Domain-Specific Languages of Mathematics: Presenting Mathematical Analysis using Functional Programming

Cezar Ionescu
Chalmers Univ. of Technology
cezar@chalmers.se

Patrik Jansson
Chalmers Univ. of Technology
patrikj@chalmers.se

In this paper, we present the approach underlying a course on *Domain-Specific Languages of Mathematics* [16], which is currently being developed at Chalmers in response to difficulties faced by third-year students in learning and applying classical mathematics (mainly real and complex analysis). The main idea is to encourage the students to approach mathematical domains from a functional programming perspective: to identify the main functions and types involved and, when necessary, to introduce new abstractions; to give calculational proofs; to pay attention to the syntax of the mathematical expressions; and, finally, to organise the resulting functions and types in domain-specific languages.

1 Introduction

In an article published in 2000 [20], de Moor and Gibbons start by presenting an exam question for a first-year course on algorithm design. The question was not easy, but it also did not seem particularly difficult. Still:

> In the exam itself, however, no one got the answer right, so apparently this kind of question is too hard. That is discouraging, especially in view of the highly sophisticated problems that the same students can solve in mathematics exams. Why is programming so much harder?

Fifteen year later, we are confronted at Chalmers with the opposite problem: many third-year students are having unusual difficulties in courses involving classical mathematics (especially analysis, real and complex) and its applications, while they seem quite capable of dealing with "highly sophisticated problems" in computer science and software engineering. Why is mathematics so much harder?

One of the reasons for that is, we suspect, that by the third year these students have grown very familiar with what could be called "the computer science perspective". For example, computer science places strong emphasis on syntax and introduces conceptual tools for describing it and resolving potential ambiguities. In contrast to this, mathematical notation is often ambiguous and context-dependent, and there is no attempt to even make this ambiguity explicit (Sussman and Wisdom talk about "variables whose meaning depends upon and changes with context, as well as the sort of impressionistic mathematics that goes along with the use of such variables", see [25]).

Further, proofs in computer science tend to be more formal, often using an equational logic format with explicit mention of the rules that justify a given step, whereas mathematical proofs are presented in natural language, with many steps being justified by an appeal to intuition and to the semantical content, leaving a more precise justification to the reader. Unfortunately, the task of providing such a justification requires a certain amount of expertise, and can be discouraging to the beginner.

Mathematics requires (and rewards) active study. Halmos, in a book that cannot be strongly enough recommended, phrases it as follows ([13], page 69):

To appear in EPTCS.

> It's been said before and often, but it cannot be overemphasised: study actively. Don't just read it; fight it!

but, as in the case of proofs, following this advice requires some expertise, otherwise it risks being taken in too physical a sense.

In this paper, we present the approach underlying a course on *Domain-Specific Languages of Mathematics* [16], which is currently being developed at Chalmers to alleviate these problems. The main idea is to show the students that they are, in fact, well-equipped to take an active approach to mathematics: they need only apply the software engineering and computer science tools they have acquired in the rest of their studies. The students should approach a mathematical domain in the same way they would any other domain they are supposed to model as a software system.

In particular, we are referring to the approach that a functional programmer would take. Functional programming deals with Modelling in terms of types and pure functions, and this seems to be ideal for a domain where functions are natural objects of study, and which is possibly the only one where we can be certain that data is immutable.

Additionally, functional programming has, from the very beginning, been connected to the notion of mathematical proof. For example, the influential language ML was originally developed in the 70s to be "a medium in which proofs ... can be expressed, as well as heuristic algorithms for finding those proofs" ([2], page 205).

Explicitly introducing functions and their types, often left implicit in mathematical texts, is an easy way to begin an active approach to study. Moreover, it serves as a way of relating new concepts to familiar ones: even in continuous mathematics, many functions turn out to be variants of the standard Haskell ones (not surprising, considering that the former were often the inspiration for the latter). Finally, the explicit elements we introduce can be reasoned about and lead to proofs in a more calculational style. Section 2 presents these elements in detail.

Section 3 deals with the higher-level question of the organisation of our types and functions. We emphasise *domain-specific languages* (DSLs, [9]), since they are a good fit for the mathematical domain, which can itself be seen as a collection of specialised languages. Moreover, building DSLs is increasingly becoming a standard industry practice [8]. Empirical studies show that DSLs can lead to fundamental increases in productivity, above alternative modelling approaches such as UML [27]. The course we are developing will exercise and develop new skills in designing and implementing DSLs. The students will not simply use previously acquired software engineering expertise, but also extend it, which can be an important motivating aspect.

Both sections contain simple examples to illustrate our approach to an active reading of mathematical texts. The text we are reading is the standard textbook used at Chalmers in the analysis course for first year students (Adams and Essex, [1]), though we shall occasionally cite a few other texts as well. At this stage, it is important that we prevent a potentially grave misunderstanding of our intentions. We do not present the results of the active reading as an ideal presentation of the mathematical concepts involved! That a presentation which is too explicit and complete can rob the readers of a precious opportunity to exercise themselves is known to mathematicians at least since Descartes' *Geometry* [6]:

> But I shall not stop to explain this in more detail, because I should deprive you of the pleasure of mastering it yourself, as well as of the advantage of training your mind by working over it, which is in my opinion the principal benefit to be derived from this science.

On the other hand, the *Geometry* was considered too obscure to be read and didn't gain in popularity until van Schooten's explanatory edition, so perhaps there is room for compromise. In any case, both mathematicians ([29, 17]) and computer scientists ([12, 4]) have argued that the computer science perspective could bring a valuable contribution to mathematical education: we see our work as a step in this direction.

We have been referring to the computer science students at Chalmers since they are our main target audience, but we hope we can also attract some of the mathematics students. Indeed, for the latter the course can serve as an introduction to functional programming and to DSLs by means of examples with which they are familiar. Thus, ideally, the course would improve the mathematical education of computer scientists and the computer science education of mathematicians.

A word of warning. We assume familiarity with Haskell (though not with calculus), and we will take certain notational and semantic liberties with it. For example, we will use : for the typing relation, instead of ::, and we will assume the existence of the set-theoretical datatypes and operations used in classical analysis, even though they are not implementable. For example, we assume we have at our disposal a powerset operation $\mathscr{P}$, (classical) real numbers $\mathbb{R}$, choice operations, and so on. We shall also use the standard notation for intervals, which can lead to an overloading of the Haskell list notation ($[a,\ b]$ may denote a closed interval or a two-element list, depending on the context).

This paper, some associated source code and the DSLsofMath course material is being collected on GitHub: `https://github.com/DSLsofMath`. Contributions are welcome!

2 Functions and types

One of the most useful actions of the student of a mathematical text is to identify and type the functions involved. If the notation she uses is inadequate for this purpose, then her ability will be severely impaired. This is one of the main reasons for using functional programming as the basis of our “requirements engineering” in a mathematical domain.

Many important mathematical objects are functions. Arguably, the basic objects of study in undergraduate analysis are sequences of one type or another. Sequences are usually defined as functions of positive integers (for example in Rudin [24]); for the functional programmer it is perhaps more natural to model them as functions of natural numbers, using $a : \mathbb{N} \to X$ where a mathematician would write $\{a_n\}$ or similar. For brevity, we shall use X to denote a $\mathbb{R}$ or $\mathbb{C}$, as is common in undergraduate analysis, but in a classroom setting this could also be an opportunity to explain type classes such as *Num*.

The notion of *limit* is first defined for sequences. The operation of taking the limit is an example of a higher-order function:

$$\mathit{lim} : (\mathbb{N} \to X) \to X$$

Higher-order functions are ubiquitous in mathematical analysis, hence the importance of using a notation that supports them in a simple way. In fact, although we will not use it in this paper, it is often necessary to account for *dependent types*. Intervals can, for instance, be represented as dependent types, and all interval operations are naturally dependently-typed. In such cases, we would prefer to use the notation of Agda [22, 14] or Idris [5].

Convergent sequences can be used to represent real numbers, but the use of sequences is much more diverse. We can think of the sequence of coefficients as a syntax that can be given multiple interpretations:

- the sequence represents the coefficients of a series. In this case, the semantics is usually given in terms of the limit (if it exists) of the sequence of partial sums:

$$\begin{aligned}\Sigma &: (\mathbb{N} \to X) \to X \\ \Sigma f &= \mathit{lim}\, s \ \textbf{where}\ s\, n = \mathit{sum}\, (\mathit{map} f\, [0 \mathinner{..} n])\end{aligned}$$

- the sequence represents the coefficients of a power series. In this case, the semantics is that of a function, whose values are defined in terms of the evaluation of a series:

  $$\begin{aligned}\mathit{Powers} &: (\mathbb{N} \to X) \to (X \to X) \\ \mathit{Powers}\, a\, x &= \Sigma f \ \textbf{where}\ f\, n = a\, n * x^n\end{aligned}$$

  Power series are perhaps *the* fundamental concept of undergraduate analysis and its applications: they lead to elementary and analytic functions, they are the starting point for the Fourier and Laplace transformations, interval analysis, etc. Therefore, the student might find it puzzling that in most textbooks they do not have a symbolic representation of their own, outside the somewhat unwieldy $\sum_{n=0}^{\infty} a_n X^n$.

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 [7] 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 $\mathscr{L}$ involves the operation of integration and yields the new function $\mathscr{L}\ \{f\ (t)\} = F\ (s)$ of a new independent variable s.

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

$$\begin{aligned}&\mathit{type}\, T = \mathbb{R} \\ &\mathit{type}\, S = \mathbb{C} \\ &\mathscr{L} : (T \to \mathbb{C}) \to (S \to \mathbb{C})\end{aligned}$$

In the following subsection, we present two simple examples of "close reading" a mathematical text, trying to identify and type the functions involved, and to relate them to the familiar elements of functional programming.

2.1 Two examples

Consider the following statement of the completeness property for $\mathbb{R}$ ([1], page 4):

> The *completeness* property of the real number system is more subtle and difficult to understand. One way to state it is as follows: if A is any set of real numbers having at least one number in it, and if there exists a real number y with the property that $x \leqslant y$ for every $x \in A$ (such a number y is called an **upper bound** for A), then there exists a smallest such number, called the **least upper bound** or **supremum** of A, and denoted $\mathit{sup}\ (A)$. Roughly speaking, this says that there can be no holes or gaps on the real line—every point corresponds to a real number.

The functional programmer trying to make sense of this "subtle and difficult to understand" property will start by making explicit the functions involved:

$$\mathit{sup} : \mathscr{P}^{+}\,\mathbb{R} \to \mathbb{R}$$

sup is defined only for those subsets of $\mathbb{R}$ which are bounded from above; for these it returns the least upper bound.

Functional programmers are acquainted with a large number of standard functions. Among these are *minimum* and *maximum*, which return the smallest and the largest element of a given (non-empty) list. It is easy enough to specify set versions of these functions, for example:

$$\begin{aligned} &\mathit{min} &&: \quad \mathscr{P}^{+}\,\mathbb{R} \to \mathbb{R} \\ &\mathit{min}\,A = x &&\Longleftrightarrow \quad (x \in A) \wedge (\forall\, a \in A.\ x \leqslant a) \end{aligned}$$

min on sets enjoys similar properties to its list counterpart, and some are easier to prove in this context, since the structure is simpler (no duplicates, no ordering of elements). For example, we have

> If $y < \mathit{min}\,A$, then $y \notin A$.

Exploring the relationship between the "new" function *sup* and the familiar *min* and *max* can dispel some of the difficulties involved in the completeness property. For example, *sup A* is similar to *max A*: if the latter is defined, then so is the former, and they are equal. But *sup A* is also the smallest element of a set, which suggests a connection to *min*. To see this, introduce the function

$$\begin{aligned} &\mathit{ubs} &&: \quad \mathscr{P}\,\mathbb{R} \to \mathscr{P}\,\mathbb{R} \\ &\mathit{ubs}\,A &&= \{x \mid x \in \mathbb{R},\ x\ \mathit{upper\ bound\ of}\ A\} \\ & &&= \{x \mid x \in \mathbb{R}, \forall\, a \in A.\ a \leqslant x\} \end{aligned}$$

which returns the set of upper bounds of A. The completeness axiom can be stated as

> Assume an $A : \mathscr{P}^{+}\,\mathbb{R}$ with an upper bound $u \in \mathit{ubs}\,A$.
>
> Then $s = \mathit{sup}\,A = \mathit{min}\,(\mathit{ubs}\,A)$ exists.

where

$$\begin{aligned} &\mathit{sup} : \mathscr{P}^{+}\,\mathbb{R} \to \mathbb{R} \\ &\mathit{sup} = \mathit{min} \circ \mathit{ubs} \end{aligned}$$

So, now we know that for any bounded set A we have a supremum $s : \mathbb{R}$, but s need not be in A — could there be a "gap"? (An example set could be $A = \{7 - 1/n \mid n \in \mathbb{N}^{+}\}$ with $s = \mathit{sup}\,A = 7 \notin A$.) If we by "gap" mean "an ε-neighbourhood between A and s" we can prove there is in fact no "gap".

The explicit introduction of functions such as *ubs* allows us to give calculational proofs in the style introduced by Wim Feijen and used in many computer science textbooks, especially in functional programming (such proofs are more amenable to automatic verification, see for example the algebra of programming library implemented in Agda [21]). For example, if $s = sup\,A$:

$$
\begin{array}{ll}
 & 0 < \varepsilon \\
\Rightarrow & \{ \text{ arithmetic } \} \\
 & s - \varepsilon < s \\
\Rightarrow & \{ s = min\,(ubs\,A), \text{ property of } min \text{ from above } \} \\
 & s - \varepsilon \notin ubs\,A \\
\Rightarrow & \{ \text{ set membership } \} \\
 & \neg \forall a \in A.\; a \leqslant s - \varepsilon \\
\Rightarrow & \{ \text{ quantifier negation } \} \\
 & \exists a \in A.\; s - \varepsilon < a \\
\Rightarrow & \{ \text{ definition of upper bound } \} \\
 & \exists a \in A.\; s - \varepsilon < a \leqslant s \\
\Rightarrow & \{ \text{ subtract } s, \text{ use } 0 < \varepsilon \} \\
 & \exists a \in A.\; -\varepsilon < a - s < \varepsilon \\
\Rightarrow & \{ \text{ absolute value } \} \\
 & \exists a \in A.\; (|a - s| < \varepsilon) \\
\Rightarrow & \{ \text{ introduce the neighbourhood function } V : X \to \mathbb{R}_{>0} \to \mathscr{P}\,X \} \\
 & \exists a \in A.\; a \in V\,s\,\varepsilon
\end{array}
$$

This simple proof shows that we can always find an element of A as near to $sup\,A$ as we want, which explains perhaps the above statement “Roughly speaking, [the completeness axiom] says that there can be no holes or gaps on the real line—every point corresponds to a real number.”

As another example of work on the text, consider the following definition ([1], page A-23):

Limit of a sequence

We say that $lim\,x_n = L$ if for every positive number ε there exists a positive number $N = N\,(\varepsilon)$ such that $|x_n - L| < \varepsilon$ holds whenever $n \geqslant N$.

There are many opportunities for functional programmers to apply their craft here, such as

- giving an explicit typing $lim : (\mathbb{N} \to X) \to X$ and writing $lim\,x$ in order to avoid the impression that the result depends on some particular value x_n;
- giving an explicit typing for the absolute value function $|\_| : X \to \mathbb{R}_{\geq 0}$;
- introducing explicitly the function $N : \mathbb{R}_{>0} \to \mathbb{N}$;
- introducing a neighbourhood function $V : X \to \mathbb{R}_{>0} \to \mathscr{P}\,X$ with

$$V\,x\,\varepsilon = \{x' \mid x' \in X, |x' - x| < \varepsilon\}$$

These are all just changes in the notation of elements already present in the text (the *neighbourhood* function V is introduced in Adams, but first on page 567, long after the chapter on sequences and convergence, page 495). Many real analysis textbooks adopt, in fact, the one or the other of these changes. However, functional programmers will probably observe that the expression $a_n\ \ldots\ \mathit{whenever}\ n \geqslant N$ refers to the Nth tail of the sequence, i.e., to the elements remaining after the first N elements have been dropped. This recalls the familiar Haskell function $\mathit{drop} : \mathit{Int} \to [a] \to [a]$, which can be recast to suit the new context:

$$\begin{aligned}
&\mathit{drop} : \mathbb{N} \to (\mathbb{N} \to X) \to (\mathbb{N} \to X)\\
&\mathit{drop}\,n\,f = \lambda\,(i : \mathbb{N}) \to f\,(n + i)\\
&\mathit{Drop} : \mathbb{N} \to (\mathbb{N} \to X) \to \mathscr{P}\,X\\
&\mathit{Drop}\,n\,f = \mathit{range}\,(\mathit{drop}\,n\,f)\\
&\qquad\quad = \{f\,i \mid i \in \mathbb{N}, n \leqslant i\}
\end{aligned}$$

The function *Drop* has many properties, for example:

- anti-monotone in the first argument

  $$m \leqslant n \Rightarrow \mathit{Drop}\,n\,f \subseteq \mathit{Drop}\,m\,f$$

  in particular $\mathit{Drop}\,n\,f \subseteq \mathit{Drop}\,0\,f$ for all n;
- if f is increasing, then, for any m and n

  $$\mathit{ubs}\,(\mathit{Drop}\,m\,f) = \mathit{ubs}\,(\mathit{Drop}\,n\,f)$$

  and therefore, if $\mathit{Drop}\,0\,f$ is bounded

  $$\mathit{sup}\,(\mathit{Drop}\,m\,f) = \mathit{sup}\,(\mathit{Drop}\,n\,f)$$
- if f is increasing, then

  $$\mathit{Drop}\,n\,f \subseteq [f\,n, \infty)$$

Using *Drop*, we have that

$$\begin{aligned}
&\mathit{lim}\,f = L\\
&\Longleftrightarrow\\
&\exists\,N : \mathbb{R}_{>0} \to \mathbb{N}.\ \forall\,\varepsilon \in \mathbb{R}_{>0}.\ \mathit{Drop}\,(N\,\varepsilon)\,f \subseteq V\,L\,\varepsilon
\end{aligned}$$

This formulation has the advantage of eliminating one of the three quantifiers in the definition of limit. In general, introducing functions and operations on functions leads to fewer quantifiers. For example, we could lift inclusion of sets to the function level: for $f,\ g : A \to \mathscr{P}\,B$ define

$$f \subseteq g \iff \forall\,a \in A.\ f\,a \subseteq g\,a$$

and we could eliminate the quantification of ε above:

$$\begin{aligned}
&\exists\,N : \mathbb{R}_{>0} \to \mathbb{N}.\ \forall\,\varepsilon \in \mathbb{R}_{>0}.\ \mathit{Drop}\,(N\,\varepsilon)\,f \subseteq V\,L\,\varepsilon\\
&\Longleftrightarrow\\
&\exists\,N : \mathbb{R}_{>0} \to \mathbb{N}.\ (\mathit{flip}\,\mathit{Drop}\,f \circ N) \subseteq V\,L
\end{aligned}$$

The application of *flip* is necessary to bring the arguments in the correct order. As this example shows, sometimes the price of eliminating quantifiers can be too high.

We can show that increasing sequences which are bounded from above are convergent. Let f be a sequence bounded from above (i.e., with $A = \mathit{Drop}\ 0\ f$ there is some $u \in \mathit{ubs}\ A$), and let $s = \mathit{sup}\ A$. Then, we know from our previous example that $\exists\ a \in A.\ a \in V\ s\ \varepsilon$ for any ε. Or equivalently, $\forall\ \varepsilon \in \mathbb{R}_{>0}.\ \exists\ i \in \mathbb{N}.\ f\ i \in V\ s\ \varepsilon$. Finally by swapping quantifier order and introducing the name N for the function that determines i from ε we obtain $\exists\ N : \mathbb{R}_{>0} \to \mathbb{N}.\ f\ (N\ \varepsilon) \in V\ s\ \varepsilon$.

If f is increasing, we have

$$
\begin{array}{ll}
 & \mathit{Drop}\ (N\ \varepsilon)\ f \\
\subseteq & \{\ f \text{ increasing }\} \\
 & [f\ (N\ \varepsilon),\ \mathit{sup}\ (\mathit{Drop}\ (N\ \varepsilon)\ f)] \\
= & \{\ f \text{ increasing} \Rightarrow \mathit{sup}\ (\mathit{Drop}\ n\ f) = \mathit{sup}\ (\mathit{Drop}\ 0\ f) = s\ \} \\
 & [f\ (N\ \varepsilon),\ s] \\
\subseteq & \{\ f\ (N\ \varepsilon) \in V\ s\ \varepsilon\ \} \\
 & V\ s\ \varepsilon
\end{array}
$$

As before, the introduction of a new function has helped in relating familiar elements (the standard Haskell function *drop*) to new ones (the concept of limit) and to formulate proofs in a calculational style.

3 Domain-specific languages

There is no clear-cut line between libraries and DSLs, and intuitions differ. For example, in Chapter 8 of *Thinking Functionally with Haskell* ([3]), Richard Bird presents a language for pretty-printing documents based on Wadler's chapter in *The Fun of Programming* [28], but refers to it as a library, only mentioning DSLs in the chapter notes.

Both libraries and DSLs are collections of types and functions meant to represent concepts from a domain at a high level of abstraction. What separates a DSL from a library is, in our opinion, the deliberate separation of syntax from semantics, which is a feature of all programming languages (and, arguably, of languages in general).

As we have seen above, in mathematics the syntactical elements are sometimes conflated with the semantical ones ($f(t)$ versus $f(s)$, for example), and disentangling the two aspects can be an important aid in coming to terms with a mathematical text. Hence, our emphasis on DSLs rather than libraries.

The distinction between syntax and semantics is, in fact, quite common in mathematics, often hiding behind the keyword "formal". For example, *formal power series* are an attempt to present the theory of power series restricted to their syntactic aspects, independent of their semantic interpretations in terms of convergence (in the various domains of real numbers, complex numbers, intervals of reals, etc.). The "formalist" texts of Bourbaki present various domains of mathematics by emphasising their formal properties (*axiomatic structure*), then relating those in terms of "lower levels", with the lowest levels expressed in terms of set theory (so, for example, groups are initially introduced axiomatically, then various interpretations are discussed, such as "groups of transformations", which in turn are interpreted in terms of endo-functions, which are ultimately represented as sets of ordered pairs). Currently, however, even the most "formalist" mathematical texts offer to the computer scientist many opportunities for active reading.

3.1 A case study: complex numbers

To illustrate the above, we present an analytic reading of the introduction of complex numbers in [1]. The simplicity of the domain is meant to allow the reader to concentrate on the essential elements of our approach without the distraction of potentially unfamiliar mathematical concepts. Because of the exemplary character of 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 I = i
```

(We have taken the liberty of introducing a lowercase constructor, which would cause a syntax error in Haskell.)

Next, we have the following definition:

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

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

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

```
data Complex = Plus₁ ℝ ℝ I
             | Plus₂ ℝ I ℝ
```

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

```
show              : Complex → String
show (Plus₁ x y i) = show x ++ " + " ++ show y ++ "i"
show (Plus₂ x i y) = show x ++ " + " ++ "i" ++ show y
```

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 $\mathit{Plus}_1\;3\;2\;i$, $\mathit{Plus}_1\;\frac{7}{2}\;(-\frac{2}{3})\;i$, $\mathit{Plus}_2\;0\;i\;\pi$, $\mathit{Plus}_1\;(-3)\;0\;i$. We interpret the sentence "The last of these examples ..." to mean that there is an embedding of the real numbers in *Complex*, which we introduce explicitly:

$$\begin{aligned}&\mathit{toComplex} : \mathbb{R} \rightarrow \mathit{Complex}\\&\mathit{toComplex}\;x = \mathit{Plus}_1\;x\;0\;i\end{aligned}$$

Again, at this stage there are many open questions. For example, we can assume that *i1* stands for the complex number $\mathit{Plus}_2\;0\;i\;1$, but what about *i* by itself? If juxtaposition is meant to denote some sort of multiplication, then perhaps *1* can be considered as a unit, in which case we would have that *i* abbreviates *i1* and therefore $\mathit{Plus}_2\;0\;i\;1$. 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:

$$\textbf{data}\;\mathit{Complex} = \mathit{Plus}\;\mathbb{R}\;\mathbb{R}\;I$$

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

$$\textbf{data}\;\mathit{Complex} = \mathit{PlusI}\;\mathbb{R}\;\mathbb{R}$$

(The renaming of the constructor from *Plus* to *PlusI* 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: *PlusI* 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 type synonym:

$$\textsf{newtype}\ \mathit{Complex} = C\ (\mathbb{R}, \mathbb{R})$$

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 $\mathit{Re}\ (z)$. We call y the **imaginary part** of z and denote it $\mathit{Im}\ (z)$:
>
> $$\begin{aligned} \mathit{Re}\ (z) &= \mathit{Re}\ (x + yi) = x \\ \mathit{Im}\ (z) &= \mathit{Im}\ (x + yi) = y \end{aligned}$$

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

$$\begin{aligned} &\mathit{Re} : \mathit{Complex} \rightarrow \mathbb{R} \\ &\mathit{Re}\ z@(C\ (x, y)) = x \end{aligned}$$

$$\begin{aligned} &\mathit{Im} : \mathit{Complex} \rightarrow \mathbb{R} \\ &\mathit{Im}\ z@(C\ (x, y)) = y \end{aligned}$$

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 *Complex*, for example:

$$\begin{aligned} &(+) : \mathit{Complex} \rightarrow \mathit{Complex} \rightarrow \mathit{Complex} \\ &(C\ (a, b)) + (C\ (x, y)) = C\ ((a + x), (b + y)) \end{aligned}$$

or we can build a datatype of "syntactic" Complex numbers from the algebraic operations to arrive at a *deep embedding*:

$$\begin{aligned} \textsf{data}\ \mathit{ComplexSyntax} = {} & i \\ \mid {} & \mathit{ToComplex}\ \mathbb{R} \\ \mid {} & \mathit{Plus}\ \ \mathit{ComplexSyntax}\ \mathit{ComplexSyntax} \\ \mid {} & \mathit{Times}\ \mathit{ComplexSyntax}\ \mathit{ComplexSyntax} \\ \mid {} & \ldots \end{aligned}$$

The type *ComplexSyntax* can then be turned into an abstract datatype, by hiding the representation and providing corresponding operations like $(+) = \mathit{Plus}$, etc. Deep embedding offers a cleaner separation between syntax and semantics, making it possible to compare and factor out the common parts of various languages. For the computer science students, this is a way of approaching structural algebra; for the mathematics students, this is a way to learn the ideas of abstract datatypes, type classes, folds, by relating them to the familiar notions of mathematical structures and homomorphisms (see [10] for a discussion of the relationships between deep and shallow embeddings and folds). We want to show the students both the shallow and the deep approach and help them understand when more or less focus on syntax is helpful.

Adams and Essex then proceed to introduce the geometric interpretation of complex numbers, i.e., the isomorphism between complex numbers and points in the Euclidean plane as pairs of coordinates. The isomorphism is not given a name, but we can use the constructor C defined above. They then define the polar representation of complex numbers, in terms of modulus and argument:

> The distance from the origin to the point (a, b) corresponding to the complex number $w = a + bi$ is called the **modulus** of w and is denoted by $|w|$ or $|a + bi|$:
>
> $$|w| = |a + bi| = \sqrt{a^2 + b^2}$$
>
> If the line from the origin to (a, b) makes angle θ with the positive direction of the real axis (with positive angles measured counterclockwise), then we call θ an **argument** of the complex number $w = a + bi$ and denote it by $\mathit{arg}\ (w)$ or $\mathit{arg}\ (a + bi)$.

Here, the constant repetitions of "$w = a + bi$" and "$f\ (w)$ or $f\ (a + bi)$" are caused not just by the unavailability of pattern-matching, but also by the absence of the explicit isomorphism C. We need only use $|C\ (a, b)| = \sqrt{a^2 + b^2}$, making clear that the modulus and arguments are actually defined by pattern matching.

Once the principal argument has been defined as the unique argument in the interval $(-\pi, \pi]$, the way is opened to a different interpretation of complex numbers (usually called the *polar representation* of complex numbers):

$$\textbf{newtype}\ \mathit{Complex}' = C'\ (\mathbb{R}_{\geq 0}, (-\pi, \pi])$$

C' constructs a "geometric" complex number from a non-negative modulus and a principal argument; the (non-implementable) constraints on the types ensure uniqueness of representation.

The importance of this alternative representation is that the operations on its elements have a different natural interpretation, namely as geometrical operations. For example, multiplication with $C'\ (m, \theta)$ represents a re-scaling of the Euclidean plane with a factor m, coupled with a rotation with angle θ. Thus, multiplication with i (which is $C'\ (\mathit{1}, \frac{\pi}{2})$ in polar representation) results in a counterclockwise rotation of the plane by $90°$. This interpretation of i seems independent of the originally proposed arithmetical one ("the square root of -1"), and the polar representation of complex numbers leads to a different, geometrical language.

It can be an interesting exercise to develop this language (of scalings, rotations, etc.) "from scratch", without reference to complex numbers. In a deep embedding, the result is a datatype representing a syntax that is quite different from the one suggested by the algebraic operations. The fact that this language can also be given semantics in terms of complex numbers could then be seen as somewhat surprising, and certainly in need of proof. This would introduce in a simple setting the fact that many fundamental theorems in mathematics establish that two languages with different syntaxes have, in fact, the same semantics. A more elaborate example is that of the identity of the language of matrix manipulations as implemented in Matlab and that of linear transformations. At the undergraduate level, the most striking example is perhaps that of the identity of holomorphic functions (the language of complex derivatives) and (regular) analytic functions (the language of complex power series).

4 Conclusions and future work

We have presented the basic ingredients of an approach that uses functional programming as a way of helping students deal with classical mathematics and its applications:

- make functions and the types explicit
- use types as carriers of semantic information, not just variable names
- introduce functions and types for implicit operations such as the power series interpretation of a sequence
- use a calculational style for proofs
- organise the types and functions in DSLs

Given the main course objective, enabling the students to better tackle mathematical domains by applying the computing science perspective, we intend to measure how well the students do in ulterior courses that require mathematical competence. For example, we will measure the percentage of students who, having taken DSLsofMath, pass the third-year courses *Transforms, signals and systems* and *Control Theory (Reglerteknik)*, which are current major stumbling blocks. Since the course will, at least initially, be an elective one, we will also have the possibility of comparing the results with those of a control group (students who have not taken the course).

The lessons in this course will be organised around the active reading of mathematical texts (suitably prepared in advance). In the opening lessons, we will deal with domains of mathematics which are relatively close to functional programming, such as elementary category theory, in order to have the chance to introduce newcomers to functional programming, and the students in general to our approach.

After that, the selection of the subjects will mostly be dictated by the requirements of the engineering curriculum. They will contain:

- basic properties of complex numbers
- the exponential function
- elementary functions
- holomorphic functions
- the Laplace transform

We shall take advantage of the fact that some parts of these topics have been treated before from a functional programming perspective [18, 19, 23].

One of the important course elements we have left out of this paper is that of using the modelling effort performed in the course for the production of actual mathematical software. One of the reasons for this omission is that we wanted to concentrate on the more conceptual part that corresponds to the specification of that software, and as such is a prerequisite for it. The development of implementations on the basis of these specifications will be the topic of most of the exercise sessions we will organise. That the computational representation of mathematical concepts can greatly help with their understanding was conclusively shown by Sussman and Wisdom in their recent book on differential geometry [26].

On the other hand, classical mathematical theorems often lead to non-implementable specifications (for example, there is no algorithm for finding the minima and maxima of arbitrary continuous functions on a closed interval, although we have an easy classical proof of their existence). There are many possibilities of dealing with such cases, and we shall explore some of them in the exercises sessions. For instance, in scientific programming, one is often interested in correctness “up to implication”: the program would work as expected, say, if one would use real numbers instead of floating-point values. Such counterfactuals are impossible to test but they can be encoded as types and proven [15].

We believe that this approach can offer an introduction to computer science for the mathematics students. We plan to actively involve the mathematics faculty at Chalmers, via guest lectures and regular meetings, in order to find the suitable middle ground we alluded to in the introduction: between a presentation that is too explicit, turning the student into a spectator of endless details, and one that is too implicit and leaves so much for the students to do that they are overwhelmed. Ideally, some of the features of our approach would be worked into the earlier mathematical courses.

The computer science perspective has been quite successful in influencing the presentation of discrete mathematics. For example, the classical textbook of Gries and Schneider, *A Logical Approach to Discrete Math* [11], has been well-received by both computer scientists and mathematicians. When it comes to continuous mathematics, however, there is no such influence to be felt. The work presented here represents the starting point of an attempt to change this state of affairs.

References

[1] Robert Alexander Adams & Christopher Essex (2010): *Calculus: a complete course*, 7th edition. Pearson Canada.

[2] Federico Biancuzzi et al. (2009): *Masterminds of programming: Conversations with the creators of major programming languages*. O’Reilly Media, Inc.

[3] Richard Bird (2014): *Thinking functionally with Haskell*. Cambridge University Press.

[4] Raymond Boute (2009): *The decibel done right: a matter of engineering the math*. *Antennas and Propagation Magazine, IEEE* 51(6), pp. 177–184.

[5] Edwin Brady (2013): *Idris, a general-purpose dependently typed programming language: Design and implementation*. *Journal of Functional Programming* 23(05), pp. 552–593.

[6] Rene Descartes (1954): *The Geometry, translated by E. Smith and ML Latham*.

[7] Charles Henry Edwards, David E Penney & David Calvis (2008): *Elementary Differential Equations*, 6h edition. Pearson Prentice Hall Upper Saddle River, NJ.

[8] Martin Fowler (2010): *Domain-specific languages*. Pearson Education.

[9] Jeremy Gibbons (2013): *Functional Programming for Domain-Specific Languages*. In Viktória Zsók, Zoltán Horváth & Lehel Csató, editors: *Central European Functional Programming - Summer School on Domain-Specific Languages*, *LNCS* 8606, Springer, pp. 1–28.

[10] Jeremy Gibbons & Nicolas Wu (2014): *Folding Domain-specific Languages: Deep and Shallow Embeddings (Functional Pearl)*. In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP ’14, ACM, New York, NY, USA, pp. 339–347, doi:10.1145/2628136.2628138.

[11] David Gries & Fred B Schneider (1993): *A logical approach to discrete math*. Springer.

[12] David Gries & Fred B Schneider (1995): *Teaching math more effectively, through calculational proofs*. *American Mathematical Monthly*, pp. 691–697.

[13] Paul Halmos (1985): *I want to be a mathematician*. Springer-Verlag New York.

[14] Cezar Ionescu & Patrik Jansson (2013): *Dependently-typed programming in scientific computing*. In: *Implementation and Application of Functional Languages*, Springer Berlin Heidelberg, pp. 140–156.

[15] Cezar Ionescu & Patrik Jansson (2013): *Testing versus proving in climate impact research*. In: *Proceedings of the 18th Workshop Types for Proofs and Programs (TYPES'11)*, 19, pp. 41–54.

[16] Cezar Ionescu & Patrik Jansson (2015): *Domain-Specific Languages of Mathematics*. Available at `https://www.student.chalmers.se/sp/course?course_id=24179`. Course plan for DAT325, Chalmers University of Technology.

[17] Roger Kraft (2004): *Functions and Parameterizations as Objects to Think With*. In: *Maple Summer Workshop, July 2004, Wilfrid Laurier University, Waterloo, Ontario, Canada.*

[18] M Douglas McIlroy (1999): *Functional pearl: Power series, power serious*. *J. of Functional Programming* 9, pp. 323–335.

[19] M Douglas McIlroy (2001): *The music of streams*. *Information Processing Letters* 77(2), pp. 189–195.

[20] Oege de Moor & Jeremy Gibbons (2000): *Pointwise Relational Programming*. In Teodor Rus, editor: *Algebraic Methodology and Software Technology*, *Lecture Notes in Computer Science* 1816, Springer Berlin Heidelberg, pp. 371–390.

[21] Shin-Cheng Mu, Hsiang-Shang Ko & Patrik Jansson (2009): *Algebra of programming in Agda: Dependent types for relational program derivation*. *Journal of Functional Programming* 19(5), p. 545.

[22] Ulf Norell (2007): *Towards a practical programming language based on dependent type theory*. 32, Chalmers University of Technology.

[23] D Pavlović & V. Pratt (1999): *On coalgebra of real numbers*. *Electronic Notes in Theoretical Computer Science* 19, pp. 103–117.

[24] Walter Rudin (1976): *Principles of Mathematical Analysis*.

[25] Gerald Jay Sussman & Jack Wisdom (2002): *The role of programming in the formulation of ideas*. Artificial Intelligence Laboratory memo AIM-2002-018, MIT.

[26] Gerald Jay Sussman & Jack Wisdom (2013): *Functional Differential Geometry*. MIT Press.

[27] J Tolvanen (2011): *Industrial Experiences on Using DSLs in Embedded Software Development*. In: *Proceedings of Embedded Software Engineering Kongress (Tagungsband), December 2011.*

[28] Philip Wadler (2003): *A prettier printer*. In Oege de Moor, Jeremy Gibbons & Geraint Jones, editors: *The Fun of Programming*, Palgrave Macmillan, pp. 223–243.

[29] Charles Wells (1995): *Communicating mathematics: Useful ideas from computer science*. *American Mathematical Monthly*, pp. 397–408.