GoodClass*, in an essentially unique way. In the language of category theory, *FunExp* is an initial algebra.

Let us explore this in the simpler context of *Monoid*. The language of monoids is given by

$$\textbf{type } \mathit{Var} \quad = \mathit{String}$$
$$\textbf{data } \mathit{MExpr} = \mathit{Unit} \mid \mathit{Op}\ \mathit{MExpr}\ \mathit{MExpr} \mid \mathit{V}\ \mathit{Var}$$

Alternatively, we could have parametrised *MExpr* over the type of variables.

Just as in the case of FOL terms, we can evaluate an *MExpr* in a monoid instance if we are given a way of interpreting variables, also called an assignment:

$$\mathit{evalM} :: \mathit{Monoid}\ a \Rightarrow (\mathit{Var} \to a) \to (\mathit{MExpr} \to a)$$

Once given an $f :: \mathit{Var} \to a$, the homomorphism condition defines *evalM*:

$$\begin{aligned}
&\mathit{evalM}\ f\ \mathit{Unit} &&= \mathit{unit}\\
&\mathit{evalM}\ f\ (\mathit{Op}\ e_1\ e_2) &&= \mathit{op}\ (\mathit{evalM}\ f\ e_1)\ (\mathit{evalM}\ f\ e_2)\\
&\mathit{evalM}\ f\ (\mathit{V}\ x) &&= f\ x
\end{aligned}$$

(Observation: In *FunExp*, the role of variables was played by *Double*, and the role of the assignment by the identity.)

The following correspondence 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

The underlying theory of this table is a fascinating topic but mostly out of scope for the DSLsofMath course. See Category Theory and Functional Programming for a whole course around this (lecture notes are available on github).

4.2.3 Other homomorphisms

Last time, we defined a *Num* instance for functions with a *Num* 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:

$$\mathit{Num}\ a \Rightarrow x \to (x \to a) \to a$$

As suggested by the type, the homomorphism is just function application:

$$\begin{aligned}
&\mathit{apply} :: a \to (a \to b) \to b\\
&\mathit{apply}\ a = \lambda f \to f\ a
\end{aligned}$$

Indeed, writing $h = \mathit{apply}\ c$ for some fixed c, we have

$$\begin{aligned}
&h\ (f + g) = \{\text{- def. } \mathit{apply} \text{ -}\}\\
&(f + g)\ c = \{\text{- def. } + \text{ for functions -}\}\\
&f\ c + g\ c = \{\text{- def. } \mathit{apply} \text{ -}\}\\
&h\ f + h\ g
\end{aligned}$$

etc.

Can we do something similar for FD?

The elements of $FD\ a$ are pairs of functions, so we can take

$$\begin{array}{l} applyFD :: a \to FD\ a\ \to (a, a) \\ applyFD \quad c \quad\ (f, f')\ =\ (f\ c, f'\ c) \end{array}$$

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 (a, a). In fact, we can *compute* this structure from the homomorphism condition. For example:

$$\begin{array}{ll} h\ ((f, f') * (g, g')) & = \{\text{- def. } * \text{ for } FD\ a \text{ -}\} \\ h\ (f * g, f' * g + f * g') & = \{\text{- def. } h = applyFD\ c \text{ -}\} \\ ((f * g)\ c, (f' * g + f * g')\ c) & = \{\text{- def. } * \text{ and } + \text{ for functions -}\} \\ (f\ c * g\ c, f'\ c * g\ c + f\ c * g'\ c) & = \{\text{- homomorphism condition from step 1 -}\} \\ h\ (f, f') \circledast h\ (g, g') & = \{\text{- def. } h = applyFD\ c \text{ -}\} \\ (f\ c, f'\ c) \circledast (g\ c, g'\ c) & \end{array}$$

The identity will hold if we take

$$(x, x') \circledast (y, y') = (x * y, x' * y + x * y')$$

Exercise: complete the instance declarations for $(Double, Double)$.

Note: As 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 redefine FD to be

$$\textbf{type}\ FD\ a = (a, a)$$

Hint: Something very similar can be used for Assignment 2.

4.3 Summing up: definitions and representation

We defined a *Num* structure on pairs $(Double, Double)$ by requiring the operations to be compatible with the interpretation $(f\ a, f'\ a)$. For example

$$(x, x') \circledast (y, y') = (x * y, x' * y + x * y')$$

There is nothing in the "nature" of pairs of *Double* 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*:

$$(x, x') *. (y, y') = (x * y - x' * y', x * y' + x' * y)$$

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 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 above). 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 see this distinction again in section 5.1.

4.3.1 Some helper functions

```
instance Num E where    -- Some abuse of notation (no proper negate, etc.)
  (+) = Add
  (*) = Mul
  fromInteger = Con
  negate = negateE
negateE (Con c) = Con (negate c)
negateE _ = error "negate: not supported"
```

4.4 Exercises

Exercise 4.1. Complete the instance declarations for *FunExp* (for *Num*, *Fractional*, and *Floating*).

Exercise 4.2. Complete the instance declarations for (*Double*, *Double*), deriving them from the homomorphism requirement for *apply* (in section 4.2.3).

Exercise 4.3. 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 $x = \pi$.

a. Find $e :: FunExp$ such that $eval\ e = f$ and use $eval'$.

b. Find an expression of (the first version of) type *FD Double* and use *apply*.

c. Apply f directly to the appropriate (x, x') and use *snd*.

Do you get the same result?

Exercise 4.4. *From exam 2017-08-22*

In exercise 1.3 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.

a. Define a type class *SemiRing* that corresponds to the semiring structure.

b. Define a *SemiRing* instance for the datatype *SR v* that you defined in exercise 1.3.

c. Find two other instances of the *SemiRing* class.

d. Specialise the evaluator that you defined in exercise 1.3 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 4.5. *From exam 2016-03-15*

In exercise 1.4, 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.

a. Define a type class *Lattice* that corresponds to the lattice structure.

b. Define a *Lattice* instance for the datatype for lattice expressions that you defined in 1.4.1.

c. Find two other instances of the *Lattice* class.

d. 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 4.6. *From exam 2016-08-23*

In exercise 1.5, 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.

a. Define a type class *AbMonoid* that corresponds to the abelian monoid structure.

b. Define an *AbMonoid* instance for the datatype for abelian monoid expressions that you defined in exercise 1.5.1.

c. Find one other instance of the *AbMonoid* class and give an example which is *not* an instance of *AbMonoid*.

d. 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 4.7. (Closely related to exam question)

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

a. 0 is the neutral element of $+$

$$\forall\ x \in A.\ \ x + 0 = 0 + x = x$$

b. $+$ is associative

$$\forall\ x, y, z \in A.\ \ x + (y + z) = (x + y) + z$$

c. *negate* inverts elements with respect to addition

$$\forall\ x \in A.\ \ x + \mathit{negate}\ x = \mathit{negate}\ x + x = 0$$

d. $+$ is commutative

$$\forall\ x, y \in A.\ \ x + y = y + x$$

e. 1 is the unit of $*$

$$\forall\ x \in A.\ \ x * 1 = 1 * x = x$$

f. $*$ is associative

$$\forall\ x, y, z \in A.\ \ x * (y * z) = (x * y) * z$$

g. $*$ 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

a. Define a type class 'Ring' that corresponds to the ring structure.

b. Define a datatype for the language of ring expressions (including variables) and define a 'Ring' instance for it.

c. Find two other instances of the 'Ring' class.

d. Define a general evaluator for 'Ring' expressions on the basis of a given assignment function.

e. Specialise the evaluator to the two 'Ring' instances defined at point 3. Take three ring expressions, give the appropriate assignments and compute the results of evaluating, in each case, the three expressions.

Exercise 4.8. *From exam 2017-03-14*

Recall the type of expressions

```
data FunExp = Const Rational
            | Id
            | FunExp :+: FunExp
            | FunExp :*: FunExp
            | FunExp :/: FunExp
            | Exp FunExp
            | Sin FunExp
            | Cos FunExp
              -- and so on
            deriving Show
```

and consider the function

```
f :: Double -> Double
f x = exp (sin x) + x
```

a. Find an expression e such that *eval e* == f and show this using equational reasoning.

b. 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 4.9. 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 4.10. Write a function

$simplify :: FunExp \to FunExp$

to simplify the expression resulted from *derive*. For example

$simplify\ (Const\ 0 :*: Exp\ Id) = Const\ 0\ simplify\ (Const\ 0 :+: Exp\ Id) = Exp\ Id\ simplify\ (Const\ 2 :*: Const\ 1) = Const\ 2$

5 Polynomials and Power Series

```
{-# LANGUAGE TypeSynonymInstances #-}
module DSLsofMath.W05 where
```

5.1 Polynomials

From Adams and Essex [2010], page 55:

> 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 polymonial [misspelled in the book], 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

> $P : \mathbb{R} \to \mathbb{R}$, $x \in \mathbb{R}$, $a_0, \ldots, a_n \in \mathbb{R}$ 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 is either 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. Assuming the coefficients are given as

> $as = [a_0, a_1, ..., a_n]$

(we prefer counting up), then the evaluation function is written

```
eval :: [ℝ] → ℝ → ℝ
eval []     x = 0
eval (a:as) x = a + x * eval as x
```

Note that we can read the type as $eval :: [\mathbb{R}] \to (\mathbb{R} \to \mathbb{R})$ and thus identify $[\mathbb{R}]$ as the tyoe for the (abstract) syntax and $(\mathbb{R} \to \mathbb{R})$ as the type of the semantics. Exercise: Show that this evaluation function gives the same result as the formula above. Exercise: Use the *Num* instance for functions to rewrite *eval* into a one-argument function. As an example, the polynomial which is usually written just x is represented by the list $[0, 1]$ and the polynomial $\lambda x \to x\hat{}2 - 1$ is represented by the list $[-1, 0, 1]$.

It is worth noting that the definition of a what is called 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: $T_n(x) = \cos(n * \arccos(x))$. It can be shown that T_n is a polynomial function of degree n. Exercise: show this by induction on n using the rule for $cos(\alpha + \beta)$. Start by computing 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

$$\{[0]\} \cup \{(a : as) \mid last\ (a : as) \neq 0\}$$

We cannot express the $last\ (a : as) \neq 0$ in Haskell, but we can express the condition that the list should not be empty:

```
data Poly a = Single a | Cons a (Poly a)
            deriving (Eq, Ord)
```

(TODO: show the version and motivation for using just $[a]$ as well. Basically, one can use [] as the syntax for the "zero polynomial" and $(c : cs)$ for all other.)

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 :: [a] ->         Poly a
fromList (a : [])       = Single a
fromList (a0 : a1 : as) = Cons a0 (fromList (a1 : as))
instance Show a => Show (Poly a) where
  show = show . toList
```

Since we only use the arithmetical operations, we can generalise our evaluator:

```
evalPoly :: Num a => Poly a -> (a -> a)
evalPoly (Single a)   x = a
evalPoly (Cons a as) x = a + x * evalPoly as x
```

Since we have $Num\ a$, there is a Num structure on $a \to a$, and $evalPoly$ looks like a homomorphism. Question: is there a Num 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

$$\begin{aligned} & (evalPoly\ as + evalPoly\ bs)\ x = evalPoly\ (as + bs)\ x \\ \Leftrightarrow & \{\text{- + on functions is defined point-wise -}\} \\ & evalPoly\ as\ x + evalPoly\ bs\ x = evalPoly\ (as + bs)\ x \end{aligned}$$

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

$$(a : as) + (b : bs) = (a + b) : (as + bs)$$

We leave the derivation of the other cases and operations as an exercise. Here, we just give the corresponding definitions.

```
instance Num a ⇒ Num (Poly a) where
  (+) = polyAdd
  (∗) = polyMul
  negate = polyNeg
  fromInteger = Single ∘ fromInteger
polyAdd :: Num 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 :: Num 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 :: Num a ⇒ Poly a → Poly a
polyNeg = fmap negate
```

Therefore, we *can* define a ring structure (the mathematical counterpart of *Num*) 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 functions *evalPoly 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: what are the ring operations on $Poly'\ A$? For example, here is addition:

$$a + b = c \Leftrightarrow a\ n + b\ n = c\ n \quad \text{-- } \forall n : \mathbb{N}$$

Observations:

a. Polynomials are not, in general, isomorphic (in one-to-one correspondence) with 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

   $$evalPoly\ [0, 1, 1] = const\ 0 = evalPoly\ [0]\{\text{- in } \mathbb{Z}_2 \to \mathbb{Z}_2 \text{ -}\}$$

   but

   $$[0, 1, 1] \neq [0]\{\text{- in } Poly\ \mathbb{Z}_2 \text{ -}\}$$

   Therefore, it is not generally a good idea to confuse polynomials with polynomial functions.

b. In keeping with 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

   $$x :: Num\ a \Rightarrow Poly\ a$$
   $$x = Cons\ 0\ (Single\ 1)$$

   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: check this.

   This justifies the standard notation

   $$as = \sum_{i=0}^{n} a_i * x^i$$

5.2 Polynomial degree as a homomorphism

TODO: textify black board notes

It is often the case that a certain function is *almost* a homomorphism and the domain or range *almost* a monoid. In the section on *eval* and *eval′* for *FunExp* we have seen "tupling" as one way to fix such a problem and here we will introduce another way.

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 there is a z such that $degree\ 0 = z$ and that we have some 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 which is not a natural number. 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 = Nothing
  op   = opMaybe
opMaybe Nothing   m         = m
opMaybe m         Nothing   = m
opMaybe (Just m1) (Just m2) = Just (op m1 m2)
```

We quote the Haskell prelude implementation:

> Lift a semigroup into *Maybe* forming a *Monoid* according to `http://en.wikipedia.org/wiki/Monoid`: "Any semigroup S may be turned into a monoid simply by adjoining an element e not in S and defining $e * e = e$ and $e * s = s = s * e$ for all $s \in S$." Since there is no *Semigroup* typeclass [..], we use *Monoid* instead.

Thus, to sum up, *degree* is a monoid homomorphism from $(Poly\ a, 1, *)$ to $(Maybe\ \mathbb{N}, Nothing, opMaybe)$.

TODO: check all the properties.

5.3 Power Series

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

$eval\ a : A \to A$
$eval\ a\ x = lim\ s$ **where** $s\ n = \sum_{i=0}^{n} a_i * x^i$

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 functions). If $a \neq b$, then the power series are different, even if $eval\ a = eval\ b$.

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 :: Num 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:

$$\begin{array}{l} eval\ 2\ (x * x)\ 1 \quad = \\ evalPoly\ (takePoly\ 2\ [0,0,1])\ 1 = \\ evalPoly\ [0,0]\ 1 \quad = \\ 0 \end{array}$$

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 $\mathit{eval}\ 2\ (x * x)\ 1 = 0 \neq 1 = 1 * 1 = (\mathit{eval}\ 2\ x\ 1) * (\mathit{eval}\ 2\ x\ 1)$.

5.4 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.

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).

$$
\begin{array}{lll}
p & == [p_0, p_1, & p_2, \quad ... \\
x * p & == [0, \ p_0, & p_1, \quad ... \\
p - x * p & == [p_0, p_1 - p_0, & p_2 - p_1, \ ... \\
1 & == [1, \ 0, & 0, \quad ...
\end{array}
$$

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 $p\ /\ q$ with $p = a : as$, $q = b : bs$, we assume that $a * b \neq 0$. Then we want to find, for any given $(a : as)$ and $(b : bs)$, the series $(c : cs)$ satisfying

$$
\begin{array}{ll}
(a : as)\ /\ (b : bs) = (c : cs) & \Leftrightarrow \{\text{- def. of division -}\} \\
(a : as) = (c : cs) * (b : bs) & \Leftrightarrow \{\text{- def. of } * \text{ for } \mathit{Cons} \text{ -}\} \\
(a : as) = (c * b) : (cs * (b : bs) + [c] * bs) & \Leftrightarrow \{\text{- equality on compnents, def. of division -}\} \\
c \ = a\ /\ b \{\text{- and -}\} & \\
as = cs * (b : bs) + [c] * bs & \Leftrightarrow \{\text{- arithmetics -}\} \\
c \ = a\ /\ b \{\text{- and -}\} & \\
cs = (as - [c] * bs)\ /\ (b : bs) &
\end{array}
$$

This leads to the implementation:

```
instance (Eq a, Fractional a) ⇒ Fractional (PowerSeries a) where
  (/) = divPS
  fromRational = Single ∘ fromRational
divPS :: (Eq a, Fractional a) ⇒ PowerSeries a → PowerSeries a → PowerSeries a
```

$$
\begin{array}{lll}
divPS\ as & (Single\ b) & = as * Single\ (1\ /\ b) \\
divPS\ (Single\ 0) & (Cons\ b\ bs) & = Single\ 0 \\
divPS\ (Single\ a) & (Cons\ b\ bs) & = divPS\ (Cons\ a\ (Single\ 0))\ (Cons\ b\ bs) \\
divPS\ (Cons\ a\ as)\ (Cons\ b\ bs) & = Cons\ c\ (divPS\ (as - (Single\ c) * bs)\ (Cons\ b\ bs)) \\
 & \quad \textbf{where}\ c = a\ /\ b
\end{array}
$$

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:

$$
\begin{array}{l}
ps0, ps1, ps2 :: (Eq\ a, Fractional\ a) \Rightarrow PowerSeries\ a \\
ps0 = 1\ /\ (1 - x) \\
ps1 = 1\ /\ (1 - x)\hat{}2 \\
ps2 = (x\hat{}2 - 2 * x + 1)\ /\ (x - 1)
\end{array}
$$

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).

$$
\begin{array}{l}
example0 = takePoly\ 10\ ps0 \\
example01 = takePoly\ 10\ (ps0 * (1 - x))
\end{array}
$$

We can get a feeling for the definition by computing $ps0$ "by hand". We let $p = [1]$ and $q = [1, -1]$ and seek $r = p\ /\ q$.

$$
\begin{array}{ll}
divPS\ p\ q & = \\
divPS\ [1]\ (1 : [-1]) & = \{\text{- 3rd case -}\} \\
divPS\ (1 : [0])\ (1 : [-1]) & = \{\text{- 4th case -}\} \\
(1\ /\ 1) : divPS\ ([0] - [1] * [-1])\ (1 : [-1]) & \\
1 : divPS\ ([0] - [-1])\ (1 : [-1]) & \\
1 : divPS\ [1]\ (1 : [-1]) & \\
1 : divPS\ p\ q &
\end{array}
$$

Thus, the answer r starts with 1 and continues with r! In other words, we have that $1\ /\ [1, -1] = [1, 1..]$ as infinite lists of coefficients and $\frac{1}{1-x} = \sum_{i=0}^{\infty} x^i$ in the more traditional mathematical notation.

5.5 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:

$$
\begin{aligned}
\left(\sum_{n=0}^{\infty} a_n * x^n\right)' &= \sum_{n=0}^{\infty} (a_n * x^n)' = \sum_{n=0}^{\infty} a_n * (x^n)' = \sum_{n=0}^{\infty} a_n * (n * x^{n-1}) \\
&= \sum_{n=0}^{\infty} (n * a_n) * x^{n-1} = \sum_{n=1}^{\infty} (n * a_n) * x^{n-1} = \sum_{m=0}^{\infty} ((m+1) * a_{m+1}) * x^m
\end{aligned} \tag{1}
$$

Thus the mth coefficient of the derivative is $(m + 1) * a_{m+1}$.

TODO: redo to arrive at the recursive formulation.

We can implement this, for example, as

```
deriv (Single a)   = Single 0
deriv (Cons a as) = deriv' as 1
  where deriv' (Single a)   n = Single (n * a)
        deriv' (Cons a as) n = Cons  (n * a) (deriv' as (n + 1))
```

Side note: we cannot in general implement a Boolean equality test for PowerSeries. For example, we know that *deriv ps0* equals *ps1* but we cannot compute *True* in finite time by comparing the coefficients of the two power series.

```
checkDeriv :: Integer → Bool
checkDeriv n = takePoly n (deriv ps0) == takePoly n ps1
```

Recommended reading: the Functional pearl: "Power series, power serious" McIlroy [1999].

5.6 Helpers

```
instance Functor Poly where
  fmap = fmapPoly
fmapPoly :: (a → b) → (Poly a → Poly b)
fmapPoly f (Single a)   = Single (f a)
fmapPoly f (Cons a as) = Cons (f a) (fmapPoly f as)
po1 :: Num a ⇒ Poly a
po1 = 1 + x^2 − 3 * x^4
```

5.7 Exercises

Exercise 5.1. Fill in the gaps in lectures on polynomials and power series. In particular:

- Give a derivation for the definition of $*$ for polynomials
- 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 5.2. Complete the following definition

```
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 5.3. Polynomial multiplication. To get a feeling for the definition it can be useful to take it step by step, starting with some easy cases.

$$
\begin{aligned}
&\mathit{mulP}\ [\,]\ p = \quad \text{-- TODO} \\
&\mathit{mulP}\ p\ [\,] = \quad \text{-- TODO}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{mulP}\ [a]\ p = \quad \text{-- TODO} \\
&\mathit{mulP}\ p\ [b] = \quad \text{-- TODO}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{mulP}\ (0 : as)\ p = \quad \text{-- TODO} \\
&\mathit{mulP}\ p\ (0 : bs) = \quad \text{-- TODO}
\end{aligned}
$$

Finally we reach the main case

$$\mathit{mulP}\ (a : as)\ q@(b : bs) = \quad \text{-- TODO}$$

TODO: more exercises!

```
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE TypeSynonymInstances #-}
module DSLsofMath.W06 where
import DSLsofMath.W05
```

6 Higher-order Derivatives and their Applications

6.1 Review

- key notion *homomorphism*: $S1 \to S2$
- questions ("equations"):
  - $S1 \overset{?}{\leftarrow} S2$ what is the homomorphism between two given structures
    - e.g., $apply : Num\ a \to Num\ (x \to a)$
  - $S1? \to S2$ what is $S1$ compatible with a given homomorphism
    - e.g., $eval : Poly\ a \to (a \to a)$
  - $S1 \to S2?$ what is $S2$ compatible with a given homomorphism
    - e.g., $applyFD : FD\ a \to (a, a)$
  - $S1 \overset{?}{\leftarrow} S2?$ can we find a good structure on $S2$ so that it becomes homomorphic w. $S1$?
    - e.g., $evalD : FunExp \to FD\ a$
- The importance of the last two is 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 $f\ x = sin\ x + 2 * x$

We have: $f\ 0 = 0$, $f\ 2 = 4.909297426825682$, etc.

The type of f is $f :: Floating\ a \Rightarrow a \to a$.

How do we compute, say, $f'\ 2$?

We have several choices.

a. Using *FunExp*

Recall (section 3.7):

```
data FunExp = Const Rational
            | Id
            | FunExp :+: FunExp
            | FunExp :*: FunExp
            | FunExp :/: FunExp
            | Exp FunExp
            | Sin FunExp
            | Cos FunExp
              -- and so on
            deriving Show
```

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

$$f'\ 2 = eval\ (derive\ e)\ 2$$

This can hardly be called "automatic", look at all the work we did in deducing e!
However, consider this:

$$e = f\ Id$$

(Perhaps it would have been better to use, in the definition of *FunExp*, X instead of *Id*.)
In general, to find the value of the derivative of a function f at a given x, we can use

$$drv\ f\ x = evalFunExp\ (derive\ (f\ Id))\ x$$

b. Using *FD*
Recall

$$
\begin{aligned}
& \textbf{type}\ FD\ a = (a \to a, a \to a) \\
& applyFD\ (f, g)\ x = (f\ x, g\ x)
\end{aligned}
$$

The operations 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

$$f'\ 2 = snd\ (applyFD\ (f\ (g, g'))\ 2)$$

We can fullfill (*) if we can find a (g, g') that is a sort of "unit" for $FD\ a$:

$$
\begin{aligned}
& sin\ (g, g') = (sin, cos) \\
& exp\ (g, g') = (exp, exp)
\end{aligned}
$$

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

$$f'\ 2 = snd\ (applyFD\ (f\ (id, const\ 1))\ 2)$$

In general

$$drvFD\ f\ x = snd\ (applyFD\ (f\ (id, const\ 1))\ x)$$

computes the derivative of f at x.

$$f_1 :: FD\ Double \to FD\ Double$$
$$f_1 = f$$

c. Using pairs

We have **instance** *Floating* $a \Rightarrow$ *Floating* (a, a), moreover, the instance declaration looks exactly the same as that for *FD* a:

instance *Floating* $a \Rightarrow$ *Floating* $(FD\ a)$ **where**
$exp\ (f, f') = (exp\ f, (exp\ f) * f')$
$sin\ (f, f') = (sin\ f, (cos\ f) * f')$
$cos\ (f, f') = (cos\ f, -(sin\ f) * f')$

instance *Floating* $a \Rightarrow$ *Floating* (a, a) **where**
$exp\ (f, f') = (exp\ f, (exp\ f) * f')$
$sin\ (f, f') = (sin\ f, cos\ f * f')$
$cos\ (f, f') = (cos\ f, -(sin\ f) * f')$

In fact, the latter represents a generalisation of the former. It is also the "maximally general" such generalisation (discounting the "noise" generated by the less-than-clean design of *Num*, *Fractional*, *Floating*).

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

$$f\ (g, g') = (f\ 2, f'\ 2)$$
$$\Leftrightarrow$$
$$(f\ g, (f'\ g) * g') = (f\ 2, f'\ 2)$$
$$\Leftarrow$$
$$g = 2, g' = 1$$

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.

$$f_2 :: (Double, Double) \to (Double, Double)$$
$$f_2 = f$$

6.2 Higher-order derivatives

Consider

$$[f, f', f'', ...]$$

representing the evaluation of an expression and its derivatives:

$$evalAll\ e = (evalFunExp\ e) : evalAll\ (derive\ e)$$

Notice that, if

$$[f, f', f'', ...] = evalAll\ e$$

then

$$[f', f'', ...] = evalAll\ (derive\ e)$$

We want to define the 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 have, writing *eval* for *evalFunExp* in order to save ink

$$\begin{aligned}
&evalAll\ (e_1 :*: e_2) = evalAll\ e_1 * evalAll\ e_2 \\
\Leftrightarrow\ &\\
&eval\ (e_1 :*: e_2) : evalAll\ (derive\ (e_1 :*: e_2)) = \\
&eval\ e_1 : evalAll\ (derive\ e) * eval\ e_1 : evalAll\ (derive\ e_2) \\
\Leftrightarrow\ &\\
&(eval\ e_1 * eval\ e_2) : evalAll\ (derive\ (e_1 :*: e_2)) = \\
&eval\ e_1 : evalAll\ (derive\ e) * eval\ e_1 : evalAll\ (derive\ e_2) \\
\Leftrightarrow\ &\\
&(eval\ e_1 * eval\ e_2) : evalAll\ (derive\ e_1 :*: e_2 :+: e_1 * derive\ e_2) = \\
&eval\ e_1 : evalAll\ (derive\ e) * eval\ e_1 : evalAll\ (derive\ e_2) \\
\Leftarrow\ &\\
&(a : as) * (b : bs) = (a * b) : (as * (b : bs) + (a : as) * bs)
\end{aligned}$$

The final line represents the definition of $(*)$ needed for ensuring the conditions are met.

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*:

```
instance Num a ⇒ Num [a] where
  (a : as) + (b : bs) = (a + b) : (as + bs)
  (a : as) * (b : bs) = (a * b) : (as * (b : bs) + (a : as) * bs)
```

Exercise: complete the instance declarations for *Fractional* and *Floating*. Write a general derivative computation, similar to *drv* functions above:

```
drvList k f x = undefined   -- kth derivative of f at x
```

Exercise: Compare the efficiency of different ways of computing derivatives.

6.3 Polynomials

```
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 Power series

No need for a separate type in Haskell

```
type PowerSeries a = Poly a   -- finite and infinite non-empty lists
```

Now we can divide, as well as add and multiply.

We can also derive:

```
deriv (Single a)   = Single 0
deriv (Cons a as) = deriv' as 1
  where deriv' (Single a)   n = Single (n * a)
        deriv' (Cons a as) n = Cons  (n * a) (deriv' as (n + 1))
```

and integrate:

```
integ :: Fractional a ⇒ PowerSeries a → a → PowerSeries a
integ as a0 = Cons a0 (integ' as 1)
  where integ' (Single a)   n = Single (a / n)
        integ' (Cons a as) n = Cons  (a / n) (integ' as (n + 1))
```

Note that a_0 is the constant that we need due to indefinite integration.

All these definitions make sense, irrespective of convergence, hence "formal".

If the power series involved do converge, then *eval* is a morphism between the formal structure and that of the functions represented:

$$eval\ as + eval\ bs = eval\ (as + bs)$$
$$eval\ as * eval\ bs = eval\ (as * bs)$$
$$eval\ (derive\ as) = D\ (eval\ as)$$
$$eval\ (integ\ as\ c)\ x\ =\ \int_0^x (eval\ as\ t)\ dt + c$$

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 g.

The fundamental theorem of calculus gives us

$$f\ x = \int_0^x (g\ f\ t)\ dt + f_0$$

If $f = eval\ as$

$$eval\ as\ x = \int_0^x (g\ (eval\ as)\ t)\ dt + f_0$$

Assuming that g is a polymorphic function that commutes with *eval*

$$eval\ as\ x = \int_0^x (eval\ (g\ as)\ t)\ dt + f_0$$
$$eval\ as\ x = eval\ (integ\ (g\ as)\ f_0)\ x$$
$$as = integ\ (g\ as)\ f_0$$

Which functions g commute with *eval*? All the ones in *Num*, *Fractional*, *Floating*, by construction; additionally, as above, *deriv* and *integ*.

Therefore, we can implement a general solver for these simple equations:

```
solve :: Fractional a ⇒ (PowerSeries a → PowerSeries a) → a → PowerSeries a
solve g f0 = f    -- solves f' = g f, f 0 = f0
  where f = integ (g f) f0
idx = solve (λf → 1) 0
idf = eval 100 idx
expx = solve (λf → f) 1
expf = eval 100 expx
sinx = solve (λf → cosx) 0
cosx = solve (λf → −sinx) 1
sinf = eval 100 sinx
cosf = eval 100 cosx
idx, expx, sinx, cosx :: Fractional a ⇒ PowerSeries a
idf, expf, sinf, cosf :: Fractional a ⇒  a → a
```

6.6 The *Floating* structure of *PowerSeries*

Can we compute *exp as*?

Specification:

$$eval\ (exp\ as) = exp\ (eval\ as)$$

Differentiating both sides, we obtain

$$D\ (eval\ (exp\ as)) = exp\ (eval\ as) * D\ (eval\ as)$$
$$\Leftrightarrow \{\text{- } eval \text{ morphism -}\}$$

$$
\begin{aligned}
&\mathit{eval}\ (\mathit{deriv}\ (\mathit{exp}\ \mathit{as})) = \mathit{eval}\ (\mathit{exp}\ \mathit{as} * \mathit{deriv}\ \mathit{as}) \\
\Leftarrow\ & \\
&\mathit{deriv}\ (\mathit{exp}\ \mathit{as}) = \mathit{exp}\ \mathit{as} * \mathit{deriv}\ \mathit{as}
\end{aligned}
$$

Adding the "initial condition" $\mathit{eval}\ (\mathit{exp}\ \mathit{as})\ 0 = \mathit{exp}\ (\mathit{head}\ \mathit{as})$, we obtain

$$\mathit{exp}\ \mathit{as} = \mathit{integ}\ (\mathit{exp}\ \mathit{as} * \mathit{deriv}\ \mathit{as})\ (\mathit{exp}\ (\mathit{val}\ \mathit{as}))$$

Note: we cannot use *solve* here, because the g function uses both $\mathit{exp}\ \mathit{as}$ and as (it "looks inside" its argument).

$$
\begin{aligned}
&\textbf{instance}\ (\mathit{Eq}\ a, \mathit{Floating}\ a) \Rightarrow \mathit{Floating}\ (\mathit{PowerSeries}\ a)\ \textbf{where} \\
&\quad \pi \quad\ \ = \mathit{Single}\ \pi \\
&\quad \mathit{exp}\ \mathit{fs} = \mathit{integ}\ (\mathit{exp}\ \mathit{fs} \ \ * \mathit{deriv}\ \mathit{fs})\ (\mathit{exp}\ (\mathit{val}\ \mathit{fs})) \\
&\quad \mathit{sin}\ \mathit{fs} = \mathit{integ}\ (\mathit{cos}\ \mathit{fs} \ \ * \mathit{deriv}\ \mathit{fs})\ (\mathit{sin}\ \ (\mathit{val}\ \mathit{fs})) \\
&\quad \mathit{cos}\ \mathit{fs} = \mathit{integ}\ (-\mathit{sin}\ \mathit{fs} * \mathit{deriv}\ \mathit{fs})\ (\mathit{cos}\ \ (\mathit{val}\ \mathit{fs})) \\
&\mathit{val} :: \mathit{PowerSeries}\ a \to a \\
&\mathit{val}\ \ (\mathit{Single}\ a) \quad\ \ = a \\
&\mathit{val}\ \ (\mathit{Cons}\ a\ \mathit{as}) \quad = a
\end{aligned}
$$

In fact, we can implement *all* the operations needed for evaluating *FunExp* functions as power series!

$$
\begin{aligned}
&\mathit{evalP} :: (\mathit{Eq}\ r, \mathit{Floating}\ r) \Rightarrow \mathit{FunExp} \to \mathit{PowerSeries}\ r \\
&\mathit{evalP}\ (\mathit{Const}\ x) \ = \mathit{Single}\ (\mathit{fromRational}\ x) \\
&\mathit{evalP}\ (e_1 :+: e_2) = \mathit{evalP}\ e_1 + \mathit{evalP}\ e_2 \\
&\mathit{evalP}\ (e_1 :*: e_2) \ = \mathit{evalP}\ e_1 * \mathit{evalP}\ e_2 \\
&\mathit{evalP}\ (e_1 :/: e_2) \ = \mathit{evalP}\ e_1 \ / \ \mathit{evalP}\ e_2 \\
&\mathit{evalP}\ \mathit{Id} \qquad\ \ = \mathit{idx} \\
&\mathit{evalP}\ (\mathit{Exp}\ e) \quad = \mathit{exp}\ (\mathit{evalP}\ e) \\
&\mathit{evalP}\ (\mathit{Sin}\ e) \quad = \mathit{sin}\ (\mathit{evalP}\ e) \\
&\mathit{evalP}\ (\mathit{Cos}\ e) \quad = \mathit{cos}\ (\mathit{evalP}\ e)
\end{aligned}
$$

6.7 Taylor series

If $f = \mathit{eval}\ [a_0, a_1, ..., a_n, ...]$, then

$$
\begin{aligned}
&f\ 0 \ = a_0 \\
&f' \quad = \mathit{eval}\ (\mathit{deriv}\ [a_0, a_1, ..., a_n, ...]) \\
&\qquad = \mathit{eval}\ ([a_1, 2 * a_2, 3 * a_3, ..., n * a_n, ...]) \\
\Rightarrow\ & \\
&f'\ 0 \ = a_1 \\
&f'' \quad = \mathit{eval}\ (\mathit{deriv}\ [a_1, 2 * a_2, ..., n * a_n, ...]) \\
&\qquad = \mathit{eval}\ ([2 * a_2, 3 * 2 * a_3, ..., n * (n-1) * a_n, ...]) \\
\Rightarrow\ & \\
&f''\ 0 = 2 * a_2
\end{aligned}
$$

In general:

$$f^{(k)}0 = \mathit{fact}\ k * a_k$$

Therefore

$$f = eval\ [f\ 0, f'\ 0, f''\ 0\ /\ 2, ..., f^{(n)}0\ /\ (fact\ n), ...]$$

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.

Therefore, if we can represent f as a power series, we can find the value of all derivatives of f at 0!

```
derivs :: Num 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 = 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

```
instance Num a ⇒ Num (x → a) where
  f + g       = λx → f x + g x
  f − g       = λx → f x − g x
  f ∗ g       = λx → f x ∗ g x
  negate f    = negate ∘ f
  abs f       = abs ∘ f
  signum f    = signum ∘ f
  fromInteger = const ∘ fromInteger
instance Fractional a ⇒ Fractional (x → a) where
  recip f       = recip ∘ f
  fromRational = const ∘ fromRational
instance Floating a ⇒ Floating (x → a) where
  π      = const π
  exp f  = exp ∘ f
  sin f  = sin ∘ f
  cos f  = cos ∘ f
  f ∗∗ g = λx → (f x) ∗∗ (g x)
    -- and so on
evalFunExp :: Floating a ⇒ FunExp → a → a
evalFunExp (Const α)  = const (fromRational 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 Num FunExp where
  (+) = (:+:)
  (∗) = (:∗:)
  fromInteger n = Const (fromInteger n)
instance Fractional FunExp where
  (/) = (:/:)
instance Floating FunExp where
  exp = Exp
  sin = Sin
```

6.8.1 Not included to avoid overlapping instances

```
instance Num a ⇒ Num (FD a) where
  (f, f′) + (g, g′) = (f + g, f′ + g′)
```

$(f, f') * (g, g') = (f * g, f' * g + f * g')$
$fromInteger\ n = (fromInteger\ n, const\ 0)$

instance $Fractional\ a \Rightarrow Fractional\ (FD\ a)$ **where**
$(f, f')\ /\ (g, g') = (f\ /\ g, (f' * g - g' * f)\ /\ (g * g))$

instance $Floating\ a \Rightarrow Floating\ (FD\ a)$ **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 $Num\ a \Rightarrow Num\ (a, a)$ **where**
$(f, f') + (g, g') = (f + g, f' + g')$
$(f, f') * (g, g') = (f * g, f' * g + f * g')$
$fromInteger\ n = (fromInteger\ n, fromInteger\ 0)$

instance $Fractional\ a \Rightarrow Fractional\ (a, a)$ **where**
$(f, f')\ /\ (g, g') = (f\ /\ g, (f' * g - g' * f)\ /\ (g * g))$

instance $Floating\ a \Rightarrow Floating\ (a, a)$ **where**
$exp\ (f, f') = (exp\ f, (exp\ f) * f')$
$sin\ (f, f') = (sin\ f, cos\ f * f')$
$cos\ (f, f') = (cos\ f, -(sin\ f) * f')$

6.9 Exercises

Exercise 6.1. 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.

a. $f_1\ x = x\hat{}2 + 4$

b. $f_2\ x = 7 * exp\ (2 + 3 * x)$

c. $f_3\ x = 1\ /\ (sin\ x + cos\ x)$

Exercise 6.2. For each of the expressions $e :: FunExp$ you found in exercise 6.1, 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 6.3. 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:

a. Using *FunExp*

b. Using *FD*

c. Using pairs

Try using each of these methods to find the values of $f1'\ 2$, $f2'\ 2$, and $f3'\ 2$, i.e. the derivatives of each of the functions in exercise 6.1, evaluated at the point 2. You can verify that the result is correct by comparing it with the expressions $e1'$, $e2'$ and $e3'$ that you found in 6.2.

Exercise 6.4. 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.*

a. $\lambda t \to exp\ t$

b. $\lambda t \to exp\ (3 * t)$

c. $\lambda t \to 3 * exp\ (2 * t)$

Exercise 6.5. In the chapter, we saw that a representation $expx :: PowerSeries$ of the exponential function can be implemented using *solve* as $expx = solve\ (\lambda f \to f)\ 1$. Use the same method to implement power series representations of the following functions:

a. $\lambda t \to exp\ (3 * t)$

b. $\lambda t \to 3 * exp\ (2 * t)$

Exercise 6.6.

a. Implement idx', $sinx'$ and $cosx'$ using *solve*

b. Complete the instance $Floating\ (PowerSeries\ a)$

Exercise 6.7. 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.

a. Implement $expx3 :: PowerSeries$, a power series representation of e^{3*t}

b. Find an expression for fs'', the second derivative of fs, in terms of $expx3$, fs', and fs.

c. Find an expression for fs' in terms of fs'', using *integ*.

d. Find an expression for fs in terms of fs', using *integ*.

e. 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 6.8. *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 6.9. *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 6.10. *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 6.11. *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 $(6*) \circ \mathit{cos}$. What are the first four coefficients of *fs*?

Exercise 6.12. *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*?

```
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE UndecidableInstances #-}
module DSLsofMath.W07 where
```

7 Matrix algebra and linear transformations

Often, especially in engineering textbooks, one encounters the 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 suplemented 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* (analogous to being an instance of *Fractional*). Vectors can be added point-wise and multiplied with scalars, i.e., elements of the field:

$$v + w = \begin{bmatrix} v_0 \\ \vdots \\ v_n \end{bmatrix} + \begin{bmatrix} w_0 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} v_0 + w_0 \\ \vdots \\ v_n + w_n \end{bmatrix}$$

$$s * v = \begin{bmatrix} s * v_0 \\ \vdots \\ s * v_n \end{bmatrix}$$

The scalar s scales all the components of v.

But, as you might have guessed, the layout of the components on paper (in a column or row) is not the most important feature of a vector. In fact, the most important feature of vectors is that they can be *uniquely* expressed as a simple sort of combination of other vectors:

$$v = \begin{bmatrix} v_0 \\ \vdots \\ v_n \end{bmatrix} = v_0 * \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} + v_1 * \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix} + \cdots + v_n * \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

We denote by

$$e_k = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \leftarrow \text{position } k$$

the vector that is everywhere 0 except at position k, where it is 1, so that

$$v = v_0 * e_0 + ... + v_n * e_n$$

The algebraic structure that captures a set of vectors, with zero, addition, and scaling is called a *vector space*. For every field S of scalars and every set G of indeces, the set $Vector\ G = G \to S$ can be given a vector space structure.

There is a temptation to model vectors by lists or tuples, but a more general (and conceptually simpler) way is to view them as *functions* from a set of indeces G:

```
type S          = ...   -- the scalars, forming a field (ℝ, or Complex, or ℤ_n, etc.)
type Vector G = G → S
```

Usually, G is finite, i.e., *Bounded* and *Enumerable* and in the examples so far we have used indeces from $G = \{0, \ldots, n\}$. We sometime use *card* G to denote the *cardinality* of the set G, the number of elements ($n + 1$ in this case).

We know from the previous lectures that if S is an instance of *Num*, *Fractional*, etc. then so is $G \to S$, with the pointwise definitions. In particular, the instance declarations for $+$, multiplication, and embedding of constants, give us exactly the structure needed for the vector operations. For example

```
s * v                    = {- s is promoted to a function -}
const s * v              = {- Num instance definition -}
λg → (const s) g * v g   = {- definition of const -}
λg → s * v g
```

The basis vectors are then

$$e\ i : G \to S, e\ i\ g = i\ `is`\ g$$

Implementation:

```
is :: Num s ⇒ Int → Int → s
is a b = if a == b then 1 else 0
e :: Num s ⇒ G → (G → s)
e (G g) = λ(G g') → g `is` g'
toL v = [v g | g ← [minBound .. maxBound]]   -- so we can actually see them
```

and every

$$v : G \to S$$

is trivially a linear combination of vectors $e\ i$:

$$v = v\ 0 * e\ 0 + ... + v\ n * e\ n$$

7.1 Functions on vectors

As we have seen in earlier chapters, morphisms between structures are often importan. Vector spaces are no different: if we have two vector spaces *Vector* G and *Vector* G' (for the same set of scalars S) we can study functions $f : Vector\ G \to Vector\ G'$:

$$f\ v = f\ (v\ 0 * e\ 0 + ... + v\ n * e\ n)$$

For f to be a "good" function it should translate the operations in *Vector* G into operations in *Vector* G', i.e., should be a homomorphism:

$$f\ v = f\ (v\ 0 * e\ 0 + ... + v\ n * e\ n) = v\ 0 * f\ (e\ 0) + ... + v\ n * f\ (e\ n)$$

But this means that we can determine the values of $f : (G \to S) \to (G' \to S)$ from just the values of $f \circ e : G \to (G' \to S)$, a much "smaller" function. Let $m = f \circ e$. Then

$$f\ v = v\ 0 * m\ 0 + ... + v\ n * m\ n$$

Each of $m\ k$ is a *Vector* G', as is the resulting $f\ v$. We have

$$\begin{aligned}
&f\ v\ g' && = \{\text{- as above -}\}\\
&(v\ 0 * m\ 0 + ... + v\ n * m\ n)\ g' && = \{\text{- } * \text{ and } + \text{ for functions are def. pointwise -}\}\\
&v\ 0 * m\ 0\ g' + ... + v\ n * m\ n\ g' && = \{\text{- using } sum \text{ -}\}\\
&sum\ [v\ i * m\ i\ g' \mid i \leftarrow [minBound\,..\,maxBound]]
\end{aligned}$$

Implementation:

This is the almost the standard vector-matrix multiplication:

$$M = [m\ 0 \mid ... \mid m\ n]$$

The columns of M are the images of the canonical base vectors $e\ i$ through f. Every $m\ k$ has *card* G' rows, and it has become standard to use $M\ i\ j$ to mean the ith element of the jth column, i.e., $m\ j\ i$, so that

$$(M * v)\ i = sum\ [M\ i\ j * v\ j \mid j \leftarrow [0\,..\,n]]$$

$$mul\ m\ v\ g' = sum\ [m\ g'\ g * v\ g \mid g \leftarrow [minBound\,..\,maxBound]]$$

Example:

$$(M * e\ k)\ i = sum\ [M\ i\ j * e\ k\ j \mid j \leftarrow [0\,..\,n]] = sum\ [M\ i\ k] = M\ i\ k$$

i.e., $e\ k$ extracts the kth column from M (hence the notation "e" for "extract").

We have seen how a homomorphism 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: compute $((M*) \circ e)\ g\ g'$.

Therefore, every linear transformation is of the form $(M*)$ and every $(M*)$ is a linear transformation.

Matrix-matrix multiplication is defined in order to ensure that

$$(M' * M) * v = M' * (M * v)$$

that is

$$((M' * M)*) = (M'*) \circ (M*)$$

Exercise: work this out in detail.

Exercise: show that matrix-matrix multiplication is associative.

Perhaps the simplest 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 argument, therefore have exactly one value: $s\ ()$, so they are often identified 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 $w :: Vector\ G$:

$$M * w = w\ 0 * fv\ 0 + ... + w\ n * fv\ n$$

$M * v$ 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 v and w.

Remark: I have not discussed the geometrical point of view. For the connection between matrices, linear transformations, and geometry, I warmly recommend binge-watching the "Essence of linear algebra" videos on youtube (start here: `https://www.youtube.com/watch?v=kjBOesZCoqc`).

7.2 Examples of matrix algebra

7.2.1 Derivative

We have represented polynomials of degree $n + 1$ by the list of their coefficients. This is quite similar to standard geometrical vectors represented by $n + 1$ coordinates. This suggests that polynomials of degree $n + 1$ 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 $+$ and $*$ are defined in the same way as they are for functions.

The *derive* function takes polynomials of degree $n + 1$ to polynomials of degree n, and since $D\ (f + g) = D\ f + D\ g$ and $D\ (s * f) = s * D\ f$, we expect it to be a linear transformation. What is its associated matrix?

To answer that, we must first determine the canonical base vectors. As for geometrical vectors, they are

$$e\ i : \{0, ..., n\} \to Real, e\ i\ j = i\ `is`\ j$$

The evaluation of $e\ i$ returns the function $\lambda x \to x\hat{}i$, as expected.

The associated matrix will be

$$M = [D\ (e\ 0), D\ (e\ 1), ..., D\ (e\ n)]$$

where each $D\ (e\ i)$ has length n. Vector $e\ (i+1)$ represents $x\hat{}(i+1)$, therefore

$$D\ (e\ (i+1)) = (i+1) * x\hat{}i$$

i.e.

$$D\ (e\ (i+1))\ !!\ j = \textbf{if}\ i == j\ \textbf{then}\ i+1\ \textbf{else}\ 0$$

and

$$D\ (e\ 0) = 0$$

Example: $n+1 = 3$:

$$M = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

Take the polynomial

$$3 * X\hat{}2 + 2 * X + 1$$

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 $6 * x + 2$.

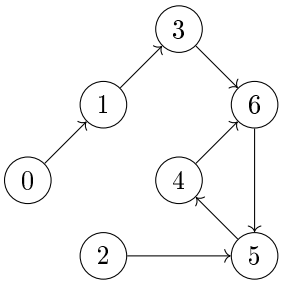
Exercise: write the (infinite-dimensional) matrix representing D for power series.

Exercise: write the matrix associated with integration of polynomials.

7.2.2 Simple deterministic systems (transition systems)

Simple deterministic systems are given by endo-functions[5] on a finite set $f : G \to G$. They can often be conveniently represented as a graph, for example

[5] 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 $f\ i = j$. Since f 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 avoid this is to extend the type of the functions to $\mathbb{R}$.

The canonical basis vectors are, as usual, $e\ i = \lambda j \to i\text{`}is\text{`}j$. Each $e\ i$ is the characteristic function of a singleton set, $\{i\}$. Thus, the inputs to f are canonical vectors.

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:

$$M = \begin{array}{c} \\ r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \end{array} \begin{array}{c} \begin{array}{ccccccc} c_0 & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Starting with a canonical base vector $e\ i$, we obtain $M * e\ i = e\ (f\ i)$, as we would expect.

It is more interesting if we start with a non-base vector. For example, $e\ 2 + e\ 4$, which represents the subset $\{2, 4\}$.

The more interesting thing is if we start with something different from a basis vector, say $[0, 1, 1]$. We obtain $\{f\ 2, f\ 4\} = \{5, 6\}$, the image of $\{2, 4\}$ through f. In a sense, we can say that the two computations were done 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: $[0, 0, 0, 0, 0, 0, 2]$. 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 gets shifted around, but in our case, we get a more interesting 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*. The problem is that $(\{0, 1\}, max, min)$ is not a field, and neither is $(\mathbb{R}, max, min)$. This 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.

In the example above, we have:

```
newtype G = G Int deriving (Eq, Show)
instance Bounded G where
  minBound = G 0
  maxBound = G 6
instance Enum G where
  toEnum           = G
  fromEnum (G n) = n
instance Num G where
  fromInteger = G ∘ fromInteger
    -- Note that this is just for convient notation (integer literals),
    -- G should normally not be used with
```

The transition function:

$$
\begin{aligned}
f_1\ 0 &= 1\\
f_1\ 1 &= 3\\
f_1\ 2 &= 5\\
f_1\ 3 &= 6\\
f_1\ 4 &= 6\\
f_1\ 5 &= 4\\
f_1\ 6 &= 5
\end{aligned}
$$

The associated matrix:

$$m_1\ (G\ g')\ (G\ g) = g'\ `is`\ f_1\ g$$

Test:

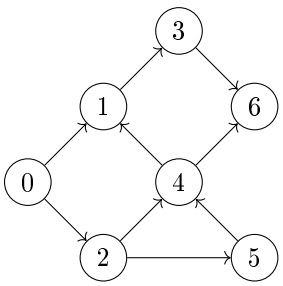
$$
\begin{aligned}
t1' &= mul\ m_1\ (e\ 3 + e\ 4)\\
t_1 &= toL\ t1'
\end{aligned}
$$

7.2.3 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).

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 and 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: 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 origin vector? Now invert the arrow from 2 to 4 and repeat the exercise. What changes? Can you prove it?

Implementation:

The transition function has type $G \to (G \to \mathit{Bool})$:

$$\begin{aligned} &f_2\ 0\ g = g == 1 \vee g == 2 \\ &f_2\ 1\ g = g == 3 \\ &f_2\ 2\ g = g == 4 \vee g == 5 \\ &f_2\ 3\ g = g == 6 \\ &f_2\ 4\ g = g == 1 \vee g == 6 \\ &f_2\ 5\ g = g == 4 \\ &f_2\ 6\ g = \mathit{False} \end{aligned}$$

The associated matrix:

$$m_2\ (G\ g')\ (G\ g) = f_2\ g\ g'$$

We need a *Num* instance for *Bool* (not a field!):

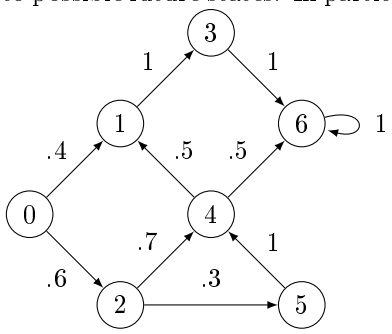
$$\begin{aligned} &\textbf{instance}\ \mathit{Num}\ \mathit{Bool}\ \textbf{where} \\ &\quad (+) = (\vee) \\ &\quad (*)\ = (\wedge) \\ &\quad \mathit{fromInteger}\ 0 = \mathit{False} \\ &\quad \mathit{fromInteger}\ 1 = \mathit{True} \\ &\quad \mathit{negate} \quad\quad = \neg \\ &\quad \mathit{abs} \quad\quad = \mathit{id} \\ &\quad \mathit{signum} \quad\quad = \mathit{id} \end{aligned}$$

Test:

$$\begin{aligned} &t2' = \mathit{mul}\ m_2\ (e\ 3 + e\ 4) \\ &t_2 = \mathit{toL}\ t2' \end{aligned}$$

7.2.4 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 possiblities. 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

$$sum\ [p\ g \mid g \leftarrow [0\,..\,6]] = 1$$

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

$$p\ a = sum\ [p\ (a \mid b) * p\ b \mid b \leftarrow [0\,..\,6]]$$

This formula in itself would be worth a lecture. For one thing, 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. For another, 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. I do not know how this notation came about: it is neither in Bayes' memoir, nor in Kolmogorov's monograph.

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 $e\ i = \lambda j \to i\ `is`\ j$ is the probability distribution *concentrated* in i:

$$M = \begin{array}{c} \\ r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \end{array} \begin{array}{c} \begin{array}{ccccccc} c_0 & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ .4 & 0 & 0 & 0 & .5 & 0 & 0 \\ .6 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & .7 & 0 & 0 & 1 & 0 \\ 0 & 0 & .3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & .5 & 0 & 1 \end{pmatrix} \end{array}$$

Exercise: 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?

Exercise: Implement the example. You will need to define:

The transition function

$$
\begin{array}{ll}
f_3 :: G \to (G \to \mathit{Double}) & \text{-- but we want only } G \to (G \to [0,1]), \text{ the unit interval} \\
f_3\ g\ g' = \mathit{undefined} & \text{-- the probability of getting to } g' \text{ from } g
\end{array}
$$

The associated matrix

$$
\begin{array}{l}
m_3 :: G \to (G \to \mathit{Double}) \\
m_3\ g'\ g = \mathit{undefined}
\end{array}
$$

Test

$$t_3 = \mathit{toL}\ (\mathit{mul}\ m_3\ (e\ 2 + e\ 4))$$

7.3 Monadic dynamical systems

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 states: 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 $f : G \to (G \to [0,1])$, the structure of the target is the probability distributions over G.

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 $\mathit{Prob}\ G = G \to [0,1]$

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:

$$
\begin{array}{ll}
M * v & \text{-- v represents the current possible states} \\
= \{\text{- } v \text{ is a linear combination of the base vectors -}\}
\end{array}
$$

$$
\begin{aligned}
& M * (v\ 0 * e\ 0 + ... + v\ n * e\ n) \\
= & \{\text{- homomorphism -}\} \\
& v\ 0 * (M * e\ 0) + ... + v\ n * (M * e\ n) \\
= & \{\text{- } e\ i \text{ represents the perfectly known current state } i\text{, therefore } M * e\ i = f\ i \text{ -}\} \\
& v\ 0 * f\ 0 + ... + v\ n * f\ n
\end{aligned}
$$

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 $*$).

If you examine the types of the operations involved

$$e : G \to Possible\ G$$

and

$$Possible\ G \to (G \to Possible\ G) \to Possible\ G$$

you see that they are very similar to the monadic operations

$$
\begin{aligned}
& return : g \to m\ g \\
& (\gg\!=) \ \ : m\ g \to (g \to m\ g') \to m\ g'
\end{aligned}
$$

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*.

7.4 The monad of linear algebra

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.

```
type S          = Double
data Vector g = V (g -> S)
toF (V v)       = v
class      (Bounded a, Enum a, Eq a) => Finite a where
instance (Bounded a, Enum a, Eq a) => Finite a where
class FinFunc f where
  func :: (Finite a, Finite b) => (a -> b) -> f a -> f b
instance FinFunc Vector where
  func = funcV
funcV :: (Finite g, Eq g') => (g -> g') -> Vector g -> Vector g'
funcV f (V v) = V (\g' -> sum [v g | g <- [minBound .. maxBound], g' == f g])
class FinMon f where
  embed  :: Finite a => a -> f a
```

```
  bind     :: (Finite a, Finite b) ⇒ f a → (a → f b) → f b
instance FinMon Vector where
  embed a       = V (λa' → if a == a' then 1 else 0)
  bind (V v) f = V (λg' → sum [toF (f g) g' * v g | g ← [minBound .. maxBound]])
```

A better implementation, using associated types, is in file *Vector.lhs* in the repository.

Exercises:

a. Prove that the functor laws hold, i.e.

$$\begin{aligned} func\ id &= id \\ func\ (g \circ f) &= func\ g \circ func\ f \end{aligned}$$

b. Prove that the monad laws hold, i.e.

$$\begin{aligned} bind\ v\ return &= v \\ bind\ (return\ g)\ f &= f\ g \\ bind\ (bind\ v\ f)\ h &= bind\ v\ (\lambda g' \to bind\ (f\ g')\ h) \end{aligned}$$

c. 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.

7.5 Associated code

TODO: import from suitable earlier lecture.

```
instance Num a ⇒ Num (x → a) where
  f + g         = λx → f x + g x
  f − g         = λx → f x − g x
  f * g         = λx → f x * g x
  negate f      = negate ∘ f
  abs f         = abs ∘ f
  signum f      = signum ∘ f
  fromInteger   = const ∘ fromInteger
instance Fractional a ⇒ Fractional (x → a) where
  recip f         = recip ∘ f
  fromRational    = const ∘ fromRational
instance Floating a ⇒ Floating (x → a) where
  π       = const π
  exp f   = exp ∘ f
  sin f   = sin ∘ f
  cos f   = cos ∘ f
  f ** g  = λx → (f x) ** (g x)
    -- and so on
```

7.6 Exercises

Search the chapter for tasks marked "Exercise".

TODO: Maybe convert these to proper exercises.

8 Exponentials and Laplace

8.1 The Exponential Function

```
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE TypeSynonymInstances #-}
module DSLsofMath.W08 where
import DSLsofMath.W05
import DSLsofMath.W06
```

One of the classical analysis textbooks, Rudin's 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 have defined the exponential function as the function represented by the power series

```
expx :: Fractional a ⇒ PowerSeries a
expx = integ expx 1
```

and approximated by

```
expf :: Fractional a ⇒ a → a
expf = eval 100 expx
```

It is easy to see, using the definition of *integ* that the power series *expx* is, indeed

$$expx = [1, 1\ /\ 2, 1\ /\ (2 * 3), ..., 1\ /\ (2 * 3 * ... * n), ..]$$

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 remind the basic operations on complex numbers.

As we saw in week 1, complex values can be represented as pairs of real values.

```
newtype Complex r = C (r, r) deriving (Eq, Show)
i :: Num a ⇒ Complex a
i = C (0, 1)
```

Now, we have, for example

```
ex1 :: Fractional a ⇒ Complex a
ex1 = expf i
```

We have $ex1 = C\ (0.5403023058681398, 0.8414709848078965)$. Note that

$$
\begin{aligned}
&\mathit{cosf}\ 1 = 0.5403023058681398\\
&\mathit{sinf}\ 1 = 0.8414709848078965
\end{aligned}
$$

and therefore $\mathit{expf}\ i == C\ (\mathit{cosf}\ 1, \mathit{sinf}\ 1)$. Coincidence?

Instead of evaluating the sum of the terms $a_n * z\hat{}n$, let us instead collect the terms in a series:

$$
\begin{aligned}
&\mathit{terms}\ \mathit{as}\ z = \mathit{terms1}\ \mathit{as}\ z\ 0\ \textbf{where}\\
&\quad \mathit{terms1}\ (\mathit{Cons}\ a\ \mathit{as})\ z\ n = \mathit{Cons}\ (a * z\hat{}n)\ (\mathit{terms1}\ \mathit{as}\ z\ (n+1))
\end{aligned}
$$

We obtain

$$
\begin{aligned}
&\mathit{ex2} :: \mathit{Fractional}\ a \Rightarrow \mathit{PowerSeries}\ (\mathit{Complex}\ a)\\
&\mathit{ex2} = \mathit{takePoly}\ 10\ (\mathit{terms}\ \mathit{expx}\ i)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{ex2} = [\,&C\ (1.0, && 0.0), && C\ (0.0, 1.0 &&)\\
,\,&C\ (-0.5, && 0.0), && C\ (0.0, -0.16666666666666666 &&)\\
,\,&C\ (4.1666666666666664\mathrm{e}{-2}, && 0.0), && C\ (0.0, 8.333333333333333\mathrm{e}{-3} &&)\\
,\,&C\ (-1.3888888888888887\mathrm{e}{-3}, && 0.0), && C\ (0.0, -1.9841269841269839\mathrm{e}{-4} &&)\\
,\,&C\ (2.4801587301587298\mathrm{e}{-5}, && 0.0), && C\ (0.0, 2.7557319223985884\mathrm{e}{-6} &&)\\
]&
\end{aligned}
$$

We can see that the real part of this series is the same as

$$\mathit{ex2R} = \mathit{takePoly}\ 10\ (\mathit{terms}\ \mathit{cosx}\ 1)$$

and the imaginary part is the same as

$$\mathit{ex2I} = \mathit{takePoly}\ 10\ (\mathit{terms}\ \mathit{sinx}\ 1)$$

(within approx 20 decimals). But the terms of a series evaluated at 1 are the coefficients of the series. Therefore, the coefficients of *cosx* are

$$[1, 0, -1\ /\ 2!, 0, 1\ /\ 4!, 0, -1\ /\ 6!, ...]$$

i.e. The function representation of the coefficients for *cos* is

$$
\begin{aligned}
&\mathit{cosa}\ (2 * n) &&= (-1)\hat{}n\ /\ (2 * n)\,!\\
&\mathit{cosa}\ (2 * n + 1) &&= 0
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{sina}\ (2 * n) &&= 0\\
&\mathit{sina}\ (2 * n + 1) &&= (-1)\hat{}n\ /\ (2 * n + 1)\,!
\end{aligned}
$$

This can be proven from the definitions of *cosx* and *sinx*. From this we obtain *Euler's formula*:

$$\mathit{exp}\ (i * x) = \mathit{cos}\ x + i * \mathit{sin}\ x$$

One thing which comes out of Euler's formula is the fact that the exponential is a *periodic function*. A function $f : A \to B$ is said to be periodic if there exists $T \in A$ such that

$$f\ x = f\ (x + T) \quad \text{-- } \forall x \in A$$

(therefore, for this definition to make sense, we need addition on A; in fact we normally assume at least group structure, i.e., addition and subtraction).

Since *sin* and *cos* are periodic, with period $2 * \pi$, we have, using the standard notation $a + i * b$ for some $z = C\ (a, b)$:

$$\begin{aligned}
&e\hat{}(z + 2 * \pi * i) && = \{\text{- Def. of } z \text{ -}\} \\
&e\hat{}((a + i * b) + 2 * \pi * i) && = \{\text{- Rearranging -}\} \\
&e\hat{}(a + i * (b + 2 * \pi)) && = \{\text{- } exp \text{ is a homomorphism from } (+) \text{ to } (*) \text{ -}\} \\
&e\hat{}a * e\hat{}(i * (b + 2 * \pi)) && = \{\text{- Euler's formula -}\} \\
&e\hat{}a * (cos\ (b + 2 * \pi) + i * sin\ (b + 2 * \pi)) && = \{\text{- } cos \text{ and } sin \text{ are } 2 * \pi\text{-periodic -}\} \\
&e\hat{}a * (cos\ b + i * sin\ b) && = \{\text{- Euler's formula -}\} \\
&e\hat{}a * e\hat{}(i * b) && = \{\text{- } exp \text{ is a homomorphism -}\} \\
&e\hat{}(a + i * b) && = \{\text{- Def. of } z \text{ -}\} \\
&e\hat{}z
\end{aligned}$$

Thus, we see that *exp* is periodic, because $exp\ z = exp\ (z + T)$ with $T = 2 * \pi * i$, for all z.

8.1.1 Exponential function: Associated code

TODO: Perhaps import from W01

```
instance Num r ⇒ Num (Complex r) where
  (+) = addC
  (*) = mulC
  fromInteger n = C (fromInteger n, 0)
  -- abs = absC – requires Floating r as context
addC :: Num r ⇒ Complex r → Complex r → Complex r
addC (C (a, b)) (C (x, y)) = C ((a + x), (b + y))
mulC :: Num r ⇒ Complex r → Complex r → Complex r
mulC (C (ar, ai)) (C (br, bi)) = C (ar * br − ai * bi, ar * bi + ai * br)
modulusSquaredC :: Num r ⇒ Complex r → r
modulusSquaredC (C (x, y)) = x^2 + y^2
absC :: Floating r ⇒ Complex r → Complex r
absC c = C (√(modulusSquaredC c), 0)
scale :: Num r ⇒ r → Complex r → Complex r
scale a (C (x, y)) = C (a * x, a * y)
conj :: Num r ⇒ Complex r → Complex r
conj (C (x, x'))      = C (x, −x')
instance Fractional r ⇒ Fractional (Complex r) where
  (/) = divC
  fromRational r = C (fromRational r, 0)
divC :: Fractional a ⇒ Complex a → Complex a → Complex a
divC x y = scale (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:

$$\begin{aligned} &fs = integ \; fs' \; 1 \\ &\quad \textbf{where } fs' = integ \; (exp \; (3 * x) + 3 * fs' - 2 * fs) \; 0 \end{aligned}$$

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 function $a : \mathbb{N} \to \mathbb{R}$ (or finding a list of sufficiently many a-values for a good approximation).

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:

$$a : \mathbb{N} \to \mathbb{R} \xrightarrow{\sum a_n * x^n} f : \mathbb{R} \to \mathbb{R}$$

We would like to go one step further

$$a : \mathbb{N} \to \mathbb{R} \xrightarrow{\sum a_n * x^n} f : \mathbb{R} \to \mathbb{R} \xrightarrow{??} F{:}?$$

That is, we are looking for a transformation of 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

$$a : \mathbb{N} \to \mathbb{R} \xrightarrow{\sum a_n * x^n} f : \mathbb{R} \to \mathbb{R} \xrightarrow{\int (ft) * x^t dt} F : \{ \, z \mid |z| < 1 \, \} \to \mathbb{C}$$

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

$$\begin{aligned} &\mathscr{L} \; f' \; x && = \{\text{- } g \; t = x \hat{} t; \; g' \; t = log \; x * x \hat{} t \; \text{-}\} \\ &\int_0^\infty \; (D \; (f \; t * x \hat{} t)) - f \; t * log \; x * x \hat{} t \; dt && = \end{aligned}$$

$$\int_0^\infty (D\ (f\ t * x\hat{}t))\ dt - \int_0^\infty f\ t * log\ x * x\hat{}t\ dt =$$
$$lim_{t\to\infty}(f\ t * x\hat{}t) - (f\ 0 * x\hat{}0) - log\ x * \int_0^\infty f\ t * x\hat{}t\ dt =$$
$$-f\ 0 - log\ x * \int_0^\infty f\ t * x\hat{}t\ dt =$$
$$-f\ 0 - log\ x * \mathscr{L}\ f\ x$$

The factor $log\ x$ is somewhat awkward. Let us therefore return to the definition of $\mathscr{L}$ and operate a change of variables:

$$\mathscr{L}\ f\ x = \int_0^\infty (f\ t) * x\hat{}t\ dt \qquad \Leftrightarrow \{\text{- } x = exp\ (log\ x)\ \text{-}\}$$
$$\mathscr{L}\ f\ x = \int_0^\infty (f\ t) * (exp\ (log\ x))\hat{}t\ dt \Leftrightarrow \{\text{- } (a\hat{}b)\hat{}c = a\hat{}(b * c)\ \text{-}\}$$
$$\mathscr{L}\ f\ x = \int_0^\infty (f\ t) * exp\ (log\ x * t)\ dt$$

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

$$\mathscr{L}\ f'\ s \qquad = \{\text{- The computation above with } s = -log\ x.\ \text{-}\}$$
$$-f\ 0 + s * \mathscr{L}\ f\ s$$

We have obtained

$$\mathscr{L}\ f'\ s = s * \mathscr{L}\ f\ s - f\ 0$$

From this, we can deduce

$$\mathscr{L}\ f''\ s \qquad =$$
$$s * \mathscr{L}\ f'\ s - f'\ 0 \qquad =$$
$$s * (s * \mathscr{L}\ f\ s - f\ 0) - f'\ 0 =$$
$$s\hat{}2 * \mathscr{L}\ f\ s - s * f\ 0 - f'\ 0$$

Exercise: what is the general formula for $\mathscr{L}\ f^{(k)}s$?

Returning to our differential equation, we have

$$f''\ x - 3 * f'\ x + 2 * f\ x = exp\ (3 * x), f\ 0 = 1, f'\ 0 = 0$$
$$\Leftrightarrow \{\text{- point-free form -}\}$$
$$f'' - 3 * f' + 2 * f = exp \circ (3*), f\ 0 = 1, f'\ 0 = 0$$
$$\Rightarrow \{\text{- applying } \mathscr{L} \text{ to both sides -}\}$$
$$\mathscr{L}\ (f'' - 3 * f' + 2 * f) = \mathscr{L}\ (exp \circ (3*)), f\ 0 = 1, f'\ 0 = 0 \quad \text{-- Eq. (1)}$$

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 $\mathbb{R}_{\geqslant 0}$. 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.

For the definition of $\mathscr{L}$ and 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

$$\mathscr{L}\ (\alpha * f + \beta * g) = \alpha * \mathscr{L}\ f + \beta * \mathscr{L}\ g$$

Note that this is an equality between functions. (Comparing to last week we can also see f and g as vectors and $\mathscr{L}$ as a linear transformation.)

Applying this to the left-hand side of (1), we have for any s

$$\begin{array}{l}
\quad \mathscr{L}\ (f'' - 3 * f' + 2 * f)\ s \\
= \{\text{- } \mathscr{L} \text{ is linear -}\} \\
\quad \mathscr{L}\ f''\ s - 3 * \mathscr{L}\ f'\ s + 2 * \mathscr{L}\ f\ s \\
= \{\text{- re-writing } \mathscr{L}\ f'' \text{ and } \mathscr{L}\ f' \text{ in terms of } \mathscr{L}\ f \text{ -}\} \\
\quad s\hat{}2 * \mathscr{L}\ f\ s - s * f\ 0 - f'\ 0 - 3 * (s * \mathscr{L}\ f\ s - f\ 0) + 2 * \mathscr{L}\ f\ s \\
= \{\text{- } f\ 0 = 1,\ f'\ 0 = 0 \text{ -}\} \\
\quad (s\hat{}2 - 3 * s + 2) * \mathscr{L}\ f\ s - s + 3
\end{array}$$

For the right-hand side, we apply the definition:

$$\begin{array}{ll}
\mathscr{L}\ (exp \circ (3*))\ s & = \{\text{- Def. of } \mathscr{L} \text{ -}\} \\
\int_0^\infty\ exp\ (3 * t) * exp\ (-s * t)\ dt & = \\
\int_0^\infty\ exp\ ((3 - s) * t))\ dt & = \\
lim_{t\to\infty} \frac{exp\ ((3-s)*t)}{3-s} - \frac{exp\ ((3-s)*0)}{3-s} & = \{\text{- for } s > 3 \text{ -}\} \\
\frac{1}{s-3} &
\end{array}$$

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

$$\begin{array}{ll}
F\ s & = \{\text{- Solve for } F\ s \text{ -}\} \\
\frac{\frac{1}{s-3}+s-3}{s\hat{}2-3*s+2} & = \{\text{- } s\hat{}2 - 3 * s + 2 = (s-1) * (s-2) \text{ -}\} \\
\frac{10-6*s+s\hat{}2}{(s-1)*(s-2)*(s-3)} &
\end{array}$$

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

$$\begin{array}{l}
A\ /\ (s-1) + B\ /\ (s-2) + C\ /\ (s-3) = (10 - 6 * s + s\hat{}2)\ /\ ((s-1) * (s-2) * (s-3)) \\
\Rightarrow \\
A * (s-2) * (s-3) + B * (s-1) * (s-3) + C * (s-1) * (s-2) = 10 - 6 * s + s\hat{}2 \quad \text{-- (2)}
\end{array}$$

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.

TODO: cite http://www.math.chalmers.se/Math/Grundutb/CTH/mve025/1516/Dokument/F-analys.pdf

(Fourier is a common tool in courses on Transforms, Signals and Systems.)

8.4 Exercises

Exercise 8.1. Find the Laplace transforms of the following functions:

a. $\lambda t.\, 3 * e^{5*t}$

b. $\lambda t.\, e^{\alpha * t} - \beta$

c. $\lambda t.\, e^{(t + \frac{\pi}{6})}$

Exercise 8.2.

a. Show that:

   (a) $\sin t = \frac{1}{2*i}\left(e^{i*t} - e^{-i*t}\right)$

   (b) $\cos t = \frac{1}{2}\left(e^{i*t} + e^{-i*t}\right)$

b. Find the Laplace transforms $\mathcal{L}\left(\lambda t.\, \sin t\right)$ and $\mathcal{L}\left(\lambda t.\, \cos t\right)$

8.4.1 Exercises from old exams

Exercise 8.3. *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):

$$\mathcal{L}\,(\lambda t.\, e^{\alpha * t})\,s = 1/(s - \alpha)$$

Exercise 8.4. *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):

$$\mathcal{L}\,(\lambda t.\, e^{\alpha * t})\,s = 1/(s - \alpha)$$

Exercise 8.5. *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 8.6. *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 8.7. *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):

$$\mathcal{L}\,(\lambda t.\, e^{\alpha * t})\,s = 1/(s - \alpha)$$

Work through the three exams from 2016:

2016-Practice: Algebra: Vector space, Typing: derivative chain law, Laplace, Proof: limits
2016-03: Algebra: Lattice, Typing: integration of functional, Laplace, Proof: continuity of (+)
2016-08: Algebra: Abelian group, Typing: conditional probability, Laplace, Proof: continuity of (.)

TODO: Add more exercises

9 End

TODO: sum up and close

Chapter 1: Haskell-intro, Types, Functions, Complex numbers, $eval : syntax \to semantics$

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

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, $Laplace$

A A parameterised type and some complex number operations on it

```
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, ai) *. CS (br, bi) = CS (ar * br − ai * bi, ar * bi + ai * 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
```

References

R. A. Adams and C. Essex. *Calculus: a complete course*. Pearson Canada, 7th edition, 2010.

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. 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.

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.

S. Marlow (ed.). The Haskell 2010 report, 2010. `http://www.haskell.org/onlinereport/haskell2010/`.

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.

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.