

# 1 A DSL for arithmetic expressions and complex numbers

This chapter is partly based on the paper [Ionescu and Jansson, 2016] from the International Workshop on Trends in Functional Programming in Education 2015. We will implement certain concepts in the functional programming language Haskell and the code for this lecture is placed in a module called *DSLsofMath.W01* that starts here:

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

## 1.1 Intro: Pitfalls with traditional mathematical notation

**A function or the value at a point?** Mathematical texts often talk about “the function  $f(x)$ ” when “the function  $f$ ” would be more clear. Otherwise there is a risk of confusion between  $f(x)$  as a function and  $f(x)$  as the value you get from applying the function  $f$  to the value bound to the name  $x$ .

Examples: let  $f(x) = x + 1$  and let  $t = 5 * f(2)$ . Then it is clear that the value of  $t$  is the constant 15. But if we let  $s = 5 * f(x)$  it is not clear if  $s$  should be seen as a constant or as a function of  $x$ .

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

**Scoping** The syntax and scoping rules for the integral sign are rarely explicitly mentioned, but looking at it from a software perspective can help. If we start from a simple example, like  $\int_1^2 x^2 dx$ , it is relatively clear: the integral sign takes two real numbers as limits and then a certain notation for a function, or expression, to be integrated. Comparing the part after the integral sign to the syntax of a function definition  $f(x) = x^2$  reveals a rather odd rule: instead of *starting* with declaring the variable  $x$ , the integral syntax *ends* with the variable name, and also uses the letter “d”. (There are historical explanations for this notation, and it is motivated by computation rules in the differential calculus, but we will not go there now.) It seems like the scope of the variable “bound” by  $d$  is from the integral sign to the final  $dx$ , but does it also extend to the limits? The answer is no, as we can see from a slightly extended example:

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

The variable  $x$  bound on the left is independent of the variable  $x$  “bound under the integral sign”. Mathematics text books usually avoid the risk of confusion by (silently) renaming variables when needed, but we believe this renaming is a sufficiently important operation to be more explicitly mentioned.

**Variable names as type hints** In mathematical texts there are often conventions about the names used for variables of certain types. Typical examples include  $i, j, k$  for natural numbers or integers,  $x, y$  for real numbers and  $z, w$  for complex numbers.

The absence of explicit types in mathematical texts can sometimes lead to confusing formulations. For example, a standard text on differential equations by Edwards, Penney, and Calvis [2008] contains at page 266 the following remark:

The differentiation operator  $D$  can be viewed as a transformation which, when applied to the function  $f(t)$ , yields the new function  $D\{f(t)\} = f'(t)$ . The Laplace transformation  $\mathcal{L}$  involves the operation of integration and yields the new function  $\mathcal{L}\{f(t)\} = F(s)$  of a new independent variable  $s$ .

This is meant to introduce a distinction between “operators”, such as differentiation, which take functions to functions of the same type, and “transforms”, such as the Laplace transform, which take functions to functions of a new type. To the logician or the computer scientist, the way of phrasing this difference in the quoted text sounds strange: surely the *name* of the independent variable does not matter: the Laplace transformation could very well return a function of the “old” variable  $t$ . We can understand that the name of the variable is used to carry semantic meaning about its type (this is also common in functional programming, for example with the conventional use of a plural “s” suffix, as in the name *xs*, to denote a list of values.). Moreover, by using this (implicit!) convention, it is easier to deal with cases such as that of the Hartley transform (a close relative of the Fourier transform), which does not change the type of the input function, but rather the *interpretation* of that type. We prefer to always give explicit typings rather than relying on syntactical conventions, and to use type synonyms for the case in which we have different interpretations of the same type. In the example of the Laplace transformation, this leads to

```
type T = Real
type S = ℂ
ℒ : (T → ℂ) → (S → ℂ)
```

## 1.2 Types of data

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

### 1.2.1 type / newtype / data

There are three keywords in Haskell involved in naming types: **type**, **newtype**, and **data**.

**type – abbreviating type expressions** The **type** keyword is used to create a type synonym - just another name for a type expression.

```
type Heltal = Integer
type Foo = (Maybe [String], [[Heltal]])
type BinOp = Heltal → Heltal → Heltal
type Env v s = [(v, s)]
```

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

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

```
newtype Age = Ag Int -- Age in years
newtype Shoe = Sh Int -- Shoe size (EU)
```

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

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

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

**The keyword data – for syntax trees** The simplest form of a recursive datatype is the unary notation for natural numbers:

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

This declaration introduces

- a new type  $N$  for unary natural numbers,
- a constructor  $Z :: N$  to represent zero, and
- a constructor  $S :: N \rightarrow N$  to represent the successor.

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

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

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

$$\mathbf{data} \ Maybe \ a = Nothing \mid Just \ a$$

This declaration introduces

- a new type  $Maybe \ a$  for every type  $a$ ,
- a constructor  $Nothing :: Maybe \ a$  to represent “no value”, and
- a constructor  $Just :: a \rightarrow Maybe \ a$  to represent “just a value”.

A maybe type is often used when a function may, or may not, return a value.

Two other examples of, often used, parameterised types are  $(a, b)$  for the type of pairs (a product type) and  $Either \ a \ b$  for either an  $a$  or a  $b$  (a sum type).

$$\mathbf{data} \ Either \ p \ q = Left \ p \mid Right \ q$$

### 1.2.2 *Env* and variable lookup.

The type synonym

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

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

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

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

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

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

```
lookup :: (Eq a) => a -> [(a, b)] -> Maybe b
```

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

## 1.3 A syntax for simple arithmetical expressions

```
data AE = V String | P AE AE | T AE AE
```

This declaration introduces

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

Example values:  $x = V\ "x"$ ,  $e_1 = P\ x\ x$ ,  $e_2 = T\ e_1\ e_1$

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

```
data AE' = V' String | AE' :+ AE' | AE' :* AE'
```

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

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

```
data AE' v = V' v | AE' v :+ AE' v | AE' v :* AE' v
```

The purpose of the parameter  $v$  here is to enable a free choice of type for the variables (be it *String* or *Int* or something else).

The careful reader will note that the same Haskell module cannot contain both these definitions of  $AE'$ . This is because the name of the type and the names of the constructors are clashing. The typical ways around this are: define the types in different modules, or rename one of them (often by adding primes as in  $AE'$ ). In this book we often take the liberty of presenting more than one version of a datatype without changing the names, to avoid multiple modules or too many primes.

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

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

```

evalAE :: Env String Integer → (AE → Maybe Integer)
evalAE env (V x)      = evalEnv env x
evalAE env (P e1 e2) = mayP (evalAE env e1) (evalAE env e2)
evalAE env (T e1 e2) = mayT (evalAE env e1) (evalAE env e2)

mayP :: Maybe Integer → Maybe Integer → Maybe Integer
mayP (Just a) (Just b) = Just (a + b)
mayP _ _              = Nothing

mayT :: Maybe Integer → Maybe Integer → Maybe Integer
mayT (Just a) (Just b) = Just (a * b)
mayT _ _              = Nothing

evalAE' :: (Eq v, Num sem) ⇒ (Env v sem) → (AE' v → Maybe sem)
evalAE' env (V' x)      = evalEnv env x
evalAE' env (e1 ÷ e2) = liftM (+) (evalAE' env e1) (evalAE' env e2)
evalAE' env (e1 * e2) = liftM (*) (evalAE' env e1) (evalAE' env e2)

liftM :: (a → b → c) → (Maybe a → Maybe b → Maybe c)
liftM op (Just a) (Just b) = Just (op a b)
liftM _ op _              = Nothing

```

## 1.4 A case study: complex numbers

We will start by an analytic reading of the introduction of complex numbers in Adams and Essex [2010]. We choose a simple domain to allow the reader to concentrate on the essential elements of our approach without the distraction of potentially unfamiliar mathematical concepts. For this section, we bracket our previous knowledge and approach the text as we would a completely new domain, even if that leads to a somewhat exaggerated attention to detail.

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

We begin by defining the symbol  $i$ , called **the imaginary unit**, to have the property

$$i^2 = -1$$

Thus, we could also call  $i$  the square root of  $-1$  and denote it  $\sqrt{-1}$ . Of course,  $i$  is not a real number; no real number has a negative square.

At this stage, it is not clear what the type of  $i$  is meant to be, we only know that  $i$  is not a real number. Moreover, we do not know what operations are possible on  $i$ , only that  $i^2$  is another name for  $-1$  (but it is not obvious that, say  $i * i$  is related in any way with  $i^2$ , since the operations of multiplication and squaring have only been introduced so far for numerical types such as  $\mathbb{N}$  or  $\mathbb{R}$ , and not for symbols).

For the moment, we introduce a type for the value  $i$ , and, since we know nothing about other values, we make  $i$  the only member of this type:

```
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

$$a + bi \quad \text{or} \quad a + ib,$$

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  $\mathbb{R}$   $\mathbb{R}$  ImagUnits
              | CPlus2  $\mathbb{R}$  ImagUnits  $\mathbb{R}$ 
```

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  $\mathbb{R}$  = Double
```

The text continues with examples:

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

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

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

The second example is somewhat problematic: it does not seem to be of the form  $a + bi$ . Given that the last two examples seem to introduce shorthand for various complex numbers, let us assume that this one does as well, and that  $a - bi$  can be understood as an abbreviation of  $a + (-b) i$ . With this provision, in our notation the examples are written as in Table 1.

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

$$\begin{aligned} toComplex &:: \mathbb{R} \rightarrow ComplexA \\ toComplex\ x &= CPlus_1\ x\ 0\ I \end{aligned}$$

Again, at this stage there are many open questions. For example, we can assume that  $i\ 1$  stands for the complex number  $CPlus_2\ 0\ I\ 1$ , but what about  $i$  by itself? If juxtaposition is meant to denote some sort of multiplication, then perhaps  $1$  can be considered as a unit, in which case we would have that  $i$  abbreviates  $i\ 1$  and therefore  $CPlus_2\ 0\ I\ 1$ . But what about, say,  $2\ i$ ? Abbreviations with  $i$  have only been introduced for the  $ib$  form, and not for the  $bi$  one!

The text then continues with a parenthetical remark which helps us dispel these doubts:

(We will normally use  $a + bi$  unless  $b$  is a complicated expression, in which case we will write  $a + ib$  instead. Either form is acceptable.)

This remark suggests strongly that the two syntactic forms are meant to denote the same elements, since otherwise it would be strange to say “either form is acceptable”. After all, they are acceptable by definition.

Given that  $a + ib$  is only “syntactic sugar” for  $a + bi$ , we can simplify our representation for the abstract syntax, eliminating one of the constructors:

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

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

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

(The renaming of the constructor to *CPlusC* serves as a guard against the case we have suppressed potentially semantically relevant syntax.)

We read further:

It is often convenient to represent a complex number by a single letter;  $w$  and  $z$  are frequently used for this purpose. If  $a$ ,  $b$ ,  $x$ , and  $y$  are real numbers, and  $w = a + bi$  and  $z = x + yi$ , then we can refer to the complex numbers  $w$  and  $z$ . Note that  $w = z$  if and only if  $a = x$  and  $b = y$ .

First, let us notice that we are given an important semantic information: to check equality for complex numbers, it is enough to check equality of the components (the arguments to the constructor *CPlusC*). (Another way of saying this is that *CPlusC* is injective.) The equality on complex numbers is what we would obtain in Haskell by using **deriving Eq**.

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

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

The point of the somewhat confusing discussion of using “letters” to stand for complex numbers is to introduce a substitute for *pattern matching*, as in the following definition:

**Definition:** If  $z = x + yi$  is a complex number (where  $x$  and  $y$  are real), we call  $x$  the **real part** of  $z$  and denote it  $Re\ (z)$ . We call  $y$  the **imaginary part** of  $z$  and denote it  $Im\ (z)$ :

$$\begin{aligned} Re\ (z) &= Re\ (x + yi) = x \\ Im\ (z) &= Im\ (x + yi) = y \end{aligned}$$

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

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

a potential source of confusion being that the symbol  $z$  introduced by the *as-pattern* is not actually used on the right-hand side of the equations.

The use of *as-patterns* such as “ $z = x + yi$ ” is repeated throughout the text, for example in the definition of the algebraic operations on complex numbers:

### The sum and difference of complex numbers

If  $w = a + bi$  and  $z = x + yi$ , where  $a$ ,  $b$ ,  $x$ , and  $y$  are real numbers, then

$$\begin{aligned} w + z &= (a + x) + (b + y)\ i \\ w - z &= (a - x) + (b - y)\ i \end{aligned}$$

With the introduction of algebraic operations, the language of complex numbers becomes much richer. We can describe these operations in a *shallow embedding* in terms of the concrete datatype *ComplexD*, for example:

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

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

Exercises:

- implement  $(*)$  for *ComplexD*

At this point we can sum up the “evolution” of the datatypes introduced so far. Starting from *ComplexA*, the type has evolved by successive refinements through *ComplexB*, *ComplexC*, ending up in *ComplexD* (see Fig. 2). (We can also make a parameterised version of *ComplexD*, by noting that the definitions for complex number operations work fine for a range of underlying numeric types. The operations for *ComplexSem* are defined in module *CSem*, available in appendix A.)



```

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

```

Figure 2: Complex number datatype refinement (semantics).

## 1.5 A syntax for (complex) arithmetical expressions

So far we have tried to find a datatype to represent the intended *semantics* of complex numbers. That approach is called “shallow embedding”. Now we turn to the *syntax* instead (“deep embedding”).

We want a datatype *ComplexE* for the abstract syntax tree of expressions. The syntactic expressions can later be evaluated to semantic values:

$$evalE :: ComplexE \rightarrow ComplexD$$

The datatype *ComplexE* should collect ways of building syntactic expression representing complex numbers and we have so far seen the symbol *i*, an embedding from  $\mathbb{R}$ , plus and times. We make these four *constructors* in one recursive datatype as follows:

```

data ComplexE = ImagUnit
                | ToComplex  $\mathbb{R}$ 
                | Plus   ComplexE ComplexE
                | Times ComplexE ComplexE
deriving (Eq, Show)

```

Note that, in *ComplexA* above, we also had a constructor for “plus”, but it was another “plus”. They are distinguished by type: *CPlus*<sub>1</sub> took (basically) two real numbers, while *Plus* here takes two complex numbers as arguments.

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

For example, when implementing the *evalE* (*Plus* *c*<sub>1</sub> *c*<sub>2</sub>) case, you can assume that you already know the values *s*<sub>1</sub>, *s*<sub>2</sub> :: *ComplexD* corresponding to the subtrees *c*<sub>1</sub> and *c*<sub>2</sub>. The only thing left is to add them up componentwise and we can assume there is a function (+) :: *ComplexD* → *ComplexD* taking care of this step. Continuing in this direction (by “wishful thinking”) we arrive at the following implementation.

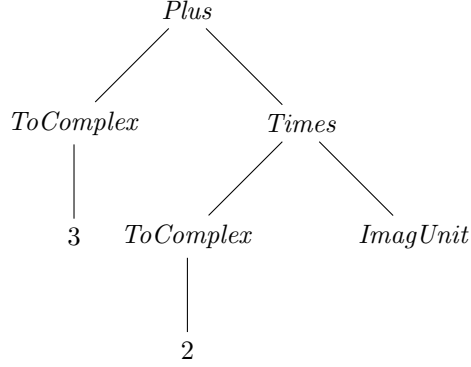
```

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

```

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

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



This is *testE1* as an abstract syntax tree:

## 1.6 Laws, properties and testing

There are certain laws we would like to hold for operations on complex numbers. To specify these laws, in a way which can be easily testable in Haskell, we use functions to *Bool* (also called *predicates* or *properties*). The intended meaning of such a boolean function is “forall inputs, this should return *True*”. This idea is at the core of *property based testing* (pioneered by Claessen and Hughes [2000]) and conveniently available in the library QuickCheck.

The simplest law is perhaps  $i^2 = -1$  from the start of the lecture,

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

and that *fromCD* is an embedding:

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

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

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

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

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

Note that *propAssocA* is a higher order function: it takes a function (a binary operator) as its first parameter. It is also polymorphic: it works for many different types *a* (all types which have an *==* operator).

Thus we can specialise it to *Plus*, *Times* and other binary operators. In Haskell there is a type class *Num* for different types of “numbers” (with operations (+), (\*), etc.). We can try out *propAssocA* for a few of them.

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

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

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

For completeness: this is the answer:

```
(2.3333333333333335      -- Notice the five at the end
, 2.3333333333333333,    -- which is not present here.
, 4.440892098500626e-16 -- The difference
, False)
```

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

```
data ComplexSyn r = FromCartesian r r
    | ComplexSyn r :+: ComplexSyn r
    | ComplexSyn r :*: ComplexSyn r

toComplexSyn :: Num a ⇒ a → ComplexSyn a
toComplexSyn x = FromCartesian x 0

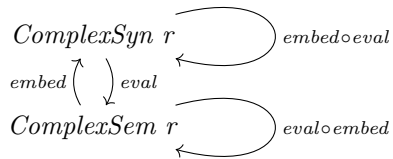
-- From CSem in appendix A: newtype ComplexSem r = CS (r, r) deriving Eq
evalCSyn :: Num r ⇒ ComplexSyn r → CSem.ComplexSem r
evalCSyn (FromCartesian x y) = CS (x, y)
evalCSyn (l :+: r) = evalCSyn l CSem+. evalCSyn r
evalCSyn (l :*: r) = evalCSyn l CSem*. evalCSyn r
instance Num a ⇒ Num (ComplexSyn a)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

fromIntegerCS :: Num r ⇒ Integer → ComplexSyn r
fromIntegerCS = toComplexSyn ∘ fromInteger
```

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

```
embed :: CSem.ComplexSem r -> ComplexSyn r
embed (CS (x, y)) = FromCartesian x y
```

The embedding should satisfy a round-trip property:  $eval (embed s) == s$  for all semantic complex numbers  $s$ . Here is a diagram showing how the types and the functions fit together



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

**More about laws** Some laws appear over and over again in different mathematical contexts. Binary operators are often as associative or commutative, and sometimes one operator distributes over another. We will work more formally with logic in chapter 2 but we introduce a few definitions already here:

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

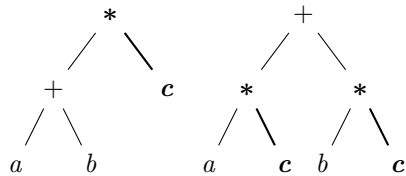
*Commutative*  $(+)$   $= \forall a, b. a + b = b + a$

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

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

We saw implementations of some of these laws as *propAssocA* and *propDistTimesPlus* earlier, and learnt that the underlying set matters:  $(+)$  for  $\mathbb{R}$  has some properties, but  $(+)$  for *Double* has other. When implementing, approximation is often necessary, but makes many laws false. Thus, we should attempt to do it late, and if possible, leave a parameter to make the degree of approximation tunable (*Int*, *Integer*, *Float*, *Double*,  $\mathbb{Q}$ , syntax trees, etc.).

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



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

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

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

## 1.7 More about functions

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

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

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

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

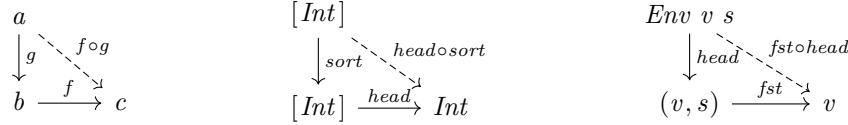


Figure 3: Function composition diagrams: in general, and two examples

In Haskell we get the following type:

$$(\circ) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$$

which may take a while to get used to.

### 1.7.1 Notation and abstract syntax for (infinite) sequences

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

Common math book notation:  $\{a_i\}_{i=0}^\infty$  or just  $\{a_i\}$  and (not always) an indication of the type  $X$  of the  $a_i$ . Note that the  $a$  at the center of this notation actually carries all of the information: an infinite family of values  $a_i : X$ . If we interpret “subscript” as function application we can see that  $a : \mathbb{N} \rightarrow X$  is a useful typing of a sequence. Some examples:

```

type  $\mathbb{N}$       = Natural  -- imported from Numeric.Natural
type  $\mathbb{Q}$       = Ratio  $\mathbb{N}$  -- imported from Data.Ratio
type Seq  $a$  =  $\mathbb{N} \rightarrow a$ 
idSeq :: Seq  $\mathbb{N}$ 
idSeq  $i$  =  $i$               --  $\{0, 1, 2, 3, \dots\}$ 
invSeq :: Seq  $\mathbb{Q}$ 
invSeq  $i$  =  $1 \div (1 + i)$  --  $\{\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \dots\}$ 
pow2 :: Num  $r \Rightarrow$  Seq  $r$ 
pow2 =  $(2^{\wedge})$             --  $\{1, 2, 4, 8, \dots\}$ 
conSeq ::  $a \rightarrow$  Seq  $a$ 
conSeq  $c$   $i$  =  $c$          --  $\{c, c, c, c, \dots\}$ 

```

What operations can be performed on sequences? We have seen the first one: given a value  $c$  we can generate a constant sequence with *conSeq*  $c$ . We can also add sequences componentwise (also called “pointwise”):

```

addSeq :: Num  $a \Rightarrow$  Seq  $a \rightarrow$  Seq  $a \rightarrow$  Seq  $a$ 
addSeq  $f$   $g$   $i$  =  $f\ i + g\ i$ 

```

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

$liftSeq2 :: (a \rightarrow b \rightarrow c) \rightarrow Seq\ a \rightarrow Seq\ b \rightarrow Seq\ c$   
 $liftSeq2\ op\ f\ g\ i = op\ (f\ i)\ (g\ i) \quad -- \{op\ (f\ 0)\ (g\ 0), op\ (f\ 1)\ (g\ 1), \dots\}$

Similarly we can lift unary operations, and “nullary” operations:

$liftSeq1 :: (a \rightarrow b) \rightarrow Seq\ a \rightarrow Seq\ b$   
 $liftSeq1\ h\ f\ i = h\ (f\ i) \quad -- \{h\ (f\ 0), h\ (f\ 1), h\ (f\ 2), \dots\}$   
 $liftSeq0 :: a \rightarrow Seq\ a$   
 $liftSeq0\ c\ i = c$

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

Another common mathematical operator on sequences is the limit. We will get back to limits in later chapters (2.11, 2.13), but here we just analyse the notation and typing. This definition is slightly adapted from Wikipedia (2017-11-08):

We call  $L$  the limit of the sequence  $\{x_n\}$  if the following condition holds: For each real number  $\epsilon > 0$ , there exists a natural number  $N$  such that, for every natural number  $n \geq N$ , we have  $|x_n - L| < \epsilon$ .

If so, we say that the sequence converges to  $L$  and write

$$L = \lim_{i \rightarrow \infty} x_i$$

There are (at least) two things to note here. First, with this syntax, the  $\lim_{i \rightarrow \infty} x_i$  expression form binds  $i$  in the expression  $x_i$ . We could just as well say that  $lim$  takes a function  $x :: \mathbb{N} \rightarrow X$  as its only argument. Second, an arbitrary  $x$ , may or may not have a limit. Thus the customary use of  $L =$  is a bit of abuse of notation, because the right hand side may not be well defined. One way to capture that is to give  $lim$  the type  $(\mathbb{N} \rightarrow X) \rightarrow Maybe\ X$ . Then  $L = \lim_{i \rightarrow \infty} x_i$  would mean *Just*  $L = lim\ x$ . We will return to limits and their proofs in Sec. 2.12 after we have reviewed some logic. Here we just define one more common operation: the sum of a sequence (like  $\sigma = \sum_{i=0}^{\infty} 1/i!$ ). Just as not all sequences have a limit, not all have a sum either. But for every sequence we can define a new sequence of partial sums:

$sums :: Num\ a \Rightarrow Seq\ a \rightarrow Seq\ a$   
 $sums = scan\ 0\ (+)$   
 $scan :: a \rightarrow (a \rightarrow a \rightarrow a) \rightarrow Seq\ a \rightarrow Seq\ a$   
 $scan\ z\ (+)\ f = s$   
**where**  $s\ 0 = z$   
 $s\ i = s\ (i - 1) + f\ i$

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

We will also return to limits in Sec. 3.3 about derivatives where we explore variants of the classical definition

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

To sum up this subsection, we have defined a small Domain Specific Language (DSL) for infinite sequences by defining a type  $(Seq\ a)$ , some operations (*conSeq*, *addSeq*, *fmap*, *sums*, ...) and some “run functions” or predicates (like *lim* and *sum*).

## 1.8 Exercises: complex numbers and DSLs

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

```
data Exp = Con Integer
        | Exp 'Plus'  Exp
        | Exp 'Minus' Exp
        | Exp 'Times' Exp
deriving (Eq, Show)
```

(Note the use of “backticks” around *Plus* etc. which makes it possible to use a name as an infix operator.)

1. Write the following expressions in Haskell, using the *Exp* data type:
  - (a)  $a_1 = 2 + 2$
  - (b)  $a_2 = a_1 + 7 * 9$
  - (c)  $a_3 = 8 * (2 + 11) - (3 + 7) * (a_1 + a_2)$
2. Create a function  $eval :: Exp \rightarrow Integer$  that takes a value of the *Exp* data type and returns the corresponding number (for instance,  $eval ((Con\ 3)\ 'Plus'\ (Con\ 3)) = 6$ ). Try it on the expressions from the first part, and verify that it works as expected.
3. Consider the following expression:

$$c_1 = (x - 15) * (y + 12) * z$$

**where**  $x = 5$   
 $y = 8$   
 $z = 13$

In order to represent this with our *Exp* data type, we are going to have to make some modifications:

- (a) Update the *Exp* data type with a new constructor *Var String* that allows variables with strings as names to be represented. Use the updated *Exp* to write an expression for  $c_1$  in Haskell.
- (b) Create a function  $varVal :: String \rightarrow Integer$  that takes a variable name, and returns the value of that variable. For now, the function just needs to be defined for the variables in the expression above, i.e.  $varVal\ "x"$  should return 5,  $varVal\ "y"$  should return 8, and  $varVal\ "z"$  should return 13.
- (c) Update the *eval* function so that it supports the new *Var* constructor, and use it get a numeric value of the expression  $c_1$ .

**Exercise 1.2.** We will now look at a slightly more generalized version of the *Exp* type from the previous exercise:

```
data E2 a = Con a
        | Var String
        | E2 a 'Plus'  E2 a
        | E2 a 'Minus' E2 a
        | E2 a 'Times' E2 a
deriving (Eq, Show)
```

The type has now been parametrized, so that it is no longer limited to representing expressions with integers, but can instead represent expressions with any type. For instance, we could have an *E2 Double* to represent expressions with doubles, or an *E2 ComplexD* to represent expressions with complex numbers.

1. Write the following expressions in Haskell, using the new *E2* data type.

- (a)  $a_1 = 2.0 + a$
- (b)  $a_2 = 5.3 + b * c$
- (c)  $a_3 = a * (b + c) - (d + e) * (f + a)$

2. In order to evaluate these expressions, we will need a way of translating a variable name into the value. The following table shows the value of each variable in the expressions above:

Name	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>
Value	1.5	4.8	2.4	7.4	5.8	1.7

In Haskell, we can represent this table using a value of type *Env String a* = [(*String*, *a*)], which is a list of pairs of variable names and values, where each entry in the list corresponds to a row in the table.

- (a) Express the table above in Haskell by creating *vars :: Env String Double*.
- (b) Create a function *varVal :: Env a → String → a* that returns the value of a variable, given an *Env* and a variable name. For instance, *varVal vars "d"* should return 7.4
- (c) Create a function *eval :: Num a ⇒ Env a → E2 → a* that takes a value of the new *E2* data type and returns the corresponding number. For instance, *eval vars ((Con 2) 'Plus' (Var "a"))* = 3.5. Try it on the expressions from the first part, and verify that it works as expected.

**Exercise 1.3.** *From exam 2017-08-22*

A semiring is a set *R* equipped with two binary operations  $+$  and  $\cdot$ , called addition and multiplication, such that:

- $(R, +, 0)$  is a commutative monoid with identity element 0:

$$\begin{aligned} (a + b) + c &= a + (b + c) \\ 0 + a &= a + 0 = a \\ a + b &= b + a \end{aligned}$$

- $(R, \cdot, 1)$  is a monoid with identity element 1:

$$\begin{aligned} (a \cdot b) \cdot c &= a \cdot (b \cdot c) \\ 1 \cdot a &= a \cdot 1 = a \end{aligned}$$

- Multiplication left and right distributes over  $(R, +, 0)$ :

$$\begin{aligned} a \cdot (b + c) &= (a \cdot b) + (a \cdot c) \\ (a + b) \cdot c &= (a \cdot c) + (b \cdot c) \\ a \cdot 0 &= 0 \cdot a = 0 \end{aligned}$$

1. Define a datatype *SR v* for the language of semiring expressions (with variables of type *v*). (These are expressions formed from applying the semiring operations to the appropriate number of arguments, e.g., all the left hand sides and right hand sides of the above equations.)
2. Give a type signature for, and define, a general evaluator for *SR v* expressions on the basis of an assignment function.



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

A *lattice* is a set  $L$  together with two operations  $\vee$  and  $\wedge$  (usually pronounced “sup” and “inf”) such that

- $\vee$  and  $\wedge$  are associative:

$$\begin{aligned}\forall x, y, z \in L. \quad (x \vee y) \vee z &= x \vee (y \vee z) \\ \forall x, y, z \in L. \quad (x \wedge y) \wedge z &= x \wedge (y \wedge z)\end{aligned}$$

- $\vee$  and  $\wedge$  are commutative:

$$\begin{aligned}\forall x, y \in L. \quad x \vee y &= y \vee x \\ \forall x, y \in L. \quad x \wedge y &= y \wedge x\end{aligned}$$

- $\vee$  and  $\wedge$  satisfy the *absorption laws*:

$$\begin{aligned}\forall x, y \in L. \quad x \vee (x \wedge y) &= x \\ \forall x, y \in L. \quad x \wedge (x \vee y) &= x\end{aligned}$$

1. Define a datatype for the language of lattice expressions.
2. Define a general evaluator for *Lattice* expressions on the basis of an assignment function.

**Exercise 1.5.** *From exam 2016-08-23*

An *abelian monoid* is a set  $M$  together with a constant (nullary operation)  $0 \in M$  and a binary operation  $\oplus : M \rightarrow M \rightarrow M$  such that:

- $0$  is a unit of  $\oplus$

$$\forall x \in M. \quad 0 \oplus x = x \oplus 0 = x$$

- $\oplus$  is associative

$$\forall x, y, z \in M. \quad x \oplus (y \oplus z) = (x \oplus y) \oplus z$$

- $\oplus$  is commutative

$$\forall x, y \in M. \quad x \oplus y = y \oplus x$$

1. Define a datatype *AbMonoidExp* for the language of abelian monoid expressions. (These are expressions formed from applying the monoid operations to the appropriate number of arguments, e.g., all the left hand sides and right hand sides of the above equations.)
2. Define a general evaluator for *AbMonoidExp* expressions on the basis of an assignment function.

**Exercise 1.6.** Read the full chapter and complete the definition of the instance for *Num* for the datatype *ComplexSyn*. Also add a constructor for variables to enable writing expressions like `(Var "z")` `toComplex 1`.

**Exercise 1.7.** Read the next few pages of Appendix I (in [Adams and Essex, 2010]) defining the polar view of Complex Numbers and try to implement complex numbers again, this time based on magnitude and phase for the semantics.

**Exercise 1.8.** Implement a simplifier  $\text{simp} :: \text{ComplexSyn } r \rightarrow \text{ComplexSyn } r$  that handles a few cases like  $0 * x = 0$ ,  $1 * x = x$ ,  $(a + b) * c = a * c + b * c$ , ... What class context do you need to add to the type of  $\text{simp}$ ?

**Exercise 1.9.** Functions and pairs (the “tupling transform”). From one function  $f :: a \rightarrow (b, c)$  returning a pair, you can always make a pair of two functions  $pf :: (a \rightarrow b, a \rightarrow c)$ . Implement this transform:

$$f2p :: (a \rightarrow (b, c)) \rightarrow (a \rightarrow b, a \rightarrow c)$$

Also implement the opposite transform:

$$p2f :: (a \rightarrow b, a \rightarrow c) \rightarrow (a \rightarrow (b, c))$$

This kind of transformation is often useful, and it works also for  $n$ -tuples.

**Exercise 1.10.** There is also a “dual” to the tupling transform: to show this, implement these functions:

$$\begin{aligned} s2p &:: (\text{Either } b \ c \rightarrow a) \rightarrow (b \rightarrow a, c \rightarrow a) \\ p2s &:: (b \rightarrow a, c \rightarrow a) \rightarrow (\text{Either } b \ c \rightarrow a) \end{aligned}$$

**Exercise 1.11.** Counting values. Now assume we have  $f2p$ ,  $s2f$ , etc used with three finite types with cardinalities  $A$ ,  $B$ , and  $C$ . (For example, the cardinality of  $\text{Bool}$  is 2, the cardinality of  $\text{Weekday}$  is 7, etc.) Then what is the cardinality of  $\text{Either } a \ b$ ?  $(a, b)$ ?  $a \rightarrow b$ ? etc. These rules for computing the cardinality suggests that  $\text{Either}$  is similar to sum,  $(,)$  is similar to product and  $(\rightarrow)$  to (flipped) power. These rules show that we can use many intuitions from high-school algebra when working with types.

**Exercise 1.12.** Functions as tuples. For any type  $t$  the type  $\text{Bool} \rightarrow t$  is basically “the same” as the type  $(t, t)$ . Implement the two functions  $\text{isoR}$  and  $\text{isoL}$  forming an isomorphism:

$$\begin{aligned} \text{isoR} &:: (\text{Bool} \rightarrow t) \rightarrow (t, t) \\ \text{isoL} &:: (t, t) \rightarrow (\text{Bool} \rightarrow t) \end{aligned}$$

and show that  $\text{isoL} \circ \text{isoR} = \text{id}$  and  $\text{isoR} \circ \text{isoL} = \text{id}$ .

**Exercise 1.13.** From section 1.7.1:

- What does function composition do to a sequence? (composition on the left?, on the right?)
- How is  $\text{liftSeq}_1$  related to  $\text{fmap}$ ?  $\text{liftSeq}_0$  to  $\text{conSeq}$ ?

**Exercise 1.14.** When is  $\text{embed } (\text{eval } e) == e$ ?

Step 0: type the quantification: what is the type of  $e$ ?

Step 1: what equality is suitable for this type?

Step 2: if you use “equality up to eval” — how is the resulting property related to the first round-trip property?