Domain Specific Languages of Mathematics: Lecture Notes

Patrik Jansson   Cezar Ionescu

February 13, 2017

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.

1 Week 1

This lecture 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**

1.1 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 1. 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:

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

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:

```haskell
data ComplexA = CPlus₁ ℝ ℝ ImagUnits
              | CPlus₂ ℝ ImagUnits ℝ
```

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

```haskell
showCA :: ComplexA →       String
showCA   (CPlus₁ x y i) = show x ++ " + " ++ show y ++ "i"
showCA   (CPlus₂ x i y) = show x ++ " + " ++ "i" ++ show y
```

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.

```haskell
type ℝ = Double
```

The text continues with examples:

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

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

With this provision, in our notation the examples are written as:

$$\begin{aligned}
&\mathit{testC1} :: [\mathit{ComplexA}] \\
&\mathit{testC1} = [\mathit{CPlus}_1\ 3\ 2\ I,\ \mathit{CPlus}_1\ (7\ /\ 2)\ (-2\ /\ 3)\ I \\
&\qquad\quad ,\ \mathit{CPlus}_2\ 0\ I\ \pi,\ \mathit{CPlus}_1\ (-3)\ 0\ I \\
&\qquad\quad] \\
&\mathit{testS1} = \mathit{map}\ \mathit{showCA}\ \mathit{testC1}
\end{aligned}$$

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:

$$\begin{aligned}
&\mathit{toComplex} :: \mathbb{R} \to \mathit{ComplexA} \\
&\mathit{toComplex}\ x = \mathit{CPlus}_1\ x\ 0\ i
\end{aligned}$$

Again, at this stage there are many open questions. For example, we can assume that *i1* stands for the complex number $\mathit{CPlus}_2\ 0\ i\ 1$, 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 *i1* and therefore $\mathit{CPlus}_2\ 0\ i\ 1$. 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:

$$\mathbf{data}\ \mathit{ComplexB} = \mathit{CPlusB}\ \mathbb{R}\ \mathbb{R}\ \mathit{ImagUnits}$$

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

$$\mathbf{data}\ \mathit{ComplexC} = \mathit{CPlusC}\ \mathbb{R}\ \mathbb{R}$$

(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: *CPlusC* is not just syntactically injective (as all constructors are), but also semantically. The equality on complex numbers is what we would obtain in Haskell by using **deriving** *Eq*.

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

> **type** *ComplexD* = *ComplexSem* $\mathbb{R}$
> **newtype** *ComplexSem* r = *CS* (r, 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*:

> $re :: ComplexSem\ r \ \to r$
> $re\ z@(CS\ (x, y)) \ = x$
>
> $im :: ComplexSem\ r \to r$
> $im\ z@(CS\ (x, y)) = y$

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 *ComplexSem*, for example:

> $(+.) :: Num\ r \Rightarrow ComplexSem\ r \to ComplexSem\ r \to ComplexSem\ r$
> $(CS\ (a, b)) +. (CS\ (x, y)) = CS\ ((a + x), (b + y))$

or we can build a datatype of "syntactic" complex numbers from the algebraic operations to arrive at a *deep embedding* as seen in the next section.

Exercises:

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

1.2 A syntax for 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:

$$\mathit{evalE} :: \mathit{ComplexE} \to \mathit{ComplexD}$$

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:

$$\begin{aligned}
&\textbf{data}\ \mathit{ComplexE} = \mathit{ImagUnit}\\
&\qquad\qquad\qquad \mid \mathit{ToComplex}\ \mathbb{R}\\
&\qquad\qquad\qquad \mid \mathit{Plus}\quad \mathit{ComplexE}\ \mathit{ComplexE}\\
&\qquad\qquad\qquad \mid \mathit{Times}\ \mathit{ComplexE}\ \mathit{ComplexE}\\
&\textbf{deriving}\ (\mathit{Eq}, \mathit{Show})
\end{aligned}$$

And we can write the evaluator by induction over the syntax tree:

$$\begin{aligned}
\mathit{evalE}\ \mathit{ImagUnit} &= \mathit{CS}\ (0,1)\\
\mathit{evalE}\ (\mathit{ToComplex}\ r) &= \mathit{CS}\ (r,0)\\
\mathit{evalE}\ (\mathit{Plus}\ c1\ c2) &= \mathit{evalE}\ c1\ {+.}\ \mathit{evalE}\ c2\\
\mathit{evalE}\ (\mathit{Times}\ c1\ c2) &= \mathit{evalE}\ c1\ {*.}\ \mathit{evalE}\ c2
\end{aligned}$$

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

$$\begin{aligned}
&\mathit{fromCS} :: \mathit{ComplexD} \to \mathit{ComplexE}\\
&\mathit{fromCS}\ (\mathit{CS}\ (x,y)) = \mathit{Plus}\ (\mathit{ToComplex}\ x)\ (\mathit{Times}\ (\mathit{ToComplex}\ y)\ \mathit{ImagUnit})\\
&\mathit{testE1} = \mathit{Plus}\ (\mathit{ToComplex}\ 3)\ (\mathit{Times}\ (\mathit{ToComplex}\ 2)\ \mathit{ImagUnit})\\
&\mathit{testE2} = \mathit{Times}\ \mathit{ImagUnit}\ \mathit{ImagUnit}
\end{aligned}$$

There are certain laws we would like to hold for operations on complex numbers. The simplest is perhaps $i^2 = -1$ from the start of the lecture,

$$\begin{aligned}
&\mathit{propImagUnit} :: \mathit{Bool}\\
&\mathit{propImagUnit} = \mathit{Times}\ \mathit{ImagUnit}\ \mathit{ImagUnit} === \mathit{ToComplex}\ (-1)\\
&(===) :: \mathit{ComplexE} \to \mathit{ComplexE} \to \mathit{Bool}\\
&z === w = \mathit{evalE}\ z == \mathit{evalE}\ w
\end{aligned}$$

and that *fromCS* is an embedding:

$$\begin{aligned}
&\mathit{propFromCS} :: \mathit{ComplexD} \to \mathit{Bool}\\
&\mathit{propFromCS}\ c = \mathit{evalE}\ (\mathit{fromCS}\ c) == c
\end{aligned}$$

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

$$\begin{aligned}
\mathit{propAssocPlus}\ x\ y\ z &= \mathit{Plus}\ (\mathit{Plus}\ x\ y)\ z === \mathit{Plus}\ x\ (\mathit{Plus}\ y\ z)\\
\mathit{propAssocTimes}\ x\ y\ z &= \mathit{Times}\ (\mathit{Times}\ x\ y)\ z === \mathit{Times}\ x\ (\mathit{Times}\ y\ z)\\
\mathit{propDistTimesPlus}\ x\ y\ z &= \mathit{Times}\ x\ (\mathit{Plus}\ y\ z) === \mathit{Plus}\ (\mathit{Times}\ x\ y)\ (\mathit{Times}\ x\ z)
\end{aligned}$$

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:

$$\mathit{propAssocA} :: \mathit{Eq}\ a \Rightarrow (a \to a \to a) \to a \to a \to a \to \mathit{Bool}$$
$$\mathit{propAssocA}\ (+?)\ x\ y\ z = (x +?\ y) +?\ z == x +?\ (y +?\ z)$$

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.

$$\mathit{propAssocAInt} = \mathit{propAssocA}\ (+) :: \mathit{Int} \to \mathit{Int} \to \mathit{Int} \to \mathit{Bool}$$
$$\mathit{propAssocADouble} = \mathit{propAssocA}\ (+) :: \mathit{Double} \to \mathit{Double} \to \mathit{Double} \to \mathit{Bool}$$

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 (fromInteger 0)
evalCSyn :: Num r => ComplexSyn r -> ComplexSem r
evalCSyn (FromCartesian x y) = CS (x, y)
evalCSyn (l :+: r) = evalCSyn l +. evalCSyn r
evalCSyn (l :*: r) = evalCSyn l *. evalCSyn r
instance Num a => Num (ComplexSyn a) where
  (+) = (:+:)
  (*) = (:*:)
  fromInteger = fromIntegerCS
    -- TODO: add a few more operations (hint: extend ComplexSyn as well)
    -- TODO: also extend eval
fromIntegerCS :: Num r => Integer -> ComplexSyn r
fromIntegerCS = toComplexSyn . fromInteger
```

1.3 TODO[PaJa]: Textify

Here are some notes about things scribbled on the blackboard during the first two lectures. At some point this should be made into text for the lecture notes.

1.3.1 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 clear 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.

Scoping Scoping rules for the integral sign:

$$\begin{aligned} f(x) &= x^2 \\ g(x) &= \int_x^{2x} f(x)dx & &= \int_x^{2x} f(y)dy \end{aligned}$$

The variable x bound on the left is independent of the variable x "bound under the integral sign".

From syntax to semantics and back We have seen evaluation functions from abstract syntax to semantics ($\mathit{eval} :: \mathit{Syn} \to \mathit{Sem}$). Often a partial inverse is also available: $\mathit{embed} :: \mathit{Sem} \to \mathit{Syn}$. For our complex numbers we have TODO: fill in a function from $\mathit{ComplexSem}\ r \to \mathit{ComplexSyn}\ r$.

The embedding should satisfy a round-trip property: $\mathit{eval}\ (\mathit{embed}\ s) == s$ for all s. Exercise: What about the opposite direction? When is $\mathit{embed}\ (\mathit{eval}\ e) == e$?

We can also state and check properties relating the semantic and the syntactic operations:

$a + b = \mathit{eval}\ (\mathit{Plus}\ (\mathit{embed}\ a)\ (\mathit{embed}\ b))$ for all a and b.

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 Edwards et al. [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 as to denote a list of as). 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

$$
\begin{aligned}
&\textbf{type } T = \mathit{Real}\\
&\textbf{type } S \ = \mathbb{C}\\
&\mathcal{L} : (T \to \mathbb{C}) \to (S \to \mathbb{C})
\end{aligned}
$$

1.3.2 Other

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.

$$
\begin{aligned}
&\textbf{type } \mathit{CS} = \mathit{ComplexSem} \quad \text{-- for shorter type expressions below}\\
&\mathit{liftPlus} :: (r \quad\ \to r \quad\ \to r \quad\ ) \to\\
&\qquad\qquad\ (\mathit{CS}\ r \to \mathit{CS}\ r \to \mathit{CS}\ r)\\
&\mathit{liftPlus}\ (+)\ (\mathit{CS}\ (x, y))\ (\mathit{CS}\ (x', y')) = \mathit{CS}\ (x + x', y + y')
\end{aligned}
$$

Note that *liftPlus* takes $(+)$ as its first parameter and uses it twice on the RHS.

Laws TODO: Associative, Commutative, Distributive, ...

TODO[PaJa]: move earlier Table of examples of notation and abstract syntax for some complex numbers:

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

1.4 Questions and answers from the exercise sessions week 1

1.4.1 Function composition

The infix operator `.` in Haskell is an implementation of the mathematical operation of function composition.

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

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.)

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

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.

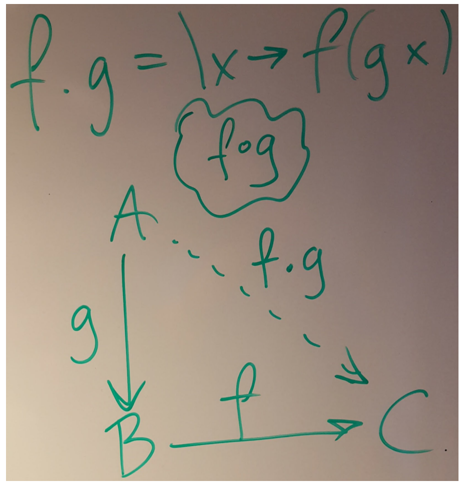

Figure 1: Function composition diagram

1.4.2 fromInteger (looks recursive)

Near the end of the lecture notes there was an instance declaration including the following lines:

> **instance** *Num r* ⇒ *Num* (*ComplexSyn r*) **where**
> -- ... several other methods and then
> *fromInteger* = *toComplexSyn* ∘ *fromInteger*

This definition looks recursive, but it is not. To see why 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:

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

1.4.3 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** *Heltal* = *Integer*
> **type** *Foo* = (*Maybe* [*String*], [[*Heltal*]])
> **type** *BinOp* = *Heltal* → *Heltal* → *Heltal*
> **type** *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

$$\textbf{newtype}\ \mathit{Age} \ = \mathit{Ag}\ \mathit{Int} \quad \text{-- Age in years}$$
$$\textbf{newtype}\ \mathit{Shoe} = \mathit{Sh}\ \mathit{Int} \quad \text{-- Shoe size (EU)}$$

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 $\mathit{Ag} :: \mathit{Int} \to \mathit{Age}$ and $\mathit{Sh} :: \mathit{Int} \to \mathit{Shoe}$ 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.

data – for syntax trees Some examples:

$$\textbf{data}\ N = Z \mid S\ N$$

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: $\mathit{zero} = Z$, $\mathit{one} = S\ Z$, $\mathit{three} = S\ (S\ \mathit{one})$

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

This declaration introduces

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

Example values: $x = V$ `"x"`, $\mathit{e1} = P\ x\ x$, $\mathit{e2} = T\ \mathit{e1}\ \mathit{e1}$

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

$$\textbf{data}\ E' = V'\ \mathit{String} \mid E' :\!+ E' \mid E' :\!* E'$$

Example values: $y = V$ `"y"`, $\mathit{e1} = y :\!+ y$, $\mathit{e2} = x :\!* \mathit{e1}$

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

$$\begin{aligned}\textbf{data}\ \mathit{ComplexSy}\ v\ r = {} & \mathit{Var}\ v \\ \mid\ & \mathit{FromCart}\ r\ r \\ \mid\ & \mathit{ComplexSy}\ v\ r :\!{+}\!{+}\ \mathit{ComplexSy}\ v\ r \\ \mid\ & \mathit{ComplexSy}\ v\ r :\!{*}{*}\ \mathit{ComplexSy}\ v\ r\end{aligned}$$

The purpose of the first parameter v here is to enable a free choice of type for the variables (be it *String* or *Int* or something else) and the second parameter r makes is possible to express "complex numbers over" different base types (like *Double*, *Float*, *Integer*, etc.).

1.4.4 *Env*, *Var*, and variable lookup

The type synonym

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

is one way of expressing a partial function from v to s.

Example value:

$$\begin{aligned}
&\mathit{env1} :: \mathit{Env}\ \mathit{String}\ \mathit{Int}\\
&\mathit{env1} = [(\texttt{"hej"}, 17), (\texttt{"du"}, 38)]
\end{aligned}$$

The *Env* type is commonly used in evaluator functions for syntax trees containing variables:

$$\begin{aligned}
&\mathit{evalCP} :: \mathit{Eq}\ v \Rightarrow \mathit{Env}\ v\ (\mathit{ComplexSem}\ r) \to (\mathit{ComplexSy}\ v\ r \to \mathit{ComplexSem}\ r)\\
&\mathit{evalCP}\ \mathit{env}\ (\mathit{Var}\ x) = \textbf{case}\ \mathit{lookup}\ x\ \mathit{env}\ \textbf{of}\\
&\quad \mathit{Just}\ c \to \mathit{undefined} \quad \text{-- ...}\\
&\quad \text{-- ...}
\end{aligned}$$

Notice that *env* maps "syntax" (variable names) to "semantics", just like the evaluator does.

1.5 Some helper functions

$$\begin{aligned}
&\mathit{propAssocAdd} :: (\mathit{Eq}\ a, \mathit{Num}\ a) \Rightarrow a \to a \to a \to \mathit{Bool}\\
&\mathit{propAssocAdd} = \mathit{propAssocA}\ (+)\\
&(*.) :: \mathit{Num}\ r \Rightarrow \mathit{ComplexSem}\ r \to \mathit{ComplexSem}\ r \to \mathit{ComplexSem}\ r\\
&\mathit{CS}\ (ar, ai) *. \mathit{CS}\ (br, bi) = \mathit{CS}\ (ar * br - ai * bi, ar * bi + ai * br)\\
&\textbf{instance}\ \mathit{Show}\ r \Rightarrow \mathit{Show}\ (\mathit{ComplexSem}\ r)\ \textbf{where}\\
&\quad \mathit{show} = \mathit{showCS}\\
&\mathit{showCS} :: \mathit{Show}\ r \Rightarrow \mathit{ComplexSem}\ r \to \mathit{String}\\
&\mathit{showCS}\ (\mathit{CS}\ (x, y)) = \mathit{show}\ x +\!\!+ \texttt{" + "} +\!\!+ \mathit{show}\ y +\!\!+ \texttt{"i"}
\end{aligned}$$

2 Week 2

Course learning outcomes:

- 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

This week we focus on "develop adequate notation for mathematical concepts" and "perform calculational proofs" (still in the context of "organize areas of mathematics in DSL terms").

module *DSLsofMath.W02* **where**

2.1 A few words about 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. The core of the language of pure set theory has the Empty set, the one-element set constructor Singleton, set Union, and Intersection. There are no "atoms" or "elements" to start from except for the empty set but it turns out that quite a large part of mathematics can still be expressed.

Natural numbers To talk about things like natural numbers in pure set theory they need to be encoded. Here is one such encoding (which is explored further in the first hand-in assignment).

$$\begin{aligned}
\mathit{vonNeumann}\ 0 &= \mathit{Empty} \\
\mathit{vonNeumann}\ (n+1) &= \mathit{Union}\ (\mathit{vonNeumann}\ n) \\
&\qquad\quad (\mathit{Singleton}\ (\mathit{vonNeumann}\ n))
\end{aligned}$$

Pairs Definition: A pair (a, b) is encoded as $\{\{a\}, \{a, b\}\}$.

2.2 Propositional Calculus

Now we turn to the main topic of this week: logic and proofs.

TODO: type up the notes + whiteboard photos

Swedish: Satslogik

False, True, And, Or, Implies

2.3 First Order Logic (predicate logic)

TODO: type up the notes + whiteboard photos

Swedish: Första ordningens logik = predikatlogik

Adds term variables and functions, predicate symbols and quantifiers (sv: kvantorer).

2.4 Basic concepts of calculus

Limit point TODO: transcribe the 2016 notes + 2017 black board pictures into notes.

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 if for every $\epsilon > 0$, there exists $q \in X$ such that $q \neq p$ and $|q - p| < \epsilon$.

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

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

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

Next: introduce the "disk function" Di.

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

Then we get

$$\mathit{Limp}\ p\ X = \exists q : Q.\ \forall\, \epsilon > 0.\ q\ \epsilon \in Di\ p\ \epsilon$$

Example: limit outside the set X

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

Show that 0 is a limit point. Note that $0 \notin X$.

We want to prove $\mathit{Limp}\ 0\ X$

$$q\ \epsilon = 1 \mathbin{/} n\ \textbf{where}\ n = \mathit{ceiling}\ (1 \mathbin{/} \epsilon)$$

(where the definition of n comes from a calculation showing the property involving Di is satisfied.)

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

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

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

Good excercise in quantifier negation!

$$f : (q : Q) \to \mathbb{R}_{>0}\ \text{\{-such that } \textbf{let}\ \epsilon = f\ q\ \textbf{in}\ q\ \epsilon \notin Di\ p\ \epsilon\ \text{-\}}$$

Note that $q\ \epsilon$ is in (TODO: To be cont.)

The limit of a sequence TODO: transcribe the 2016 notes + 2017 black board pictures into notes.

$$P\ a\ \epsilon\ L = (\epsilon > 0) \to \exists N : \mathbb{Z}.\ (\forall n : \mathbb{N}.\ (n \geqslant N) \to (\mid a_n - L \mid < \epsilon))$$

2.5 Questions and answers from the exercise sessions week 2

Variables, *Env* and *lookup* This was a frequently 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.

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

We have a type $\mathit{Rat}\ v$ for the syntax trees of rational number expressions and a type RatSem for the semantics of those rational number expressions as pairs of integers. The constructor $\mathit{RV} :: v \to \mathit{Rat}\ v$ is used to embed variables with names of type v in $\mathit{Rat}\ v$. 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. To evaluate some $e :: \mathit{Rat}\ v$ 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 $v \to \mathit{RatSem}$) so we can give the following implementation of the evaluator:

$$\begin{aligned}
&\mathit{evalRat1} :: (v \to \mathit{RatSem}) \to (\mathit{Rat}\ v \to \mathit{RatSem})\\
&\mathit{evalRat1}\ \mathit{ev}\ (\mathit{RV}\ v) \quad\ \ = \mathit{ev}\ v\\
&\mathit{evalRat1}\ \mathit{ev}\ (\mathit{FromI}\ i) \ \ = \mathit{fromISem}\ i\\
&\mathit{evalRat1}\ \mathit{ev}\ (\mathit{RPlus}\ l\ r) = \mathit{plusSem}\ (\mathit{evalRat1}\ \mathit{ev}\ l)\ (\mathit{evalRat1}\ \mathit{ev}\ r)\\
&\mathit{evalRat1}\ \mathit{ev}\ (\mathit{RDiv}\ l\ r) \ = \mathit{divSem}\ \ (\mathit{evalRat1}\ \mathit{ev}\ l)\ (\mathit{evalRat1}\ \mathit{ev}\ r)
\end{aligned}$$

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 $\mathit{plusSem}$, etc.

$$\begin{aligned}
&\mathit{fromISem} :: \mathit{Integer} \to \mathit{RatSem}\\
&\mathit{fromISem}\ i = \mathit{RSem}\ (i, 1)\\
&\mathit{plusSem} :: \mathit{RatSem} \to \mathit{RatSem} \to \mathit{RatSem}\\
&\mathit{plusSem} = \mathit{undefined} \quad \text{-- TODO: exercise}\\
&\quad \text{-- Division of rational numbers}\\
&\mathit{divSem} :: \mathit{RatSem} \to \mathit{RatSem} \to \mathit{RatSem}\\
&\mathit{divSem}\ (\mathit{RSem}\ (a, b))\ (\mathit{RSem}\ (c, d)) = \mathit{RSem}\ (a * d, b * c)
\end{aligned}$$

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

$$\begin{aligned}
&\textbf{type}\ \mathit{Env}\ v\ s = [(v, s)]\\
&\mathit{envToFun} :: (\mathit{Show}\ v, \mathit{Eq}\ v) \Rightarrow \mathit{Env}\ v\ s \to (v \to s)\\
&\mathit{envToFun}\ [\,]\ v = \mathit{error}\ (\texttt{"envToFun: variable "} + \!\! + \mathit{show}\ v + \!\! + \texttt{" not found"})\\
&\mathit{envToFun}\ ((w, s) : \mathit{env})\ v\\
&\quad \mid w == v \qquad = s\\
&\quad \mid \mathit{otherwise} = \mathit{envToFun}\ \mathit{env}\ v
\end{aligned}$$

Thus, $\mathit{Env}\ v\ s$ 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 $\mathit{envToFun}$ in our hands we can implement a second version of the evaluator:

$$\begin{aligned}
&\mathit{evalRat2} :: (\mathit{Show}\ v, \mathit{Eq}\ v) \Rightarrow (\mathit{Env}\ v\ \mathit{RatSem}) \to (\mathit{Rat}\ v \to \mathit{RatSem})\\
&\mathit{evalRat2}\ \mathit{env}\ e = \mathit{evalRat1}\ (\mathit{envToFun}\ \mathit{env})\ e
\end{aligned}$$

The law of the excluded middle Many had problems with implementing the "law of the excluded middle" in the exercises and it is indeed a tricky property to prove. They key to implementing it lies in double negation and as that is encoded with higher order functions it gets a bit hairy.

TODO[Daniel]: more explanation

SET and PRED Several groups have had trouble grasping the difference between *SET* and *PRED*. This is understandable, beacuse 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:

```
type Term v = Rat v
data RPred v = Equal    (Term v) (Term v)
             | LessThan (Term v) (Term v)
             | Positive (Term v)
             | And (RPred v) (RPred v)
             | Not (RPred v)
  deriving (Eq, Show)
```

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.

```
checkRP :: (Eq v, Show v) ⇒ Env v RatSem → RPred v → Bool
checkRP env (Equal    t1 t2) = eqSem       (evalRat2 env t1) (evalRat2 env t2)
checkRP env (LessThan t1 t2) = lessThanSem (evalRat2 env t1) (evalRat2 env t2)
checkRP env (Positive t1)    = positiveSem (evalRat2 env t1)
checkRP env (And p q)  = (checkRP env p) ∧ (checkRP env q)
checkRP env (Not p)    = ¬ (checkRP env p)
```

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

```
eqSem       :: RatSem → RatSem → Bool
lessThanSem :: RatSem → RatSem → Bool
positiveSem :: RatSem → Bool
eqSem       = error "TODO"
lessThanSem = error "TODO"
positiveSem = error "TODO"
```

2.6 More general code for first order languages

"överkurs"

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

TODO: add explanatory text

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

```haskell
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
```

3 Week 3

```haskell
{-# LANGUAGE FlexibleInstances #-}
module DSLsofMath.W03 where
```

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 $D\ f$
- $\partial f/\partial x_i : (\mathbb{R}^{\mathrm{n}} \to \mathbb{R}) \to \mathbb{R}^{\mathrm{n}} \to \mathbb{R}$
- 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: partial derivative

As as an example we will try to type the elements of a mathematical definition.

For example, on page 169 of Mac Lane [1986], we read

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

$$\partial z/\partial x = f'_x(x, y) = \lim_{h \to 0}(f(x + h, y) - f(x, y))/h$$

What are the types of the elements involved? We have

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

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

The expression (x, y) has several occurrences. The first two denote variables of type U, the third is just a name ((x, y)-plane). The third denotes a variable of type U, it is bound by a universal quantifier

$$\forall\ (x, y) \in U$$

The variable h appears to be a non-zero real number, bound by a universal quantifier, but that is incorrect. In fact, h is used as a variable to construct the arguments of a function, whose limit is then taken at 0.

That function, which we can denote by 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:

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

$z = f\ (x, y)$ probably does not mean that $z \in \mathbb{R}$, although the phrase "the quantity z" 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). There is the added difficulty that, just like $_x$, the x in ∂x is not the x bound by the universal quantifier, but just a symbol.

3.3 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:

1. $\partial L / \partial q$ suggests that L is a function of at least a pair of arguments:

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

   This is consistent with the description: "Lagrangian function of the system state (time, coordinates, and velocities)". So we can take $n = 3$:

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

2. $\partial L / \partial q$ 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

   $$L\ (t, q, v) = \ldots$$

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

   $$\partial\, L\ /\ \partial q : \mathbb{R}^3 \to \mathbb{R}$$

   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 0:

   $$const\ 0 : \mathbb{R}^3 \to \mathbb{R}$$
   $$const\ 0\ (t, q, v) = 0$$

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

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{q}} : \mathbb{R} \rightarrow \mathbb{R}$$

Since we subtract from this the function $\partial L/\partial q$, it follows that this, too, must be of type $\mathbb{R} \rightarrow \mathbb{R}$, contradiction.

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

6. Looking back at the description above, we see that the only 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

$$w : \mathbb{R} \rightarrow \mathbb{R}, \text{where } w(t) \text{ is a coordinate at time } t$$

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 be a function

$$w : \mathbb{R} \rightarrow \mathbb{R}^{\mathrm{n}}$$

which is equivalent to n functions of type $\mathbb{R} \rightarrow \mathbb{R}$. We would then have an equation for each of them. We will use $n = 1$ for the rest of this example.

7. The Lagrangian is a "function of the system state (time, coordinates, and velocities)". If we have a path, then the coordinates at any time are given by the path. The velocity is the derivative of the path, also fixed by the path:

$$\begin{aligned}
&q : \mathbb{R} \rightarrow \mathbb{R}\\
&q\ t = w\ t\\
&\dot{q} : \mathbb{R} \rightarrow \mathbb{R}\\
&\dot{q}\ t = dw\ /\ dt
\end{aligned}$$

The equations do not use a function $L : \mathbb{R}^3 \rightarrow \mathbb{R}$, but rather

$$L \circ \mathit{expand}\ w : \mathbb{R} \rightarrow \mathbb{R}$$

where the "combinator" *expand* is given by

$$\begin{aligned}
&\mathit{expand} &&: (\mathbb{R} \rightarrow \mathbb{R}) \rightarrow \mathbb{R} \rightarrow \mathbb{R}^3\\
&\mathit{expand}\ w\ t &&= (t, w\ t, D\ w\ t)
\end{aligned}$$

8. Similarly, using D_1, D_2, D_3 instead of $\partial L\ /\partial\ t$ etc., we have that, instead of $\partial L\ /\partial\ q$ what is meant is

$$D_2\ L \circ \mathit{expand}\ w : \mathbb{R} \rightarrow \mathbb{R}$$

and instead of $\partial L\ /\partial\ \dot{q}$

$$D_3\ L \circ \mathit{expand}\ w : \mathbb{R} \rightarrow \mathbb{R}$$

The equation becomes

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

a relation between functions of type $\mathbb{R} \rightarrow \mathbb{R}$. In particular, the right-hand 0 is the constant function

$$\mathit{const}\ 0 : \mathbb{R} \rightarrow \mathbb{R}$$

3.4 Types in Mathematics (Part II)

3.4.1 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*.

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

TODO: insert proper citation **?**

This is taken from the Haskell documentation[1] but it appears that *Eq* and *Show* are not necessary, because there are meaningful instances of *Num* which don't support them:

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

Next we have *Fractional* for when we also have division:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Num}\ a \Rightarrow \mathit{Fractional}\ a\ \textbf{where} \\
\quad \begin{array}{ll}
(/) & :: a \to a \to a
\end{array}
\end{array}
$$

[1] Fig. 6.2 in section 6.4 of the Haskell 2010 report: `https://www.haskell.org/onlinereport/haskell2010/haskellch6.html`.

$$
\begin{array}{l}
\quad \mathit{recip} :: a \to a \\
\quad \mathit{fromRational} :: \mathit{Rational} \to a
\end{array}
$$

and *Floating* when we can implement the “standard” funtions from calculus:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{class } \mathit{Fractional}\ a \Rightarrow \mathit{Floating}\ a \textbf{ where}} \\
\quad \pi & :: a \\
\quad \mathit{exp}, \mathit{log}, \sqrt{\cdot} & :: a \to a \\
\quad (**), \mathit{logBase} & :: a \to a \to a \\
\quad \mathit{sin}, \mathit{cos}, \mathit{tan} & :: a \to a \\
\quad \mathit{asin}, \mathit{acos}, \mathit{atan} & :: a \to a \\
\quad \mathit{sinh}, \mathit{cosh}, \mathit{tanh} & :: a \to a \\
\quad \mathit{asinh}, \mathit{acosh}, \mathit{atanh} & :: a \to a
\end{array}
$$

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

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{instance } \mathit{Fractional}\ a \Rightarrow \mathit{Fractional}\ (x \to a) \textbf{ where}} \\
\quad \mathit{recip}\ f & = \mathit{recip} \circ f \\
\quad \mathit{fromRational} & = \mathit{const} \circ \mathit{fromRational}
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{instance } \mathit{Floating}\ a \Rightarrow \mathit{Floating}\ (x \to a) \textbf{ where}} \\
\quad \pi & = \mathit{const}\ \pi \\
\quad \mathit{exp}\ f & = \mathit{exp} \circ f \\
\quad f ** g & = \lambda x \to (f\ x) ** (g\ x) \\
\quad\quad \text{-- and so on} &
\end{array}
$$

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.

3.5 Computing derivatives

The “little language” of derivatives:

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

and so on.

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 DSL of expressions (representing functions in one variable):

$$
\begin{array}{ll}
\mathit{expression} & ::= \mathit{const}\ \mathbb{R} \\
 & \mid\ \ \mathit{id}
\end{array}
$$

$$
\begin{array}{ll}
| & \mathit{expression} + \mathit{expression} \\
| & \mathit{expression} * \mathit{expression} \\
| & \mathit{exp}\ \mathit{expression} \\
| & \ldots
\end{array}
$$

etc.

We can implement this in a datatype:

$$
\begin{array}{l}
\textbf{data}\ \mathit{FunExp} = \mathit{Const}\ \mathit{Double} \\
\qquad\qquad\quad |\ \ \mathit{Id} \\
\qquad\qquad\quad |\ \ \mathit{FunExp}\ {:}{+}{:}\ \mathit{FunExp} \\
\qquad\qquad\quad |\ \ \mathit{FunExp}\ {:}{*}{:}\ \mathit{FunExp} \\
\qquad\qquad\quad |\ \ \mathit{Exp}\ \mathit{FunExp} \\
\qquad\qquad\quad\ \ \text{-- and so on} \\
\qquad\qquad\quad \textbf{deriving}\ \mathit{Show}
\end{array}
$$

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

$$
\begin{array}{llll}
\mathit{eval} :: \mathit{FunExp} & \rightarrow \mathit{Double} \rightarrow \mathit{Double} & \\
\mathit{eval}\ \ (\mathit{Const}\ \mathit{alpha}) & = \mathit{const}\ \mathit{alpha} & \\
\mathit{eval}\ \ \mathit{Id} & = \mathit{id} & \\
\mathit{eval}\ \ (\mathit{e1}\ {:}{+}{:}\ \mathit{e2}) & = \mathit{eval}\ \mathit{e1} + \mathit{eval}\ \mathit{e2} & \text{-- note the use of ``lifted +''} \\
\mathit{eval}\ \ (\mathit{e1}\ {:}{*}{:}\ \mathit{e2}) & = \mathit{eval}\ \mathit{e1} * \mathit{eval}\ \mathit{e2} & \text{-- ``lifted *''} \\
\mathit{eval}\ \ (\mathit{Exp}\ \mathit{e1}) & = \mathit{exp}\ (\mathit{eval}\ \mathit{e1}) & \text{-- and ``lifted } \mathit{exp}\text{''} \\
\quad \text{-- and so on} & &
\end{array}
$$

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

$$
\begin{tikzcd}
\mathit{FunExp} \arrow[r, "\mathit{eval}"] \arrow[d, "\mathit{derive}"] & \mathit{Func} \arrow[d, "D"] \\
\mathit{FunExp} \arrow[r, "\mathit{eval}"] & \mathit{Func}
\end{tikzcd}
$$

In other words, for any expression e, we want

$$\mathit{eval}\ (\mathit{derive}\ e) = D\ (\mathit{eval}\ e)$$

For example, let us derive the *derive* function for $\mathit{Exp}\ e$:

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

$$
\begin{aligned}
&= \{\text{-specification of } \textit{derive} \text{ -}\}\\
&\quad (\textit{exp} \circ \textit{eval } e) * (\textit{eval } (\textit{derive } e))\\
&= \{\text{-def. of } \textit{eval} \text{ for } \textit{Exp} \text{ -}\}\\
&\quad (\textit{eval } (\textit{Exp } e)) * (\textit{eval } (\textit{derive } e))\\
&= \{\text{-def. of } \textit{eval} \text{ for } {:}{*}{:} \text{ -}\}\\
&\quad \textit{eval } (\textit{Exp } e \mathbin{{:}{*}{:}} \textit{derive } e)
\end{aligned}
$$

Therefore, the specification is fulfilled by taking

$$\textit{derive } (\textit{Exp } e) = \textit{Exp } e \mathbin{{:}{*}{:}} \textit{derive } e$$

Similarly, we obtain

$$
\begin{aligned}
&\textit{derive } (\textit{Const alpha}) &&= \textit{Const } 0\\
&\textit{derive Id} &&= \textit{Const } 1\\
&\textit{derive } (\textit{e1} \mathbin{{:}{+}{:}} \textit{e2}) &&= \textit{derive e1} \mathbin{{:}{+}{:}} \textit{derive e2}\\
&\textit{derive } (\textit{e1} \mathbin{{:}{*}{:}} \textit{e2}) &&= (\textit{derive e1} \mathbin{{:}{*}{:}} \textit{e2}) \mathbin{{:}{+}{:}} (\textit{e1} \mathbin{{:}{*}{:}} \textit{derive e2})\\
&\textit{derive } (\textit{Exp } e) &&= \textit{Exp } e \mathbin{{:}{*}{:}} \textit{derive } e
\end{aligned}
$$

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

3.6 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

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

For example:

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

$$
\begin{aligned}
& \quad \mathit{eval}\ (\mathit{Exp}\ e) \mathbin{:*:} \mathit{eval}\ (\mathit{derive}\ e) \\
= & \quad \{\text{-def. } \mathit{eval} \text{ for } \mathit{Exp} \text{ -}\} \\
& \quad \mathit{exp}\ (\mathit{eval}\ e) * \mathit{eval}\ (\mathit{derive}\ e) \\
= & \quad \{\text{-def. } \mathit{eval}' \text{ -}\} \\
& \quad \mathit{exp}\ (\mathit{eval}\ e) * \mathit{eval}'\ e
\end{aligned}
$$

and the first e doesn't go away. 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:

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

Is *evalD* compositional?

We compute, for example:

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

This semantics *is* compositional. We can now define a shallow embedding for the computation of derivatives, using the numerical type classes.

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

Exercise: implement the rest

4 Week 4

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

4.1 A simpler example of a non-compositional function

Consider a very simple datatype of integer expressions:

$$
\begin{aligned}
&\mathbf{data}\ E = \mathit{Add}\ E\ E \mid \mathit{Mul}\ E\ E \mid \mathit{Con}\ \mathit{Integer}\ \mathbf{deriving}\ \mathit{Eq}\\
&\mathit{e1}, \mathit{e2} :: E\\
&\mathit{e1} = \mathit{Add}\ (\mathit{Con}\ 1)\ (\mathit{Mul}\ (\mathit{Con}\ 2)\ (\mathit{Con}\ 3))\\
&\mathit{e2} = \mathit{Mul}\ (\mathit{Add}\ (\mathit{Con}\ 1)\ (\mathit{Con}\ 2))\ (\mathit{Con}\ 3)
\end{aligned}
$$

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"`.

$$\mathit{pretty} :: E \to \mathit{String}$$

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 syntact constructor:

$$
\begin{aligned}
&\mathit{pretty}\ (\mathit{Add}\ x\ y) = \mathit{prettyAdd}\ (\mathit{pretty}\ x)\ (\mathit{pretty}\ y)\\
&\mathit{pretty}\ (\mathit{Mul}\ x\ y) = \mathit{prettyMul}\ (\mathit{pretty}\ x)\ (\mathit{pretty}\ y)\\
&\mathit{pretty}\ (\mathit{Con}\ c) \quad\ = \mathit{prettyCon}\ c\\
&\mathit{prettyAdd} :: \mathit{String} \to \mathit{String} \to \mathit{String}\\
&\mathit{prettyMul} :: \mathit{String} \to \mathit{String} \to \mathit{String}\\
&\mathit{prettyCon} :: \mathit{Integer} \to \mathit{String}
\end{aligned}
$$

Now, if we try to implement the semantic constructors without thinking too much we would get the following:

$$
\begin{aligned}
&\mathit{prettyAdd}\ \mathit{sx}\ \mathit{sy} = \mathit{sx} +\!\!+ \texttt{"+"} +\!\!+ \mathit{sy}\\
&\mathit{prettyMul}\ \mathit{sx}\ \mathit{sy} = \mathit{sx} +\!\!+ \texttt{"*"} +\!\!+ \mathit{sy}\\
&\mathit{prettyCon}\ i \qquad = \mathit{show}\ i\\
&\mathit{p1}, \mathit{p2} :: \mathit{String}\\
&\mathit{p1} = \mathit{pretty}\ \mathit{e1}\\
&\mathit{p2} = \mathit{pretty}\ \mathit{e2}\\
&\mathit{trouble} :: \mathit{Bool}\\
&\mathit{trouble} = \mathit{p1} == \mathit{p2}
\end{aligned}
$$

Note that both *e1* and *e2* 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:

$$
\begin{aligned}
&\mathit{prTop} :: E \to \mathit{String}\\
&\mathit{prTop}\ e = \mathbf{let}\ (\mathit{pTop}, \_, \_) = \mathit{prVersions}\ e\\
&\qquad\qquad\ \mathbf{in}\ \mathit{pTop}\\
&\mathit{prVersions} = \mathit{foldE}\ \mathit{prVerAdd}\ \mathit{prVerMul}\ \mathit{prVerCon}\\
&\mathit{prVerAdd}\ (\mathit{xTop}, \mathit{xInA}, \mathit{xInM})\ (\mathit{yTop}, \mathit{yInA}, \mathit{yInM}) =\\
&\quad \mathbf{let}\ s = \mathit{xInA} +\!\!+ \texttt{"+"} +\!\!+ \mathit{yInA} \qquad \text{-- use } \mathit{InA} \text{ because we are ``in } \mathit{Add}\text{''}\\
&\quad \mathbf{in}\ (s, \mathit{paren}\ s, \mathit{paren}\ s) \qquad\quad\ \text{-- parens needed except at top level}\\
&\mathit{prVerMul}\ (\mathit{xTop}, \mathit{xInA}, \mathit{xInM})\ (\mathit{yTop}, \mathit{yInA}, \mathit{yInM}) =\\
&\quad \mathbf{let}\ s = \mathit{xInM} +\!\!+ \texttt{"*"} +\!\!+ \mathit{yInM} \quad\ \text{-- use } \mathit{InM} \text{ because we are ``in } \mathit{Mul}\text{''}\\
&\quad \mathbf{in}\ (s, s, \mathit{paren}\ s) \qquad\qquad\quad\ \ \text{-- parens only needed inside } \mathit{Mul}
\end{aligned}
$$

```
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.

4.2 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*.

TODO: Picture to illustrate

```
    Add                              add
   /   \                            /   \
Con 1   Mul         |----->     con 1   mul
       /   \                           /   \
    Con 2  Con 3                    con 2  con 3
```

As an example we can define a general *foldE* for the integer expressions:

```
foldE :: (t → t → t) → (t → t → t) → (Integer → t) →
         E → t
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**
  $add :: t \to t \to t$
  $mul :: t \to t \to t$
  $con :: Integer \to t$

In this way we can make "hide" the arguments to the fold:

$foldIE :: IntExp\ t \Rightarrow E \to t$
$foldIE = foldE\ add\ mul\ con$
instance $IntExp\ E$ **where**
  $add = Add$
  $mul = Mul$
  $con = Con$
instance $IntExp\ Integer$ **where**
  $add = (+)$
  $mul = (*)$
  $con = id$
$idE' :: E \to E$
$idE' = foldIE$
$evalE' :: E \to Integer$
$evalE' = foldIE$

4.3 Back to derivatives and evaluation

Review section 3.6 again with the definition of $eval'$ being non-compositional (just like *pretty*) and *evalD* a more complex, but compositional, semantics.

5 Algebraic Structures and DSLs

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

5.1 Algebras, homomorphisms

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**
> $unit :: a$
> $op\ \ :: a \to a \to a$
>
> 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

$$\forall\, x : a.\ (\mathit{unit}\ \text{`}\mathit{op}\text{`}\ x == x \wedge x\ \text{`}\mathit{op}\text{`}\ \mathit{unit} == x)$$
$$\forall\, x, y, z : a.\ (x\ \text{`}\mathit{op}\text{`}\ (y\ \text{`}\mathit{op}\text{`}\ z) == (x\ \text{`}\mathit{op}\text{`}\ y)\ \text{`}\mathit{op}\text{`}\ z)$$

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), $h\ (fA\ (x1, ..., x_n)) = fB\ (h\ (x1), ..., h\ (x_n))$.

Example: Monoid homomorphism

$$h\ \mathit{unit} = \mathit{unit}$$
$$h\ (x\ \text{`}\mathit{op}\text{`}\ y) = h\ x\ \text{`}\mathit{op}\text{`}\ h\ y$$

```
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 $h : \mathit{ANat} \to \mathit{MNat}$ be a homomorphism. Then

$$h\ 0 = 1$$
$$h\ (x + y) = h\ x * h\ y$$

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.

5.2 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

$$\begin{aligned} &eval\ (e1 :*: e2) = eval\ e1 * eval\ e2 \\ &eval\ (Exp\ e) \quad\ = exp\ (eval\ e) \end{aligned}$$

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:

$$eval :: GoodClass\ a \Rightarrow FunExp \to a$$

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

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

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

$$\begin{aligned} &\mathbf{type}\ Var \quad\ = String \\ &\mathbf{data}\ MExpr = Unit \mid Op\ MExpr\ MExpr \mid V\ Var \end{aligned}$$

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:

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

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

$$\begin{aligned} &evalM\ f\ Unit \qquad\ \ = unit \\ &evalM\ f\ (Op\ e1\ e2) = op\ (evalM\ f\ e1)\ (evalM\ f\ e2) \\ &evalM\ f\ (V\ x) \qquad = f\ x \end{aligned}$$

(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.

5.3 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:

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

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

$$\begin{array}{l}\mathit{apply} :: a \rightarrow (a \rightarrow b) \rightarrow b \\ \mathit{apply}\ a = \lambda f \rightarrow f\ a\end{array}$$

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

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

etc.

Can we do something similar for *FD*?

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

$$\begin{array}{l}\mathit{apply} :: a \rightarrow \mathit{FD}\ a\ \rightarrow (a, a) \\ \mathit{apply}\quad c \quad\ (f, f') = (f\ c, f'\ c)\end{array}$$

We now have the domain of the homomorphism ($\mathit{FD}\ a$) and the homomorphism itself ($\mathit{apply}\ c$), 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:

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

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.

6 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"
```

7 Week 5: Polynomials and Power Series

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

7.1 Preliminaries

Last time, 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.

7.2 Polynomials

From Adams and Essex [2010], page 55:

> A **polynomial** is a function P whose value at x is
>
> $$Px = a_n x^n + a_{n-1} x^{n-1} + ... a_1 x + a_0$$
>
> where a_n, a_{n-1}, $\ldots$, a_1, and a_0, called the **coefficients** of the polymonial [original spelling], 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, ... $a_n \in \mathbb{R}$ 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, ..., $a_n \in \mathbb{R}$ with $a_n \neq 0$ such that, for any $x \in \mathbb{R}$
>
> $$Px = a_n x^n + a_{n-1} x^{n-1} + ... a_1 x + a_0$$

Obviously, given the coefficients a_i we can evaluate P at any given x. Assuming the coefficients are given as

$as = [a0, a1, ..., a_n]$

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

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

Not every list 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 $\mathit{Poly}\ a$ and $[a]$ is given by the following functions:

$$\begin{array}{l}
\mathit{toList} :: \mathit{Poly}\ a \to [a] \\
\mathit{toList}\ (\mathit{Single}\ a) \quad = a : [] \\
\mathit{toList}\ (\mathit{Cons}\ a\ \mathit{as}) = a : \mathit{toList}\ \mathit{as} \\
\mathit{fromList} :: [a] \to \mathit{Poly}\ a \\
\mathit{fromList}\ (a : []) \qquad = \mathit{Single}\ a \\
\mathit{fromList}\ (a0 : a1 : \mathit{as}) = \mathit{Cons}\ a0\ (\mathit{fromList}\ (a1 : \mathit{as})) \\
\textbf{instance}\ \mathit{Show}\ a \Rightarrow \mathit{Show}\ (\mathit{Poly}\ a)\ \textbf{where} \\
\quad \mathit{show} = \mathit{show} \circ \mathit{toList}
\end{array}$$

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

$$\begin{array}{l}
\mathit{evalPoly} :: \mathit{Num}\ a \Rightarrow \mathit{Poly}\ a \to a \to a \\
\mathit{evalPoly}\ (\mathit{Single}\ a) \quad x = a \\
\mathit{evalPoly}\ (\mathit{Cons}\ a\ \mathit{as})\ x = a + x * \mathit{evalPoly}\ \mathit{as}\ x
\end{array}$$

Since we have $\mathit{Num}\ a$, there is a Num structure on $a \to a$, and $\mathit{evalPoly}$ looks like a homomorphism. Question: is there a Num structure on $\mathit{Poly}\ a$, such that $\mathit{evalPoly}$ is a homomorphism?

For example, the homomorphism condition gives for $(+)$

$$\mathit{evalPoly}\ \mathit{as} + \mathit{evalPoly}\ \mathit{bs} = \mathit{evalPoly}\ (\mathit{as} + \mathit{bs})$$

Both sides are functions, they are equal iff they are equal for every argument. For an arbitrary x

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

To proceed further, we need to consider the various cases in the definition of $\mathit{evalPoly}$. We give here the computation for the last case (where as has at least one Cons), using the traditional list notation (:) for brevity.

$$\mathit{evalPoly}\ (a : \mathit{as})\ x + \mathit{evalPoly}\ (b : \mathit{bs})\ x = \mathit{evalPoly}\ ((a : \mathit{as}) + (b : \mathit{bs}))\ x$$

For the left-hand side, we have:

$$\begin{array}{ll}
 & \mathit{evalPoly}\ (a : \mathit{as})\ x + \mathit{evalPoly}\ (b : \mathit{bs})\ x \\
= & \{\text{-def. } \mathit{evalPoly}\ \text{-}\} \\
 & (a + x * \mathit{evalPoly}\ \mathit{as}\ x) + (b + x * \mathit{eval}\ \mathit{bs}\ x) \\
= & \{\text{-properties of +, valid in any ring -}\} \\
 & (a + b) + x * (\mathit{evalPoly}\ \mathit{as}\ x + \mathit{evalPoly}\ \mathit{bs}\ x) \\
= & \{\text{-homomorphism condition -}\} \\
 & (a + b) + x * (\mathit{evalPoly}\ (\mathit{as} + \mathit{bs})\ x) \\
= & \{\text{-def. } \mathit{evalPoly}\ \text{-}\} \\
 & \mathit{evalPoly}\ ((a + b) : (\mathit{as} + \mathit{bs}))\ x
\end{array}$$

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.

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Num}\ a \Rightarrow \mathit{Num}\ (\mathit{Poly}\ a)\ \textbf{where} \\
\quad (+) = \mathit{polyAdd} \\
\quad (*) = \mathit{polyMul} \\
\quad \mathit{negate} = \mathit{polyNeg} \\
\quad \mathit{fromInteger} = \mathit{Single} \circ \mathit{fromInteger} \\
\mathit{polyAdd} :: \mathit{Num}\ a \Rightarrow \mathit{Poly}\ a \to \mathit{Poly}\ a \to \mathit{Poly}\ a \\
\mathit{polyAdd}\ (\mathit{Single}\ a)\ \ \ (\mathit{Single}\ b)\ \ \ = \mathit{Single}\ (a + b) \\
\mathit{polyAdd}\ (\mathit{Single}\ a)\ \ \ (\mathit{Cons}\ b\ bs) = \mathit{Cons}\ (a + b)\ bs \\
\mathit{polyAdd}\ (\mathit{Cons}\ a\ as)\ (\mathit{Single}\ b)\ \ \ = \mathit{Cons}\ (a + b)\ as \\
\mathit{polyAdd}\ (\mathit{Cons}\ a\ as)\ (\mathit{Cons}\ b\ bs) = \mathit{Cons}\ (a + b)\ (\mathit{polyAdd}\ as\ bs) \\
\mathit{polyMul} :: \mathit{Num}\ a \Rightarrow \mathit{Poly}\ a \to \mathit{Poly}\ a \to \mathit{Poly}\ a \\
\mathit{polyMul}\ (\mathit{Single}\ a)\ \ \ (\mathit{Single}\ b)\ \ \ = \mathit{Single}\ (a * b) \\
\mathit{polyMul}\ (\mathit{Single}\ a)\ \ \ (\mathit{Cons}\ b\ bs) = \mathit{Cons}\ (a * b)\ (\mathit{polyMul}\ (\mathit{Single}\ a)\ bs) \\
\mathit{polyMul}\ (\mathit{Cons}\ a\ as)\ (\mathit{Single}\ b)\ \ \ = \mathit{Cons}\ (a * b)\ (\mathit{polyMul}\ as\ (\mathit{Single}\ b)) \\
\mathit{polyMul}\ (\mathit{Cons}\ a\ as)\ (\mathit{Cons}\ b\ bs) = \mathit{Cons}\ (a * b)\ (\mathit{polyAdd}\ (\mathit{polyMul}\ as\ (\mathit{Cons}\ b\ bs)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{polyMul}\ (\mathit{Single}\ a)\ bs)) \\
\mathit{polyNeg} :: \mathit{Num}\ a \Rightarrow \mathit{Poly}\ a \to \mathit{Poly}\ a \\
\mathit{polyNeg} = \mathit{fmap}\ \mathit{negate}
\end{array}
$$

Therefore, we *can* define a ring structure (the mathematical counterpart of Num) on $\mathit{Poly}\ a$, 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 $\mathit{Poly}\ A$ together with the operations defined above is the ring of **polynomials** with coefficients in A.

The functions $\mathit{evalPoly}\ as$ are known as *polynomial functions*.

Caveat: The canonical representation of polynomials in algebra does not use finite lists, but the equivalent

$$\mathit{Poly}'\ A = \{\, a : \mathbb{N} \to A \mid \ \text{\{-}a\text{ has only a finite number of non-zero values -\}}\ \}$$

Exercise: what are the ring operations on $\mathit{Poly}'\ A$? For example, here is addition:

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

Observations:

1. 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.

2. 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

   $$\begin{array}{l} x :: Num\ a \Rightarrow Poly\ a \\ x = Cons\ 0\ (Single\ 1) \end{array}$$

   Then (again, using the list notation for brevity) for any polynomial $as = [a0, a1, ..., a_n]$ we have

   $$as = a0 + a1 * x + a2 * x\hat{\ }2 + ... + a_n * x\hat{\ }n$$

   Exercise: check this.

   This justifies the standard notation

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

7.3 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

$$\begin{array}{l} degree\ (Single\ 1) = 0 \\ \forall\ x, y.\ degree\ (x\ `op`\ y) = degree\ x + degree\ y \end{array}$$

The first law is no problem and for most polynomials the second law is also straighforward to prove (exercise: prove it) except for 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

$$
\begin{aligned}
& z \\
= \; & \{\text{-assumption -}\} \\
& \mathit{degree}\ 0 \\
= \; & \{\text{-simple calculation -}\} \\
& \mathit{degree}\ (0 * p) \\
= \; & \{\text{-homomorphism condition -}\} \\
& \mathit{degree}\ 0 + \mathit{degree}\ p \\
= \; & \{\text{-assumption -}\} \\
& z + n
\end{aligned}
$$

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 $z = -\mathit{Infinity}$, 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 $(\mathit{Poly}\ a, 1, *)$ to $(\mathit{Maybe}\ \mathbb{N}, \mathit{Nothing}, \mathit{opMaybe})$.

TODO: check all the properties.

8 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.

$$\mathit{PowerSeries}'\ a = \{f : \mathbb{N} \to a\}$$

```
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 evaluation of a power series represented by $a{:}\mathbb{N}{\rightarrow}A$ is defined, in case the necessary operations make sense on A, as a function

$$\begin{aligned}
&\mathit{eval}\ a : A \rightarrow A \\
&\mathit{eval}\ a\ x = \mathit{lim}\ s\ \textbf{where}\ s\ n = \textstyle\sum_{i=0}^{n} a_i * x^i
\end{aligned}$$

Note that $\mathit{eval}\ a$ 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 $\mathit{eval}\ a = \mathit{eval}\ b$.

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.

$$\begin{aligned}
&\mathit{eval}\ n\ \mathit{as}\ x = \mathit{evalPoly}\ (\mathit{takePoly}\ n\ \mathit{as})\ x \\
&\mathit{takePoly} :: \mathit{Integer} \rightarrow \mathit{Poly}\ a \rightarrow \mathit{Poly}\ a \\
&\mathit{takePoly}\ n\ (\mathit{Single}\ a) \quad\ \ = \mathit{Single}\ a \\
&\mathit{takePoly}\ n\ (\mathit{Cons}\ a\ \mathit{as}) = \textbf{if}\ n \leqslant 1 \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ \mathit{Single}\ a \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ \mathit{Cons}\ a\ (\mathit{takePoly}\ (n-1)\ \mathit{as})
\end{aligned}$$

9 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 $\mathbb{Q}$). Assume that $a * b \neq 0$. Then (again, using list notation for brevity), we want to find, for any given $(a : \mathit{as})$ and $(b : \mathit{bs})$, the series $(c : \mathit{cs})$ satisfying

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

This leads to the implementation:

$$\begin{aligned}
&\textbf{instance}\ (\mathit{Eq}\ a, \mathit{Fractional}\ a) \Rightarrow \mathit{Fractional}\ (\mathit{PowerSeries}\ a)\ \textbf{where} \\
&\quad (/) = \mathit{divPS} \\
&\quad \mathit{fromRational} = \mathit{Single} \circ \mathit{fromRational} \\
&\mathit{divPS} :: (\mathit{Eq}\ a, \mathit{Fractional}\ a) \Rightarrow \mathit{PowerSeries}\ a \rightarrow \mathit{PowerSeries}\ a \rightarrow \mathit{PowerSeries}\ a
\end{aligned}$$

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

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:

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

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).

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

10 Formal derivative

Considering the analogy between power series and polynomial functions (via polynomials), we can define a formal derivative for power series according to the formula

$$
(\sum_{n=0}^{\infty} a_n * x^n)' = \sum_{n=0}^{\infty} (a_n * x^n)' = \sum_{n=0}^{\infty} (a_n * (n * x^{n-1})) = \sum_{n=0}^{\infty} ((n * a_n) * x^{n-1})
$$

We can implement this, for example, as

$$
\begin{array}{l}
\mathit{deriv}\ (\mathit{Single}\ a)\ \ \ = \mathit{Single}\ 0 \\
\mathit{deriv}\ (\mathit{Cons}\ a\ \mathit{as}) = \mathit{deriv}'\ \mathit{as}\ 1 \\
\quad \mathbf{where}\ \mathit{deriv}'\ (\mathit{Single}\ a)\ \ \ \ n = \mathit{Single}\ (n * a) \\
\qquad\qquad \mathit{deriv}'\ (\mathit{Cons}\ a\ \mathit{as})\ n = \mathit{Cons}\ \ (n * a)\ (\mathit{deriv}'\ \mathit{as}\ (n + 1))
\end{array}
$$

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.

$$
\begin{array}{l}
\mathit{checkDeriv} :: \mathit{Integer} \to \mathit{Bool} \\
\mathit{checkDeriv}\ n = \mathit{takePoly}\ n\ (\mathit{deriv}\ \mathit{ps0}) == \mathit{takePoly}\ n\ \mathit{ps1}
\end{array}
$$

11 Signals and Shapes

Shallow and deep embeddings of a DSL

TODO: textify DSL/

12 Helpers

instance *Functor Poly* **where**
  $fmap = fmapPoly$

$fmapPoly :: (a \to b) \to (Poly\ a \to Poly\ b)$
$fmapPoly\ f\ (Single\ a) \quad = Single\ (f\ a)$
$fmapPoly\ f\ (Cons\ a\ as) = Cons\ (f\ a)\ (fmapPoly\ f\ as)$

$po1 :: Num\ a \Rightarrow Poly\ a$
$po1 = 1 + x\hat{}2 - 3 * x\hat{}4$

References

R. A. Adams and C. Essex. *Calculus: a complete course*. Pearson Canada, 7th edition, 2010.

C. H. Edwards, D. E. Penney, and D. Calvis. *Elementary Differential Equations*. Pearson Prentice Hall Upper Saddle River, NJ, 6h edition, 2008.

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.

E. Landau. *Einführung in die Differentialrechnung und Integralrechnung*. Noordhoff, 1934.

E. Landau. *Differential and Integral Calculus*. AMS/Chelsea Publication Series. AMS Chelsea Pub., 2001.

S. Mac Lane. *Mathematics: Form and function*. Springer New York, 1986.

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.