Domain Specific Languages of Mathematics: Lecture Notes

Patrik Jansson Cezar Ionescu

January 16, 2017

Abstract

TODO: Fill in abstract

1 Lecture 01

This lecture is partly based on the paper [Ionescu and Jansson, 2016] from the International Workshop on Trends in Functional Programming in Education 2015. We will implement certain concepts in the functional programming language Haskell and the code for this lecture is placed in a module called *DSLsofMath.L01* that starts here:

```
module DSLsofMath.L01 where
```

1.1 A case study: complex numbers

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

Adams and Essex introduce complex numbers in Appendix 1. The section *Definition of Complex Numbers* begins with:

> We begin by defining the symbol i, called **the imaginary unit**, to have the property
>
> $$i^2 = -1$$
>
> Thus, we could also call i the square root of -1 and denote it $\sqrt{-1}$. Of course, i is not a real number; no real number has a negative square.

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

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

```
data ImagUnits = I
```

$$
\begin{aligned}
&i :: \mathit{ImagUnits}\\
&i \;=\; I
\end{aligned}
$$

We use a capital I in the **data** declaration because a lowercase constructor name would cause a syntax error in Haskell.

Next, we have the following definition:

> **Definition:** A **complex number** is an expression of the form
>
> $$a + bi \qquad \text{or} \qquad a + ib,$$
>
> where a and b are real numbers, and i is the imaginary unit.

This definition clearly points to the introduction of a syntax (notice the keyword "form"). This is underlined by the presentation of *two* forms, which can suggest that the operation of juxtaposing i (multiplication?) is not commutative.

A profitable way of dealing with such concrete syntax in functional programming is to introduce an abstract representation of it in the form of a datatype:

$$
\begin{aligned}
&\textbf{data}\ \mathit{ComplexA} \;=\; \mathit{CPlus}_1\ \mathbb{R}\ \mathbb{R}\ \mathit{ImagUnits}\\
&\qquad\qquad\quad \mid\ \ \mathit{CPlus}_2\ \mathbb{R}\ \mathit{ImagUnits}\ \mathbb{R}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{showCA} :: \mathit{ComplexA} \to \mathit{String}\\
&\mathit{showCA}\ (\mathit{CPlus}_1\ x\ y\ i) \;=\; \mathit{show}\ x\ +\!\!+\ \texttt{" + "}\ +\!\!+\ \mathit{show}\ y\ +\!\!+\ \texttt{"i"}\\
&\mathit{showCA}\ (\mathit{CPlus}_2\ x\ i\ y) \;=\; \mathit{show}\ x\ +\!\!+\ \texttt{" + "}\ +\!\!+\ \texttt{"i"}\ +\!\!+\ \mathit{show}\ y
\end{aligned}
$$

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.

$$\mathit{type}\ \mathbb{R} \;=\; \mathit{Double}$$

The text continues with examples:

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

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

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

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

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

$toComplex :: \mathbb{R} \rightarrow ComplexA$
$toComplex\ x\ =\ CPlus_1\ x\ 0\ i$

Again, at this stage there are many open questions. For example, we can assume that *i1* 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 *i1* 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:

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:

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

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

$type\ ComplexD\ =\ ComplexS\ \mathbb{R}$
newtype $ComplexS\ r\ =\ CS\ (r\ ,\ \ r)$ *deriving Eq*

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

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

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

$$\begin{aligned} &re :: ComplexS\ r \quad \to\ r \\ &re\ z@(CS\ (x\ ,\ \ y)) \ =\ x \end{aligned}$$

$$\begin{aligned} &im :: ComplexS\ r \quad \to\ r \\ &im\ z@(CS\ (x\ ,\ \ y)) \ =\ y \end{aligned}$$

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 *ComplexS*, for example:

$$\begin{aligned} &(+.) :: Num\ r\ \Rightarrow\ ComplexS\ r\ \to\ ComplexS\ r\ \to\ ComplexS\ r \\ &(CS\ (a\ ,\ \ b)) +. (CS\ (x\ ,\ \ y)) \ =\ CS\ ((a + x)\ ,\ \ (b + y)) \end{aligned}$$

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 *ComplexS*

1.2 A syntax for arithmetical expressions

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

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

$$evalE :: ComplexE\ \to\ ComplexD$$

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

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

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

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

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

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

$$
\begin{aligned}
&\mathit{testE1} = \mathit{Plus}\ (\mathit{ToComplex}\ 3)\ (\mathit{Times}\ (\mathit{ToComplex}\ 2)\ \mathit{ImagUnit})\\
&\mathit{testE2} = \mathit{Times}\ \mathit{ImagUnit}\ \mathit{ImagUnit}
\end{aligned}
$$

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

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

and that *fromCS* is an embedding:

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

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

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

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

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

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

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

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

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

$$
\begin{aligned}
&\mathit{notAssocEvidence} :: (\mathit{Double}\ ,\ \ \mathit{Double}\ ,\ \ \mathit{Double}\ ,\ \ \mathit{Bool})\\
&\mathit{notAssocEvidence} = (\mathit{lhs}\ ,\ \ \mathit{rhs}\ ,\ \ \mathit{lhs} - \mathit{rhs}\ ,\ \ \mathit{lhs} == \mathit{rhs})
\end{aligned}
$$

```
  where lhs = (1 +1) +1/3
        rhs = 1 + (1 +1/3)
```

For completeness: this is the answer:

```
( 2.3333333333333335          -- Notice the five at the end
, 2.333333333333333 ,         -- which is not present here.
, 4.440892098500626e − 16     -- The difference
, False)
```

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

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


toComplexP :: Num a ⇒ a → ComplexP a
toComplexP x = FromCartesian x (fromInteger 0)


evalCP :: Num r ⇒ ComplexP r → ComplexS r
evalCP (FromCartesian x y) = CS (x , y)
evalCP (l :+: r) = evalCP l +. evalCP r
evalCP (l :*: r) = evalCP l *. evalCP r


instance Num a ⇒ Num (ComplexP a) where
  (+) = (:+:)
  (*) = (:*:)
  fromInteger = toComplexP ∘ fromInteger
    -- TODO: add a few more operations
```

1.3 Some helper functions

```
propAssocAdd :: (Eq a, Num a) ⇒ a → a → a → Bool
propAssocAdd = propAssocA (+)


(*.) :: Num r ⇒ ComplexS r → ComplexS r → ComplexS r
CS (ar, ai) *. CS (br, bi) = CS (ar ∗ br − ai ∗ bi, ar ∗ bi + ai ∗ br)


instance Show r ⇒ Show (ComplexS r) where
  show = showCS


showCS :: Show r ⇒ ComplexS r → String
showCS (CS (x, y)) = show x ++ " + " ++ show y ++ "i"
```

References

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

C. Ionescu and P. Jansson. Domain-specific languages of mathematics: Presenting mathematical analysis using functional programming. In J. Jeuring and J. McCarthy, editors, *Proceedings of the 4th and 5th International Workshop on Trends in Functional Programming in Education, Sophia-Antipolis, France and University of Maryland College Park, USA, 2nd June 2015 and 7th June 2016*, volume 230 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–15. Open Publishing Association, 2016. doi: 10.4204/EPTCS.230.1.