Domain Specific Languages of Mathematics: Lecture Notes

Patrik Jansson   Cezar Ionescu

January 30, 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:

```
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:

```
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*:

```
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.

```
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:

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

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:

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

Again, at this stage there are many open questions. For example, we can assume that *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*:

$re :: ComplexSem\ r \ \rightarrow r$
$re\ z@(CS\ (x, y))\ = x$

$im :: ComplexSem\ r \rightarrow r$
$im\ z@(CS\ (x, y)) = y$

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:

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

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:

$$evalE :: ComplexE \to ComplexD$$

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:

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

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

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

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

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

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,

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

and that *fromCS* is an embedding:

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

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

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

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:

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

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.

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

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:

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

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*:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{ComplexSyn}\ r = \mathit{FromCartesian}\ r\ r \\
&\qquad\qquad\qquad\qquad\ \mid \mathit{ComplexSyn}\ r\ :+:\ \mathit{ComplexSyn}\ r \\
&\qquad\qquad\qquad\qquad\ \mid \mathit{ComplexSyn}\ r\ :*:\ \mathit{ComplexSyn}\ r \\
&\mathit{toComplexSyn} :: \mathit{Num}\ a \Rightarrow a \to \mathit{ComplexSyn}\ a \\
&\mathit{toComplexSyn}\ x = \mathit{FromCartesian}\ x\ (\mathit{fromInteger}\ 0) \\
&\mathit{evalCSyn} :: \mathit{Num}\ r \Rightarrow \mathit{ComplexSyn}\ r \to \mathit{ComplexSem}\ r \\
&\mathit{evalCSyn}\ (\mathit{FromCartesian}\ x\ y) = \mathit{CS}\ (x, y) \\
&\mathit{evalCSyn}\ (l\ :+:\ r) = \mathit{evalCSyn}\ l +.\ \mathit{evalCSyn}\ r \\
&\mathit{evalCSyn}\ (l\ :*:\ r) = \mathit{evalCSyn}\ l\ *.\ \mathit{evalCSyn}\ r \\
&\mathbf{instance}\ \mathit{Num}\ a \Rightarrow \mathit{Num}\ (\mathit{ComplexSyn}\ a)\ \mathbf{where} \\
&\quad (+) = (:+:) \\
&\quad (*) = (:*:) \\
&\quad \mathit{fromInteger} = \mathit{fromIntegerCS} \\
&\qquad \text{-- TODO: add a few more operations (hint: extend ComplexSyn as well)} \\
&\qquad \text{-- TODO: also extend eval} \\
&\mathit{fromIntegerCS} :: \mathit{Num}\ r \Rightarrow \mathit{Integer} \to \mathit{ComplexSyn}\ r \\
&\mathit{fromIntegerCS} = \mathit{toComplexSyn} \circ \mathit{fromInteger}
\end{aligned}
$$

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 *a*s). 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	$\mathit{CPlus}_1\ 3\ 2\ i$
7/2 - 2/3 i = 7/2 + (-2/3) i	$\mathit{CPlus}_1\ (7\ /\ 2)\ (-2\ /\ 3)\ i$
i pi = 0 + i pi	$\mathit{CPlus}_2\ 0\ i\ \pi$
-3 = -3 + 0 i	$\mathit{CPlus}_1\ (-3)\ 0\ i$

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 ∘ typically used in mathematics. (The symbol ∘ 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.

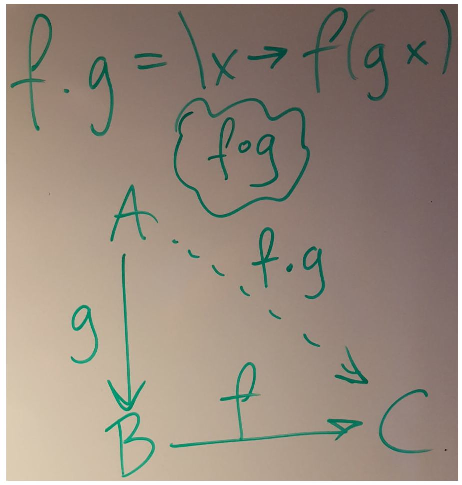

Figure 1: Function composition diagram

1.4.2 fromInteger (looks recursive)

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

$$\begin{array}{l}\textbf{instance}\ \mathit{Num}\ r \Rightarrow \mathit{Num}\ (\mathit{ComplexSyn}\ r)\ \textbf{where}\\ \quad\quad \text{-- ... several other methods and then}\\ \quad \mathit{fromInteger} = \mathit{toComplexSyn} \circ \mathit{fromInteger}\end{array}$$

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 $\mathit{fromInteger} :: \mathit{Integer} \to r$ and $\mathit{toComplexSyn} :: r \to \mathit{ComplexSyn}\ r$ while the resulting function is $\mathit{fromIntC} :: \mathit{Integer} \to \mathit{ComplexSyn}\ r$. 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:

$$\mathit{fromIntC} :: \mathit{Num}\ r \Rightarrow \mathit{Integer} \to \mathit{ComplexSyn}\ r$$

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.

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

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

$$\begin{array}{ll}\textbf{newtype}\ \mathit{Age}\ \ = \mathit{Ag}\ \mathit{Int} & \text{-- Age in years} \\ \textbf{newtype}\ \mathit{Shoe} = \mathit{Sh}\ \mathit{Int} & \text{-- Shoe size (EU)}\end{array}$$

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

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

In the lecture notes we used a newtype for the semantics of complex numbers as a pair of numbers in the cartesian representation but may also be useful to have another newtype for complex as a pair of numbers in the polar representation.

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

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

This declaration introduces

- a new type E for simple arithmetic expressions,
- a constructor $V :: String \to E$ 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"`, $e1 = P\ x\ x$, $e2 = T\ e1\ e1$

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

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

Example values: $y = V$ `"y"`, $e1 = y :+ y$, $e2 = x :* e1$

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

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

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

```
type Env v s = [(v, s)]
```

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

Example value:

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

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

```
evalCP :: Eq v => Env v (ComplexSem r) -> (ComplexSy v r -> ComplexSem r)
evalCP env (Var x) = case lookup x env of
  Just c -> undefined   -- ...
  -- ...
```

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

1.5 Some helper functions

```
propAssocAdd :: (Eq a, Num a) => a -> a -> a -> Bool
propAssocAdd = propAssocA (+)

(*.) :: Num r => ComplexSem r -> ComplexSem r -> ComplexSem r
CS (ar, ai) *. CS (br, bi) = CS (ar * br - ai * bi, ar * bi + ai * br)

instance Show r => Show (ComplexSem r) where
  show = showCS

showCS :: Show r => ComplexSem r -> String
showCS (CS (x, y)) = show x ++ " + " ++ show y ++ "i"
```

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).

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

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 **?**, 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

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

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

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

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

(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.

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

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:

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

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.

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

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

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

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:

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

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:

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

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

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

$$
\begin{aligned}
&\mathit{checkRP} :: (\mathit{Eq}\ v, \mathit{Show}\ v) \Rightarrow \mathit{Env}\ v\ \mathit{RatSem} \to \mathit{RPred}\ v \to \mathit{Bool}\\
&\mathit{checkRP}\ \mathit{env}\ (\mathit{Equal}\ \ \ t1\ t2) = \mathit{eqSem}\ \ \ (\mathit{evalRat2}\ \mathit{env}\ t1)\ (\mathit{evalRat2}\ \mathit{env}\ t2)\\
&\mathit{checkRP}\ \mathit{env}\ (\mathit{LessThan}\ t1\ t2) = \mathit{lessThanSem}\ (\mathit{evalRat2}\ \mathit{env}\ t1)\ (\mathit{evalRat2}\ \mathit{env}\ t2)\\
&\mathit{checkRP}\ \mathit{env}\ (\mathit{Positive}\ \ t1) \ \ = \mathit{positiveSem}\ \ (\mathit{evalRat2}\ \mathit{env}\ t1)\\
&\mathit{checkRP}\ \mathit{env}\ (\mathit{And}\ p\ q)\ \ = (\mathit{checkRP}\ \mathit{env}\ p) \land (\mathit{checkRP}\ \mathit{env}\ q)\\
&\mathit{checkRP}\ \mathit{env}\ (\mathit{Not}\ p)\ \ \ \ = \neg\ (\mathit{checkRP}\ \mathit{env}\ p)
\end{aligned}
$$

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

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

2.6 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

```
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

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

3.1 Types in mathematics

Types are sometimes mentioned explicitly in mathematical texts:

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

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 which doesn't rely on the names given to the arguments was popularised by Landau in Landau (1934) (English edition Landau (2001)): D_1 for the partial derivative with respect to x_1, etc.
- Exercise: for $\mathrm{f} : \mathbb{R}^2 \to \mathbb{R}$ 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:

$lim : (X \to \mathbb{R}) \to \{p \mid p \in \mathbb{R}, p \ \textit{limit point} \ \textbf{of} \ X\} \to \mathbb{R}$

$z = f\ (x, y)$ 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:

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

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:

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

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

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

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

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

   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

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

   which is equivalent to n functions of type $\mathbb{R} \to \mathbb{R}$. 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:

   $q : \mathbb{R} \to \mathbb{R}$
   $q\ t = w\ t$
   $\dot{q} : \mathbb{R} \to \mathbb{R}$
   $\dot{q}\ t = dw\ /\ dt$

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

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

   where the "combinator" *expand* is given by

   $\mathit{expand} \quad : (\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R}^3$
   $\mathit{expand}\ w\ t = (t, w\ t, D\ w\ t)$

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

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

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

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

   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 $\mathbb{R} \to \mathbb{R}$. In particular, the right-hand 0 is the constant function

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

4 Types in Mathematics (Part II)

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: **?**

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:

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

[1] https://www.haskell.org/onlinereport/haskell2010/haskellch6.html#x13-1350142

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

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.

4.2 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):

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

$$\begin{array}{l} \mid \ expression + expression \\ \mid \ expression * expression \\ \mid \ exp\ expression \\ \mid \ \ldots \end{array}$$

etc.

We can implement this in a datatype:

$$test = 1$$

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

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

$$\begin{array}{lll} eval :: FunExp & \to Double \to Double & \\ eval\ \ (Const\ alpha) & = const\ alpha & \\ eval\ \ Id & = id & \\ eval\ \ (e1 :+: e2) & = eval\ e1 + eval\ e2 & \text{-- note the use of “lifted +”} \\ eval\ \ (e1 :*: e2) & = eval\ e1 * eval\ e2 & \text{-- “lifted *”} \\ eval\ \ (Exp\ e1) & = exp\ (eval\ e1) & \text{-- and “lifted } exp\text{”} \\ \ \ \text{-- and so on} & & \end{array}$$

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

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

In other words, for any expression e, we want

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

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

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

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

Therefore, the specification is fulfilled by taking

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

Similarly, we obtain

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

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

4.3 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

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

For example:

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

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

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:

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

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

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

Exercise: implement the rest

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.

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