3 Types in Mathematics

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

3.1 Types in mathematics

Types are sometimes mentioned explicitly in mathematical texts:

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

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

- $lim : (\mathbb{N} \to \mathbb{R}) \to \mathbb{R}$ for $lim_{n\to\infty}\{a_n\}$
- $d/dt : (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$
- sometimes, instead of df/dt one sees f' or $\dot{f}$ or $D\,f$
- $\partial f/\partial x_i : (\mathbb{R}^n \to \mathbb{R}) \to (\mathbb{R}^n \to \mathbb{R})$
- we mostly see $\partial f/\partial x$, $\partial f/\partial y$, $\partial f/\partial z$ etc. when, in the context, the function f has been given a definition of the form $f(x, y, z) = \ldots$
- a better notation (by Landau) which doesn't rely on the names given to the arguments was popularised in Landau [1934] (English edition Landau [2001]): D_1 for the partial derivative with respect to x_1, etc.
- Exercise: for $f : \mathbb{R}^2 \to \mathbb{R}$ define D_1 and D_2 using only D.

3.2 Typing Mathematics: derivative of a function

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

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

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

Now, with the type of D sorted out, we can turn to the actual definition of the function $D\,f$. The definition is given for a fixed (but arbitrary) x. (At this point it is useful to briefly look back to the definition of "limit of a function" in Section 2.13.) The *lim* expression is using the (anonymous) function $g\ h = \frac{f\ (x+h) - f\ x}{h}$ and that the limit of g is taken at 0. Note that g is defined in the scope of x and that its definition uses x so it can be seen as having x as an implicit, first argument. To be more explicit we write $\varphi\ x\ h = \frac{f\ (x+h) - f\ x}{h}$ and take the limit of $\varphi\ x$ at 0. So, to sum up, $D\ f\ x = lim\ (\varphi\ x)\ 0$. [3]

The key here is that we name, type, and specify the operation of computing the derivative (of a one-argument function). We will use this operation quite a bit in the rest of the book, but here are just a few examples to get used to the notation.

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

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

3.3 Typing Mathematics: partial derivative

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

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

What are the types of the elements involved? We have

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

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

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

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

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

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

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

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

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

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

On line 1, $z = f\ (x, y)$ probably does not mean that $z \in \mathbb{R}$, although the phrase "the quantity z" (on line 2) suggests this. A possible interpretation is that z is used to abbreviate the expression $f\ (x, y)$; thus, everywhere we can replace z with $f\ (x, y)$. In particular, $\partial z\ /\ \partial x$ becomes $\partial f\ (x, y)\ /\ \partial x$, which we can interpret as $\partial f\ /\ \partial x$ applied to (x, y) (remember that (x, y) is bound in the context by a universal quantifier on line 4). There is the added difficulty that, just like the subscript in f'_x, the x in ∂x is not the x bound by the universal quantifier, but just a symbol.

3.4 Type inference and understanding: Lagrangian case study

From (Sussman and Wisdom 2013):

> A mechanical system is described by a Lagrangian function of the system state (time, coordinates, and velocities). A motion of the system is described by a path that gives the coordinates for each moment of time. A path is allowed if and only if it satisfies the Lagrange equations. Traditionally, the Lagrange equations are written
>
> $$\frac{d}{dt}\frac{\partial L}{\partial \dot{q}} - \frac{\partial L}{\partial q} = 0$$
>
> What could this expression possibly mean?

To start answering the question, we start typing the elements involved:

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

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

   This is consistent with the description: "Lagrangian function of the system state (time, coordinates, and velocities)". So, if we let "coordinates" be just one coordinate, we can take $i = 3$:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The equation becomes

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

or, after simplification:

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

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

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

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

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

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

3.5 Playing with types

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

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

Typing the variables and the integration operators in this text was an exam question in 2016 and we will use it here as an example of getting feedback from a type checker. We start by declaring two types, X and Y, and a function f between them:

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

These "empty" **data**-declarations mean that Haskell now knows the types exist, but nothing about any values of those types. Similarly, f has a type, but no proper implementation. We will declare types of the rest of the variables as well, and as we are not implementing any of them right now, we can just make one "dummy" implementation of a few of them in one go:

```
(x, deriv, ff, a, b, int) = undefined
```

We write *ff* for the capital F (to fit with Haskell's rules for variable names), *deriv* for the postix prime, and *int* for the integral operator. On line 6 "values of x" hints at the type X for x and the way y is used indicates that it is to be seen as an alias for f (and thus must have the same type) As we have discussed above, the derivative normally preserves the type and thus we can write:

```
x  :: X
y  :: X → Y
y  = f
y' :: X → Y
y' = deriv f
deriv :: (X → Y) → (X → Y)
```

Next up (on line 9) is the "expression F" (which we write *ff*). It should take three arguments: y, y', x, and return "a quantity". We can invent a new type Z and write:

```
data Z   -- Probably also some subset of the real numbers
ff :: (X → Y) → (X → Y) → X → Z
```

Then we have the operation of definite integration, which we know should take two limits $a, b :: X$ and a function $X \to Z$. The traditional mathematics notation for integration uses an expression (in x) followed by dx, but we can treat that as a function *expr* binding x:

$$
\begin{aligned}
&a, b :: X \\
&integral = int\ a\ b\ expr \\
&\quad \textbf{where}\ expr\ x = ff\ y\ y'\ x \\
&int :: X \to X \to (X \to Z) \to Z
\end{aligned}
$$

Now we have reached a stage where all the operations have types and the type checker is happy with them. At this point it is possible to experiment with variations based on alternative interpretations of the text. For this kind of "refactoring" is very helpful to have the type checker to make sure the types still make sense. For example, we could write $ff2 :: Y \to Y \to X \to Z$ as a variant of ff as long as we also change the expression in the integral:

$$
\begin{aligned}
&ff2 :: Y \to Y \to X \to Z \\
&ff2 = undefined \\
&integral2 = int\ a\ b\ expr \\
&\quad \textbf{where}\ expr\ x = ff2\ y\ y'\ x \\
&\qquad\qquad \textbf{where}\ y\ = f\ x \\
&\qquad\qquad\qquad\quad y' = deriv\ f\ x
\end{aligned}
$$

Both versions (and a few more minor variations) would be fine as exam solutions, but not something where the types don't match up.

3.6 Type in Mathematics (Part II)

3.6.1 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

$$fromInteger :: Integer \to a$$

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*.

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

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

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

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

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

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

Exercise: play around with this a bit in ghci.

3.6.2 Overloaded integers literals

As an aside, we will spend some time explaining a convenient syntactic shorthand which is very useful but which can be confusing: overloaded integers. In Haskell, every use of an integer literal like 2, 1738, etc., is actually implicitly an application of *fromInteger* to the literal. This means that the same program text can have different meaning depending on the type of the context. The literal $\mathit{three} = 3$, for example, can be used as an integer, a real number, a complex number, or even as a (constant) function (by the instance $\mathit{Num}\ (x \to a)$).

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

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

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

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

I have placed the types in the comment, with "backwards-pointing" arrows indicating that *fromInteger*:: $\mathit{Integer} \to r$ 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:

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

As an example we have that

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

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

3.6.3 Back to the numeric hierarchy instances for functions

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

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

```
class Num a ⇒ Fractional a where
  (/)          :: a → a → a
  recip        :: a → a          -- λx → 1 / x
  fromRational :: Rational → a   -- similar to fromInteger
```

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

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

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

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


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

Exercise: complete the instance declarations.

These type classes represent an abstract language of algebraic and standard operations, abstract in the sense that the exact nature of the elements involved is not important from the point of view of the type class, only from that of its implementation.

3.7 Type classes in Haskell

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

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

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

Instance declarations can also be parameterised:

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

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

3.8 Computing derivatives

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

```
D (f + g)   = D f + D g
D (f * g)   = D f * g + f * D g
D (f ∘ g) x = D f (g x) * D g x   -- the chain rule
D (const a) = const 0
D id        = const 1
D (^n) x    = n * (x^(n - 1))
D sin x     = cos x
D cos x     = - (sin x)
D exp x     = exp x
```

and so on.

If we want to get a bit closer to actually implementing D we quickly notice a problem: if D has type $(\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$ we have no way of telling which of these rules we should apply. Given a real (semantic) function f as an argument, D cannot know if this function was written using a $+$, or *sin* or *exp* as outermost operation. The only thing D could do would be to numerically approximate the derivative, and that is not what we are exploring in this course. Thus we need

to take a step back and change the type that we work on. All the rules in the table seem to work on *syntactic* functions: abstract syntax trees *representing* the real (semantic) functions.

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

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

We can implement this in a datatype:

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

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

$$
\begin{aligned}
&\textbf{type}\ \mathit{Func} = \mathbb{R} \to \mathbb{R} \\
&\mathit{eval} :: \mathit{FunExp} \to \mathit{Func} \\
&\mathit{eval}\ \ (\mathit{Const}\ \alpha) = \mathit{const}\ \alpha \\
&\mathit{eval}\ \ \mathit{Id} = \mathit{id} \\
&\mathit{eval}\ \ (e_1 :+: e_2) = \mathit{eval}\ e_1 + \mathit{eval}\ e_2 \quad \text{-- note the use of ``lifted +'',} \\
&\mathit{eval}\ \ (e_1 :*: e_2) = \mathit{eval}\ e_1 * \mathit{eval}\ e_2 \quad \text{-- ``lifted *'',} \\
&\mathit{eval}\ \ (\mathit{Exp}\ e_1) = \mathit{exp}\ (\mathit{eval}\ e_1) \quad \text{-- and ``lifted } \mathit{exp}\text{''.} \\
&\quad \text{-- and so on}
\end{aligned}
$$

An example:

$$
\begin{aligned}
&f_1 :: \mathbb{R} \to \mathbb{R} \\
&f_1\ x = \mathit{exp}\ (x\hat{}2) \\
&e_1 :: \mathit{FunExp} \\
&e_1 = \mathit{Exp}\ (\mathit{Id} :*: \mathit{Id})
\end{aligned}
$$

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

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

As a formula we want

$$\mathit{eval} \circ \mathit{derive}\ e = D \circ \mathit{eval}$$

or, in other words, for any expression $e :: \mathit{FunExp}$, we want

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

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

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

Therefore, the specification is fulfilled by taking

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

Similarly, we obtain

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

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

3.9 Shallow embeddings

The DSL of expressions, whose syntax is given by the type *FunExp*, turns out to be almost identical to the DSL defined via type classes in the first part of this lecture. The correspondence between them is given by the *eval* function.

The difference between the two implementations is that the first one separates more cleanly from the semantical one. For example, :+: *stands for* a function, while + *is* that function.

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

Note that the reason the shallow embedding is possible is that the *eval* function is a *fold*: first evaluate the sub-expressions of e, then put the evaluations together without reference to the sub-expressions. This is sometimes referred to as "compositionality".

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

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

For example:

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

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

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

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

Is *evalD* compositional?

We compute, for example:

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

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

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

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

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

Exercise: implement the rest of the *Num* instance for $FD\ a$.

3.10 Exercises

Exercise 3.1. To get a feeling for the Lagrange equations, let $L\ (t, q, v) = m * v\hat{}2\ /\ 2 + m * g * q$, compute *expand* w, perform the derivatives and check if the equation is satisfied for

- $w_1 = id$ or
- $w_2 = sin$ or
- $w_3 = (q0-) \circ (g*) \circ (/2) \circ (\hat{}2)$

3.11 Exercises from old exams

Exercise 3.2. *From exam 2016-Practice*

Consider the following text from Mac Lane's *Mathematics: Form and Function* (page 168):

> If $z = g(y)$ and $y = h(x)$ are two functions with continuous derivatives, then in the relevant range $z = g(h(x))$ is a function of x and has derivative
>
> $$z'(x) = g'(y) * h'(x)$$

Give the types of the elements involved (x, y, z, g, h, z', g', h', $*$ and ').

Exercise 3.3. *From exam 2016-03-16*

Consider the following text from Mac Lane's *Mathematics: Form and Function* (page 182):

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

Give the types of the variables involved (x, y, y', f, F, a, b) and the type of the four-argument integration operator:

$$\int_{\cdot}^{\cdot} \cdot d \cdot$$

Exercise 3.4. *From exam 2016-08-23*

In the simplest case of probability theory, we start with a *finite*, non-empty set Ω of *elementary events*. *Events* are subsets of Ω, i.e. elements of the powerset of Ω, (that is, $\mathcal{P}\Omega$). a *probability function* P associates to each event a real number between 0 and 1, such that

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

Conditional probabilities are defined as follows (*Elementary Probability 2nd Edition*, Stirzaker 2003):

> Let A and B be events with $P\ B > 0$. given that B occurs, the *conditional probability* that A occurs is denoted by $P\ (A \mid B)$ and defined by
> $P\ (A \mid B) = P\ (A \cap B)\ /\ P\ B$

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

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

Exercise 3.5. *From exam 2017-03*

(Note that this exam question was later included as one of the examples in this chapter, see 3.3. It is kept here in case you want to check if you remember it!)

Consider the following text from page 169 of Mac Lane [1968]:

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

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

What are the types of the elements involved in the equation on the last line? You are welcome to introduce functions and names to explain your reasoning.

Exercise 3.6. *From exam 2017-08-22*

Multiplication for matrices (from the matrix algebra DSL).

Consider the following definition, from "Linear Algebra" by Donald H. Pelletier:

Definition: If A is an $m \times n$ matrix and B is an $n \times p$ matrix, then the *product*, AB, is an $m \times p$ matrix; the $(i, j)^{th}$ entry of AB is the sum of the products of the pairs that are obtained when the entries from the i^{th} row of the left factor, A, are paired with those from the j^{th} column of the right factor, B.

a. Introduce precise types for the variables involved: A, m, n, B, p, i, j. You can write *Fin n* for the type of the values $\{0, 1, ..., n - 1\}$.

b. Introduce types for the functions *mul* and *proj* where AB = *mul A B* and *proj i j M* = "take the $(i, j)^{th}$ entry of M". What class contraints (if any) are needed on the type of the matrix entries in the two cases?

c. Implement *mul* in Haskell. You may use the functions *row* and *col* specified by *row i M* = "the i^{th} row of M" and *col j M* = "the j^{th} column of M". You don't need to implement them and here you can assume they return plain Haskell lists.

Exercise 3.7. (Extra material outside the course.) In the same direction as the Lagrangian case study in section 3.4 there are two nice blog posts about Hamiltonian dynamics: one introductory and one more advanced. It is a good exercise to work through the examples in these posts.