4 Compositional Semantics and Algebraic Structures

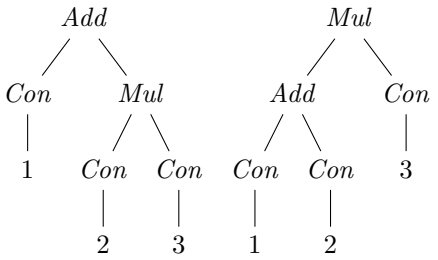
```
{-# LANGUAGE FlexibleInstances, GeneralizedNewtypeDeriving #-}
module DSLsofMath.W04 where
import Prelude hiding (Monoid)
import DSLsofMath.FunExp
```

4.1 Compositional semantics

4.1.1 A simpler example of a non-compositional function

Consider a very simple datatype of integer expressions:

```
data E = Add E E | Mul E E | Con Integer deriving Eq
e1, e2 :: E                                  -- 1 + 2 * 3
e1 = Add (Con 1) (Mul (Con 2) (Con 3))       -- 1 + (2 * 3)
e2 = Mul (Add (Con 1) (Con 2)) (Con 3)       -- (1 + 2) * 3
```



When working with expressions it is often useful to have a "pretty-printer" to convert the abstract syntax trees to strings like `"1+2*3"`.

```
pretty :: E → String
```

We can view *pretty* as an alternative *eval* function for a semantics using *String* as the semantic domain instead of the more natural *Integer*. We can implement *pretty* in the usual way as a "fold" over the syntax tree using one "semantic constructor" for each syntactic constructor:

```
pretty (Add x y) = prettyAdd (pretty x) (pretty y)
pretty (Mul x y) = prettyMul (pretty x) (pretty y)
pretty (Con c)   = prettyCon c
prettyAdd :: String → String → String
prettyMul :: String → String → String
prettyCon :: Integer → String
```

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

```
prettyAdd xs ys = xs ++ "+" ++ ys
prettyMul xs ys = xs ++ "*" ++ ys
prettyCon c     = show c
p1, p2 :: String
p1 = pretty e1
p2 = pretty e2
```

$trouble :: Bool$
$trouble = p_1 == p_2$

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

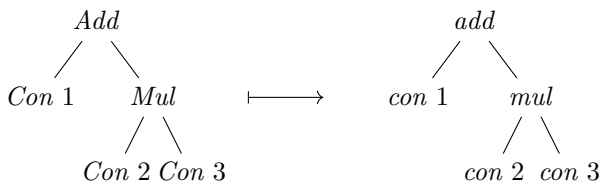
For the curious. One solution to the problem with parentheses is to create three (slightly) different functions intended for printing in different contexts. The first of them is for the top level, the second for use inside *Add*, and the third for use inside *Mul*. These three functions all have type $E \to String$ and can thus be combined with the tupling transform into one function returning a triple: $prVersions :: E \to (String, String, String)$. The result is the following:

```
prTop :: E -> String
prTop e = let (pTop, _, _) = prVersions e
          in pTop

type ThreeVersions = (String, String, String)
prVersions :: E -> ThreeVersions
prVersions = foldE prVerAdd prVerMul prVerCon

prVerAdd :: ThreeVersions -> ThreeVersions -> ThreeVersions
prVerAdd (xTop, xInA, xInM) (yTop, yInA, yInM) =
  let s = xInA ++ "+" ++ yInA     -- use InA because we are "in Add"
  in (s, paren s, paren s)        -- parens needed except at top level

prVerMul :: ThreeVersions -> ThreeVersions -> ThreeVersions
prVerMul (xTop, xInA, xInM) (yTop, yInA, yInM) =
  let s = xInM ++ "*" ++ yInM     -- use InM because we are "in Mul"
  in (s, s, paren s)              -- parens only needed inside Mul

prVerCon :: Integer -> ThreeVersions
prVerCon i =
  let s = show i
  in (s, s, s)                    -- parens never needed

paren :: String -> String
paren s = "(" ++ s ++ ")"
```

Exercise: Another way to make this example go through is to refine the semantic domain from *String* to $Precedence \to String$. This can be seen as another variant of the result after the tupling transform: if *Precedence* is an n-element type then $Precedence \to String$ can be seen as an n-tuple. In our case a three-element *Precedence* would be enough.

4.1.2 Compositional semantics in general

In general, for a syntax *Syn*, and a possible semantics (a type *Sem* and an *eval* function of type $Syn \to Sem$), we call the semantics *compositional* if we can implement *eval* as a fold. Informally a "fold" is a recursive function which replaces each abstract syntax constructor *Ci* of *Syn* with a "semantic constructor" *ci*. Thus, in our datatype *E*, a compositional semantics means that *Add* maps to *add*, *Mul* $\mapsto$ *mul*, and *Con* $\mapsto$ *con* for some "semantic functions" *add*, *mul*, and *con*.



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

```
foldE :: (s → s → s) → (s → s → s) → (Integer → s) → (E → s)
foldE add mul con = rec
  where rec (Add x y) = add (rec x) (rec y)
        rec (Mul x y) = mul (rec x) (rec y)
        rec (Con i)   = con i
```

Notice that *foldE* has three function arguments corresponding to the three constructors of E. The "natural" evaluator to integers is then easy:

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

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

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

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

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

Finally, it is often useful to capture the semantic functions (the parameters to the fold) in a type class:

```
class IntExp t where
  add :: t → t → t
  mul :: t → t → t
  con :: Integer → t
```

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

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

To get a more concrete feeling for this, we define some concrete values, not just functions:

```
seven :: IntExp a ⇒ a
seven = add (con 3) (con 4)
```

```
testI :: Integer
testI = seven

testE :: E
testE = seven

check :: Bool
check = and [testI == 7
  , testE == Add (Con 3) (Con 4)
  , testP == "3+4"
  ]
```

We can also see *pretty* as instance:

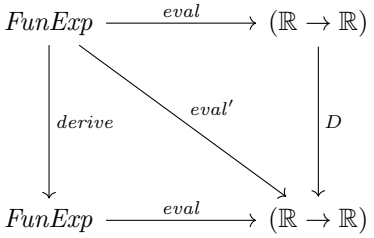
```
instance IntExp String where
  add = prettyAdd
  mul = prettyMul
  con = prettyCon

pretty' :: E → String
pretty' = foldIE

testP :: String
testP = seven
```

To sum up, by defining a class *IntExp* (and some instances) we can use the metods (*add*, *mul*, *con*) of the class as "smart constructors" which adapt to the context. An overloaded expression, like $seven :: Num\ a \Rightarrow a$, which only uses these smart constructors can be instantiated to different types, ranging from the syntax tree type E to different semantic interpretations (like *Integer*, and *String*).

4.1.3 Back to derivatives and evaluation

Review section 3.9 again with the definition of $eval'$ being non-compositional (just like *pretty*) and *evalD* a more complex, but compositional, semantics.

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



As we saw in section 3.9 this does not work in the sense that $eval'$ cannot directly be implemented compositionally. The problem is that some of the rules of computing the derivative depends not only on the derivative of the subexpressions, but also on the subexpressions before taking the derivative. A typical example of the problem is $derive\ (f :*: g)$ where the result involves not only $derive\ f$ and $derive\ g$, but also f and g.

The solution is to extend the the return type of $eval'$ from one semantic value f of type $Func = \mathbb{R} \to \mathbb{R}$ to two such values $(f, f') :: (Func, Func)$ where $f' = D\ f$. One way of expressing this is to say that in order to implement $eval' :: FunExp \to Func$ we need to also compute $eval :: FunExp \to Func$. Thus we need to implement a pair of *eval*-functions $(eval, eval')$ together. Using the "tupling transform" we can express this as computing just one function $evalD :: FunExp \to (Func, Func)$ returning a pair of f and $D\ f$ at once.

This combination *is* compositional, and we can then get $eval'$ back as the second component of $evalD\ e$:

$$eval' :: FunExp \to Func$$
$$eval' = snd \circ evalD$$

4.2 Algebraic Structures and DSLs

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

4.2.1 Algebras, homomorphisms

The matematical theory behind compositionality talks about homomorphisms between algebraic structures.

From Wikipedia:

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

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

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

```
exp :: ℝ      → ℝ>0
exp 0         = 1                 -- e^0 = 1
exp (a + b)   = exp a * exp b     -- e^(a+b) = e^a e^b
log :: ℝ>0    → ℝ
```

$$
\begin{array}{lll}
log\ 1 & = 0 & \text{-- } \log 1 = 0 \\
log\ (a * b) & = log\ a\ + log\ b & \text{-- } \log(ab) = \log a + \log b
\end{array}
$$

What we recognize as the familiar laws of exponentiation and logarithms are actually examples of the homomorphism conditions for *exp* and *log*. Back to Wikipedia:

> More formally, a homomorphism between two algebras A and B is a function $h: A \to B$ from the set A to the set B such that, for every operation f_A of A and corresponding f_B of B (of arity, say, n), $h\ (f_A\ (x_1, ..., x_n)) = f_B\ (h\ (x_1), ..., h\ (x_n))$.

Our examples *exp* and *log* are homomorphisms between monoids and the general monoid homomorphism conditions for $h : A \to B$ are:

$$
\begin{array}{lll}
h\ unit & = unit & \text{-- } h \text{ takes units to units} \\
h\ (x\ `op`\ y) & = h\ x\ `op`\ h\ y & \text{-- and distributes over } op \text{ (for all } x \text{ and } y)
\end{array}
$$

Note that both *unit* and *op* have different types on the left and right hand sides. On the left they belong to the monoid $(A, unit_A, op_A)$ and on the right the belong to $(B, unit_B, op_B)$.

To make this a bit more concrete, here are two examples of monoids in Haskell: the additive monoid *ANat* and the multiplicative monoid *MNat*.

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

In mathematical texts the constructors M and A are usually omitted and below we will stick to that tradition.

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

Solution:

Let $h : ANat \to MNat$ be a homomorphism. Then it must satisfy the following conditions:

$$
\begin{array}{lll}
h\ 0 & = 1 & \\
h\ (x + y) & = h\ x * h\ y & \text{-- for all } x \text{ and } y
\end{array}
$$

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

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

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

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

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

$$
\begin{array}{ll}
h\ a + h\ b & = \{\text{- } h = const \text{ in this case -}\} \\
const\ a + const\ b & = \{\text{- By def. of } (+) \text{ on functions -}\} \\
(\lambda x \to const\ a\ x + const\ b\ x) & = \{\text{- By def. of } const, \text{ twice -}\}
\end{array}
$$

$$
\begin{array}{ll}
(\lambda x \to a + b) & = \{\text{- By def. of } const \text{ -}\} \\
const\ (a + b) & = \{\text{- } h = const \text{ -}\} \\
h\ (a + b) &
\end{array}
$$

We now have a homomorphism from values to functions, and you may wonder if there is a homomorphism in the other direction. The answer is "Yes, many". Exercise: Show that *apply c* is a homomorphism for all c, where $apply\ x\ f = f\ x$.

4.2.2 Homomorphism and compositional semantics

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

```
instance Num FunExp where
  (+)         = (:+:)
  (*)         = (:*:)
  fromInteger = Const . fromInteger
instance Fractional FunExp where
    -- Exercise: fill in
instance Floating FunExp where
  exp         = Exp
    -- Exercise: fill in
```

and so on.

Exercise: complete the type instances for *FunExp*.

For instance, we have

$$
\begin{array}{l}
eval\ (e_1 :*: e_2) = eval\ e_1 * eval\ e_2 \\
eval\ (Exp\ e) \quad = exp\ (eval\ e)
\end{array}
$$

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

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

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

where *GoodClass* gives exactly the structure we need for the translation. With this class in place we can define generic expressions using smart constructors just like in the case of *IntExp* above. For example, we could define

$$
\begin{array}{l}
twoexp :: GoodClass\ a \Rightarrow a \\
twoexp = mulF\ (constF\ 2)\ (expF\ idF)
\end{array}
$$

and instantiate it to either syntax or semantics:

$$
\begin{array}{l}
testFE :: FunExp \\
testFE = twoexp \\
\\
testFu :: Func \\
testFu = twoexp
\end{array}
$$

Exercise: define the class *GoodClass* and instances for *FunExp* and $Func = \mathbb{R} \to \mathbb{R}$ to make the example work. Find another instance of *GoodClass*.

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

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

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

```
type Var    = String
data MExpr = Unit | Op MExpr MExpr | V Var
```

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

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

```
evalM :: Monoid a ⇒ (Var → a) → (MExpr → a)
```

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

```
evalM f Unit       = unit
evalM f (Op e1 e2) = op (evalM f e1) (evalM f e2)
evalM f (V x)      = f x
```

(Observation: In *FunExp*, the role of variables was played by $\mathbb{R}$, and the role of the assignment by the identity.)

The following correspondence summarises the discussion so far:

Computer Science	Mathematics
DSL	structure (category, algebra, ...)
deep embedding, abstract syntax	initial algebra
shallow embedding	any other algebra
semantics	homomorphism from the initial algebra

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

4.2.3 Other homomorphisms

In Section 3.6.1, we defined a *Num* instance for functions with a *Num* codomain. If we have an element of the domain of such a function, we can use it to obtain a homomorphism from functions to their codomains:

$$Num\ a \Rightarrow x \to (x \to a) \to a$$

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

$$apply :: a \to (a \to b) \to b$$
$$apply\ a = \lambda f \to f\ a$$

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

$$h\ (f + g) = \{\text{- def. } apply \text{ -}\}$$
$$(f + g)\ c = \{\text{- def. } + \text{ for functions -}\}$$
$$f\ c + g\ c = \{\text{- def. } apply \text{ -}\}$$
$$h\ f + h\ g$$

etc.

Can we do something similar for *FD*?

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

$$applyFD :: a \to FD\ a \to (a, a)$$
$$applyFD\quad c\quad (FD\ (f, f')) = FD\ (f\ c, f'\ c)$$

We now have the domain of the homomorphism ($FD\ a$) and the homomorphism itself ($applyFD\ c$), but we are missing the structure on the codomain, which now consists of pairs (a, a). In fact, we can *compute* this structure from the homomorphism condition. For example (we skip the constructor *FD* for brevity):

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

The identity will hold if we take

$$\textbf{type}\ Dup\ a = (a, a)$$
$$(\circledast) :: Num\ a \Rightarrow Dup\ a \to Dup\ a \to Dup\ a$$
$$(x, x') \circledast (y, y') = (x * y, x' * y + x * y')$$

Thus, if we define a "multiplication" on pairs of values using $(\circledast)$, we get that ($applyFD\ c$) is a *Num*-homomorphism for all c (or, at least for the operation $(*)$). We can now define an instance

```
instance Num a ⇒ Num (Dup a) where
  (*) = (⊛)
    -- ... exercise
```

Exercise: complete the instance declarations for $(\mathbb{R}, \mathbb{R})$.

Note: As this computation goes through also for the other cases we can actually work with just pairs of values (at an implicit point $c :: a$) instead of pairs of functions. Thus we can define a variant of *FD a* to be **type** *Dup* $a = (a, a)$

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

4.3 Summing up: definitions and representation

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

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

There is nothing in the "nature" of pairs of $\mathbb{R}$ that forces this definition upon us. We chose it, because of the intended interpretation.

This multiplication is obviously not the one we need for *complex numbers*:

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

Again, there is nothing in the nature of pairs that foists this operation on us. In particular, it is, strictly speaking, incorrect to say that a complex number *is* a pair of real numbers. The correct interpretation is that a complex number can be *represented* by a pair of real numbers, provided we define the operations on these pairs in a suitable way.

The distinction between definition and representation is similar to the one between specification and implementation, and, in a certain sense, to the one between syntax and semantics. All these distinctions are frequently obscured, for example, because of prototyping (working with representations / implementations / concrete objects in order to find out what definition / specification / syntax is most adequate). They can also be context-dependent (one man's specification is another man's implementation). Insisting on the difference between definition and representation can also appear quite pedantic (as in the discussion of complex numbers above). In general though, it is a good idea to be aware of these distinctions, even if they are suppressed for reasons of brevity or style. We will see this distinction again in section 5.1.

4.3.1 Some helper functions

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

4.4 Co-algebra and the Stream calculus

In the coming chapters there will be quite a bit of material on infinite structures. These are often captured not by algebras, but by co-algebras. We will not build up a general theory of co-algebras in these notes, but I could not resist to introduce a few examples which hint at the important role co-algebra plays in calculus.

Streams as an abstract datatype. Consider the API for streams of values of type A represented by some abstract type X:

$$
\begin{array}{l}
\textbf{data}\ X \\
\textbf{data}\ A \\
\mathit{head} :: X \to A \\
\mathit{tail} :: X \to X \\
\mathit{cons} :: A \to X \to X \\
\mathit{law1}\ \ s = \quad s == \mathit{cons}\ (\mathit{head}\ s)\ (\mathit{tail}\ s) \\
\mathit{law2}\ a\ s = \quad s == \mathit{tail}\ (\mathit{cons}\ a\ s) \\
\mathit{law3}\ a\ s = \quad a == \mathit{head}\ (\mathit{cons}\ a\ s)
\end{array}
$$

With this API we can use *head* to extract the first element of the stream, and *tail* to extract the rest as a new stream of type X. Using *head* and *tail* recursively we can extract an infinite list of values of type A:

$$
\begin{array}{l}
\mathit{toList} :: X \to [A] \\
\mathit{toList}\ x = \mathit{head}\ x : \mathit{toList}\ (\mathit{tail}\ x)
\end{array}
$$

In the other direction, if we want to build a stream we only have one constructor: *cons* but no "base case". In Haskell, thanks to laziness, we can still define streams directly using *cons* and recursion As an example, we can construct a constant stream as follows:

$$
\begin{array}{l}
\mathit{constS} :: A \to X \\
\mathit{constS}\ a = ca \\
\quad \textbf{where}\ ca = \mathit{cons}\ a\ ca
\end{array}
$$

Instead of specifying a stream in terms of how to contruct it, we could describe it in terms of how to take it apart; by specifying its *head* and *tail*. In the constant stream example we would get something like:

$$
\begin{array}{l}
\mathit{head}\ (\mathit{constS}\ a) = a \\
\mathit{tail}\ (\mathit{constS}\ a) = \mathit{constS}\ a
\end{array}
$$

but this syntax is not supported in Haskell.

The last part of the API are a few laws we expect to hold. The first law simply states that if we first take a stream s apart into its head and its tail, we can get back to the original stream by *cons*ing them back together. The second and third are variant on this theme, and together the three laws specify how the three operations interact.

An unusual stream (Credits: [Pavlovic and Escardó, 1998])

Now consider $X = \mathbb{R} \to \mathbb{R}$, and $A = \mathbb{R}$ with the following definitions:

$$
\begin{array}{lll}
\textbf{type}\ X = \mathbb{R} \to \mathbb{R} & & \\
\textbf{type}\ A = \mathbb{R} & & \\
\mathit{deriv} :: X \to X & & \\
\mathit{integ} :: X \to X & & \\
\mathit{head}\ f \quad = f\ 0 & \text{-- value of } f \text{ at } 0 & \\
\mathit{tail}\ \ f \quad = \mathit{deriv}\ f & \text{-- derivative of } f & \\
\mathit{cons}\ a\ f = \mathit{const}\ a + \mathit{integ}\ f & \text{-- start at } a\text{, integrate } f \text{ from } 0 &
\end{array}
$$

Then the first law becomes

$$
\begin{aligned}
&\mathit{law1c}\ f = \\
&\quad f \equiv \mathit{cons}\ (\mathit{head}\ f)\ (\mathit{tail}\ f) \\
&\qquad \equiv (\mathit{head}\ f) + \mathit{integ}\ (\mathit{tail}\ f) \\
&\qquad \equiv f\ 0 + \mathit{integ}\ (\mathit{deriv}\ f)
\end{aligned}
$$

or, in traditional notation:

$$f(x) = f(0) + \int_0^x f'(t)\mathrm{d}t$$

which we recognize as the fundamental law of calculus! There is much more to discover in this direction and we present some of it in the next few chapters.

For the curious. Here are the other two stream laws, in case you wondered.

$$
\begin{aligned}
&\mathit{law2c}\ a\ f = \\
&\quad f \equiv \mathit{tail}\ (\mathit{cons}\ a\ f) \\
&\qquad \equiv \mathit{deriv}\ (\mathit{const}\ a + \mathit{integ}\ f) \\
&\qquad \equiv \mathit{deriv}\ (\mathit{integ}\ f)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{law3c}\ a\ f = \\
&\quad a \equiv \mathit{head}\ (\mathit{cons}\ a\ f) \\
&\quad a \equiv \mathit{head}\ (\mathit{const}\ a + \mathit{integ}\ f) \\
&\quad a \equiv (\mathit{const}\ a + \mathit{integ}\ f)\ 0 \\
&\quad a \equiv a + (\mathit{integ}\ f)\ 0 \\
&\quad 0 \equiv \mathit{integ}\ f\ 0
\end{aligned}
$$

4.5 Exercises

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

Exercise 4.2. Complete the instance declarations for $(\mathbb{R}, \mathbb{R})$, deriving them from the homomorphism requirement for *apply* (in section 4.2.3).

Exercise 4.3. We now have three different ways of computing the derivative of a function such as $f\ x = \mathit{sin}\ x + \mathit{exp}\ (\mathit{exp}\ x)$ at a given point, say $x = \pi$.

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

b. Find an expression of type $\mathit{FD}\ \mathbb{R}$ and use *apply*.

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

Do you get the same result?

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

In exercise 1.3 we looked at the datatype $\mathit{SR}\ v$ for the language of semiring expressions. We will now use some of the concepts discussed in this chapter to expand on this language.

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

b. Define a *SemiRing* instance for the datatype $\mathit{SR}\ v$ that you defined in exercise 1.3.

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

d. Specialise the evaluator that you defined in exercise 1.3 to the two *SemiRing* instances defined above. Take three semiring expressions of type *SR String*, give the appropriate assignments and compute the results of evaluating, in each case, the three expressions.

Exercise 4.5. *From exam 2016-03-15*

In exercise 1.4, we looked a datatype for the language of lattice expressions. We will now use some of the concepts discussed in this chapter to expand on this language.

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

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

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

d. Specialise the evaluator you defined in exercise 1.4.2 to the two *Lattice* instances defined above. Take three lattice expressions, give the appropriate assignments and compute the results of evaluating, in each case, the three expressions.

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

In exercise 1.5, we looked a datatype for the language of abelian monoid expressions. We will now use some of the concepts discussed in this chapter to expand on this language.

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

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

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

d. Specialise the evaluator that you defined in exercise 1.5.2 to the *AbMonoid* instance defined above. Take three 'AbMonoidExp' expressions, give the appropriate assignments and compute the results of evaluating the three expressions.

Exercise 4.7. (Closely related to exam question)

A *ring* is a set A together with two constants (or nullary operations), 0 and 1, one unary operation, *negate*, and two binary operations, $+$ and $*$, such that

a. 0 is the neutral element of $+$

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

b. $+$ is associative

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

c. *negate* inverts elements with respect to addition

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

d. $+$ is commutative

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

e. 1 is the unit of $*$

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

f. $*$ is associative

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

g. $*$ distributes over $+$

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

Remarks:

- a. and b. say that $(A, 0, +)$ is a monoid
- a–c. say that $(A, 0, +, \mathit{negate})$ is a group
- a–d. say that $(A, 0, +, \mathit{negate})$ is a commutative group
- e. and f. say that $(A, 1, *)$ is a monoid

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

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

iii Find two other instances of the *Ring* class.

iv Define a general evaluator for *Ring* expressions on the basis of a given assignment function.

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

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

Recall the type of expressions

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

and consider the function

```
f :: ℝ → ℝ
f x = exp (sin x) + x
```

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

b. Implement a function *deriv2* such that, for any $f : Fractional\ a \Rightarrow a \to a$ constructed with the grammar of *FunExp* and any x in the domain of f, we have that $deriv2\ f\ x$ computes the second derivative of f at x. Use the function $derive :: FunExp \to FunExp$ from the lectures ($eval\ (derive\ e)$ is the derivative of $eval\ e$). What instance declarations do you need?

   The type of $deriv2\ f$ should be $Fractional\ a \Rightarrow a \to a$.

Exercise 4.9. Based on the lecture notes, complete all the instance and datatype declarations and definitions in the files FunNumInst.lhs, FunExp.lhs, Derive.lhs, EvalD.lhs, and ShallowD.lhs.

Exercise 4.10. Write a function

$$simplify :: FunExp \to FunExp$$

to simplify the expression resulted from *derive*. For example, the following tests should work:

$$\begin{aligned}
&simplify\ (Const\ 0 :*: Exp\ Id) &&== Const\ 0\\
&simplify\ (Const\ 0 :+: Exp\ Id) &&== Exp\ Id\\
&simplify\ (Const\ 2 :*: Const\ 1) &&== Const\ 2\\
&simplify\ (derive\ (Id :*: Id)) &&== Const\ 2 :*: Id
\end{aligned}$$

As a motivating example, note that $derive\ (Id :*: Id)$ evalutes to $(Const\ 1.0 :*: Id) :+: (Id :*: Const\ 1.0)$ without *simplify*, and that the second derivative looks even worse.