

Proceedings TFPIE 2015: the Fourth International Workshop on Trends in Functional Programming in Education

Johan Jeuring (editor)

Technical Report UU-CS-2015-015
November 2015

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Preface

The Fourth International Workshop on Trends in Functional Programming in Education, TFPIE 2015, was held on June 2, 2015 in Sophia-Antipolis, France. It was co-located with TFP 2015, the Symposium on Trends in Functional Programming. The goal of TFPIE is to gather researchers, professors, teachers, and all professionals interested in functional programming in education. This includes the teaching *of* functional programming, but also the application of functional programming as a tool for teaching other topics, e.g. computational concepts, complexity, logic and reasoning, and even disciplines, e.g. philosophy or music. TFPIE is the heir of previous events, like Functional and Declarative Programming in Education (FDPE), to which it owes a great deal and from which it has borrowed experience and ideas. It diverges from these previous events in a significant way by fostering a spirit of frank and open discussion in a manner that differs from the modus operandi used by most annual events. With its post-workshop review process, it allows authors to improve their manuscripts by incorporating the feedback they receive during discussions at the workshop. In addition, this model allows for material that may not yet be publication-ripe to be discussed in a congenial environment. In short, TFPIE aims to offer those with novel ideas, work-in-progress and class-room tested ideas a forum for discussion. This publication model has worked well for the previous editions of TFPIE and we expect it to be a hallmark of success for future editions, as it has been for sister events like aforementioned TFP and Implementation and Application of Functional Languages (IFL).

TFPIE 2015 received 6 submissions and had 16 participants. All submissions were found to be sound and in scope by the PC Chair and invited to give a presentation. In addition, Christian Queinnec accepted our invitation to give an invited talk. He kicked-off this year's edition of TFPIE with a presentation entitled "*Teaching recursion: a repeated MOOC experiment*". The workshop comprised 6 presentations and an interesting plenary discussion.

The post-workshop review process received 6 submissions, which were vetted by the program committee, assuming scientific journal standards of publication. The two articles in this volume were selected for publication as the result of this process. In order to give you a taste of these articles, we briefly enumerate them in the order that they were presented during the workshop. In "*Domain-Specific Languages of Mathematics: Presenting Mathematical Analysis using Functional Programming*", Cezar Ionescu and Patrik Jansson present an approach to encourage students in a course on Domain-Specific Languages of Mathematics to approach mathematical domains from a functional programming perspective: to identify the main functions and types involved; 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. In "*Teaching Functional Patterns through Robotic Applications*", J. Boender, E. Currie, M. Loomes, G. Primiero, F. Raimondi describe an approach to teaching functional programming to first year Computer Science students through projects in robotics. To support these, they developed the Middlesex Robotic plATfOrm (MIRTO), an open-source platform built using Raspberry Pi, Arduino, HUB-ee wheels and running Racket.

Although the number of submitted papers was relatively low, TFPIE 2015 was an interesting meeting, with insightful discussions. It could not have taken place without the seamless and very hospitable local organization by Manuel Serrano and the TFP 2015 organizing committee. Of course, a workshop is nothing without the submitting and presenting authors, the program committee and all participants. As

PC Chair, we would like to extend our gratitude to all of these people and we're quite confident we do so for everyone in attendance. In continued support of the community of people interested in advancing education using functional programming, all of this year's event's resources have been made available on the TFPIE wiki page.

Johan Jeuring
Program Committee Chair

Program Committee

Peter Achten	Radboud Universiteit Nijmegen
Edwin Brady	University of St. Andrews
Johan Jeuring	Utrecht University and Open Universiteit Nederland
Shriram Krishnamurthi	Brown University
Rita Loogen	Philipps-Universität Marburg
Marco T. Morazán	Seton Hall University
Norman Ramsey	Tufts University

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):

Submitted to:

4th International Workshop on Trends in Functional Programming in Education, TFPPIE 2015

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 \mathcal{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 $a : \mathbb{N} \rightarrow X$ 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:

$$\lim : (\mathbb{N} \rightarrow X) \rightarrow X$$

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:

$$\begin{aligned} \Sigma & : (\mathbb{N} \rightarrow X) \rightarrow X \\ \Sigma f & = \text{lim } s \text{ where } s \ n = \text{sum } (\text{map } f \ [0 \ .. \ n]) \end{aligned}$$

- 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:

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

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 \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 as to denote a list of as). 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

$$\begin{aligned} \text{type } T & = \mathbb{R} \\ \text{type } S & = \mathbb{C} \\ \mathcal{L} & : (T \rightarrow \mathbb{C}) \rightarrow (S \rightarrow \mathbb{C}) \end{aligned}$$

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 $x \leq y$ 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 $\sup(A)$. 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:

$$\text{sup} : \mathcal{P}^+ \mathbb{R} \rightarrow \mathbb{R}$$

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:

$$\begin{aligned} \text{min} & : \mathcal{P}^+ \mathbb{R} \rightarrow \mathbb{R} \\ \text{min } A & = x \iff (x \in A) \wedge (\forall a \in A. x \leq a) \end{aligned}$$

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

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

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, $\text{sup } A$ is similar to $\text{max } A$: if the latter is defined, then so is the former, and they are equal. But $\text{sup } A$ is also the smallest element of a set, which suggests a connection to min . To see this, introduce the function

$$\begin{aligned} \text{ubs} & : \mathcal{P} \mathbb{R} \rightarrow \mathcal{P} \mathbb{R} \\ \text{ubs } A & = \{x \mid x \in \mathbb{R}, x \text{ upper bound of } A\} \\ & = \{x \mid x \in \mathbb{R}, \forall a \in A. a \leq x\} \end{aligned}$$

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

$$\text{Assume an } A : \mathcal{P}^+ \mathbb{R} \text{ with an upper bound } u \in \text{ubs } A.$$

$$\text{Then } s = \text{sup } A = \text{min } (\text{ubs } A) \text{ exists.}$$

where

$$\begin{aligned} \text{sup} & : \mathcal{P}^+ \mathbb{R} \rightarrow \mathbb{R} \\ \text{sup} & = \text{min} \circ \text{ubs} \end{aligned}$$

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 $s = \text{sup } A = 7 \notin A$.) 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 $s = \sup A$:

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

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 $n \geq N$.

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

- giving an explicit typing $\lim : (\mathbb{N} \rightarrow X) \rightarrow X$ 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 $|_| : X \rightarrow \mathbb{R}_{\geq 0}$;
- introducing explicitly the function $N : \mathbb{R}_{>0} \rightarrow \mathbb{N}$;
- introducing a neighbourhood function $V : X \rightarrow \mathbb{R}_{>0} \rightarrow \mathcal{P} X$ 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 $a_n \dots$ whenever $n \geq N$ refers to the N th tail of the sequence, i.e., to the elements remaining after the first N elements have been dropped. This recalls the familiar Haskell function $drop : Int \rightarrow [a] \rightarrow [a]$, which can be recast to suit the new context:

$$\begin{aligned} drop &: \mathbb{N} \rightarrow (\mathbb{N} \rightarrow X) \rightarrow (\mathbb{N} \rightarrow X) \\ drop\ n\ f &= \lambda (i : \mathbb{N}) \rightarrow f\ (n + i) \\ Drop &: \mathbb{N} \rightarrow (\mathbb{N} \rightarrow X) \rightarrow \mathcal{P}\ X \\ Drop\ n\ f &= range\ (drop\ n\ f) \\ &= \{f\ i \mid i \in \mathbb{N}, n \leq i\} \end{aligned}$$

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

- anti-monotone in the first argument

$$m \leq n \Rightarrow Drop\ n\ f \subseteq Drop\ m\ f$$

in particular $Drop\ n\ f \subseteq Drop\ 0\ f$ for all n ;

- if f is increasing, then, for any m and n

$$ubs\ (Drop\ m\ f) = ub\ s\ (Drop\ n\ f)$$

and therefore, if $Drop\ 0\ f$ is bounded

$$sup\ (Drop\ m\ f) = sup\ (Drop\ n\ f)$$

- if f is increasing, then

$$Drop\ n\ f \subseteq [f\ n, \infty)$$

Using $Drop$, we have that

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

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 $f, g : A \rightarrow \mathcal{P}\ B$ define

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

and we could eliminate the quantification of ε above:

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

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 $A = \text{Drop } 0 f$ there is some $u \in \text{ubs } A$), and let $s = \text{sup } A$. 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 $\exists N : \mathbb{R}_{>0} \rightarrow \mathbb{N}. f (N \varepsilon) \in V s \varepsilon$.

If f is increasing, we have

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

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

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

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 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, $3 + 2i$, $\frac{7}{2} - \frac{2}{3}i$, $i\pi = 0 + i\pi$, 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.

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 $Plus_1\ 3\ 2\ i$, $Plus_1\ \frac{7}{2}\ (-\frac{2}{3})\ i$, $Plus_2\ 0\ i\ \pi$, $Plus_1\ (-3)\ 0\ i$. 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:

$$\begin{aligned} toComplex &: \mathbb{R} \rightarrow Complex \\ toComplex\ x &= Plus_1\ x\ 0\ i \end{aligned}$$

Again, at this stage there are many open questions. For example, we can assume that iI stands for the complex number $Plus_2\ 0\ i\ I$, but what about i by itself? If juxtaposition is meant to denote some sort of multiplication, then perhaps I can be considered as a unit, in which case we would have that i abbreviates iI and therefore $Plus_2\ 0\ i\ I$. 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:

$$\mathbf{data}\ Complex = Plus\ \mathbb{R}\ \mathbb{R}\ I$$

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

$$\mathbf{data}\ Complex = PlusI\ \mathbb{R}\ \mathbb{R}$$

(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:

```
newtype Complex = C (ℝ, 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 $Re(z)$. 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*:

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

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

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:

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

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

```
data ComplexSyntax = i
  | ToComplex ℝ
  | Plus ComplexSyntax ComplexSyntax
  | Times ComplexSyntax ComplexSyntax
  | ...
```

The type *ComplexSyntax* can then be turned into an abstract datatype, by hiding the representation and providing corresponding operations like $(+) = Plus$, 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 $arg(w)$ or $arg(a + bi)$.

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):

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

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 $C'(1, \frac{\pi}{2})$ 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*, 6th 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.

Teaching Functional Patterns through Robotic Applications

J. Boender, E. Currie, M. Loomes, G. Primiero, F. Raimondi

School of Science and Technology
Middlesex University, London

{j.boender,e.currie,m.loomes,g.primiero,f.raimondi}@mdx.ac.uk

We present our approach to teaching functional programming to First Year Computer Science students at Middlesex University through projects in robotics. A holistic approach is taken to the curriculum, emphasising the connections between different subject areas. A key part of the students' learning is through practical projects that draw upon and integrate the taught material. To support these, we developed the Middlesex Robotic platform (MIRTO), an open-source platform built using Raspberry Pi, Arduino, HUB-ee wheels and running Racket (a LISP dialect). In this paper we present the motivations for our choices and explain how a number of concepts of functional programming may be employed when programming robotic applications. We present some students' work with robotics projects: we consider the use of robotics projects to have been a success, both for their value in reinforcing students' understanding of programming concepts and for their value in motivating the students.

1 Introduction

This paper discusses how the language Racket has been used in the first year of the Computer Science programme at Middlesex University, with a focus on the use of physical devices and robotics to teach aspects of functional and imperative programming and to reinforce other areas of the curriculum. The background lies in the development of a new BSc CS programme, which has now reached the end of its second year, so that the first year has seen now two cohorts of students. The first year of the programme takes a holistic approach to providing a solid grounding in computer science; there are no modules, but rather a number of interwoven themes, namely programming, physical computing, formal underpinnings, design and project work. The approach involves exposing students to key concepts in each of these areas. Taking propositional logic as an example, there is a theoretical treatment in the formal sessions, practical implementation of logic formulae with gates in the physical computing sessions, implementation as boolean functions in programming labs, modelling the language in design sessions and application of the above in project work.

One of the key decisions in the design of this programme was the choice of the programming language. Racket was chosen because it could be used as the 'glue' to hold together the other parts of the programme. Many of the concepts covered on the course can be implemented in Racket and this language proved ideal for interfacing with microcontrollers and robots in the integrative project work. It was decided at an early stage that we would try to motivate students and draw together the various topics by having them engage in projects that involved practical and 'physical' manifestations of software. The academic year was divided into three blocks and each of these had an associated project. The first block project involved the use of an Arduino micro-controller controlled using Racket. For the first cohort, the project was the design of a 3-way traffic light system for roadworks; the second cohort used an LED matrix to implement a noughts and crosses game. The second block focused on data structures, and the associated projects involved the design of a 'dungeon' game. For the third block, the students did projects based on a robot developed in-house and this is the main topic of the paper, as described below.

These projects enabled students to apply and integrate a number of topics from other areas of the curriculum. For example, they used finite state machines to describe the required mutual behaviour of the robot wheel motors. As discussed in Section 3, students used propositional logic functions implemented in Racket for tasks such as verifying that a proposed speed was within a robot's designated range. Principles from the design and formal underpinnings sessions were applied in creating new applications for the robots; for example, open- versus closed-loop feedback systems. Finally, the projects were carried out in groups, which developed the students' associated transferable skills.

The rest of the paper is organised as follows. We first provide an overview of our robotic platform in Section 2. In Section 3 we describe the patterns that we have observed and taught in the programme, and in Section 4 we present examples of students' projects. We discuss related literature in Section 5.

2 Overview of Racket and MIRTO

As with all choices of programming language, our choice of Racket was a compromise. Perhaps the major factor in our decision was that Racket could be used as a unifying notation with which to explore all of the first year material; because it is also an imperative language, we could also use it to cover the concepts of state and iteration with loops that the students would meet in their second year work with Java; and because of its functional flavour, we could use it to highlight some of the logic notions recurring in all other contexts.

A convenient feature of Racket is that all the imperative 'functions' (procedures) in the language have names that end with an exclamation mark (!). Thus students can be aware when they are programming imperatively, and if they want to use a purely functional style, they can do so by not using these functions. The ability to use 'functions' that return void and do their tasks by side effects adds the flexibility needed for many of the robot-controlling functions used in the course, while of course also helping students to learn about side effects. Therefore, while not for the functional programming purist, the flexibility and range of Racket made it an ideal first language for our CS programme.

Some features of functional programming are not so easy in Racket. For example, the use of infinite data structures is difficult because the language uses eager evaluation. However, the practical nature of the first year meant that these more esoteric aspects of functional programming were not as important as the flexibility of the language for a range of practical projects. To this aim, the Computer Science Department at Middlesex University, in collaboration with the Design, Engineering and Mathematics Department, have developed MIRTO (Middlesex Robotic plaTfOrm), a flexible open-source platform; its current design and all the source code are available on-line [10]. Mirto is composed of two units:

1. The base platform provides two HUB-ee wheels [12], which include motors and encoders (to measure actual rotation) built in, a rechargeable battery pack, front and rear castors, two bump sensors and an array of six infra-red sensors (mounted under the base), and an Arduino microcontroller board with shield to interface to all of these.
2. The top layer consists of a Raspberry Pi, running a bespoke Linux image extending the standard Raspbian image, with Racket 6.1 installed and connected to the Arduino by the serial port available on its interface connection.

The control and monitoring of the micro-controllers is obtained through running the Arduino Service Interface Protocol (ASIP), a protocol similar, in spirit, to the Firmata protocol [13] in that it enables a computer to discover, configure, read and write a microcontroller's general purpose IO pins. However,

ASIP has a smaller footprint than Firmata (using around 20% less RAM) and it supports high level abstractions that can be easily attached to hundreds of different services for accessing sensors or controlling actuators. These abstractions can decouple references to specific hardware, thus enabling different microcontrollers to be used without software modification. For an overview of the ASIP protocol see [6]. The Racket ASIP client library is available at [3] together with implementations for Input-Output, distance, motor with encoders, Infra-red sensors for line following, and NeoPixels services. The following is an example of Racket code to set pins 11, 12 and 13 of the Arduino board to HIGH:

```
1 (map (lambda (x) (digital-write x HIGH))(list 11 12 13))
```

The code above makes use of the higher-order function `map`, applied to a λ -function which applies the ASIP library function `digital-write` to the list of numbers 11, 12 and 13. As already shown in this short example, Racket provides an opportunity to teach functional programming languages in physical computing sessions.

An additional advantage of the setup with the Arduino and the Raspberry Pi is that it can be used to teach several other important concepts. For example, we use the Arduino to teach some elementary assembly programming (Atmel Studio [4] is an excellent simulator and IDE, currently in use at Middlesex). Additionally, we can teach some rudimentary Linux skills as well, such as command line operations.

3 Functional patterns for robots

Our work with MIRTO robots induces the use of functional programming patterns by students. The philosophy of the course is for students to explore ideas and learn abstract concepts by a process of practical guided discovery, with the role of the tutors as facilitators. Students' understanding is deepened in a 'spiral curriculum' approach by applying previously covered ideas in their project work. The approach is supported by the interactive nature of Racket, which enables students to try things quickly to explore why they get particular results. Students' understanding of concepts and their implementation is deepened by returning to the concept in a new context, either in a different subject area or by applying it in their project work. The project work is undertaken in groups, and the groups present their work to each other, which promotes peer learning.

3.1 Random application of functions from a list and side effects

A first example is the exploration of the concept of side effects and the difference between symbols and their evaluation through a number of small exercises. While initially such exercises can be rather abstract and the understanding gained can be shallow and transient, our students returned to these concepts when they were asked to use the Racket library for ASIP to make a MIRTO robot explore an unknown area, as follows:

- The robot should start by moving forward.
- When the robot hits an obstacle, it should stop immediately and move backwards for 0.5 seconds
- At this point, the robot should perform a left or a right rotation (randomly), and then restart and move forward until it hits the next obstacle.

As an additional feature, the time for the rotation was also to be random, say between 0.3 and 1.5 seconds, although we will ignore this aspect here. To provide a solution for this exercise, a group of students wrote two functions, one to rotate left and one to rotate right, something similar to the following:

```

1 (define moveLeft
2   (lambda ()
3     ;; code here to move left, using the
4     ;; racket-asip library
5     (printf "The robot moves left \n")
6   )
7 )
8
9 (define moveRight
10  (lambda ()
11    ;; code here to move right, using the
12    ;; racket-asip library
13    (printf "The robot moves right \n")
14  )
15 )
16
17 (list-ref (list (moveLeft) (moveRight)) (random 2))

```

We abstract here from the details of the Racket-Asip library, as the key point here is the last line: the students defined a list of two functions with `(list (moveLeft) (moveRight))` and then used `list-ref` to get one element from this list at a position which is randomly 0 or 1, depending on the result of `(random 2)`. They independently came up with a neat solution, and were clearly thinking ‘in a functional style’ when defining a list of functions. There is, however, a problem with the code above; running it causes the robot to move both left and right, as both functions `moveLeft` and `moveRight` are executed. This led to an interesting seminar discussion that helped to deepen students’ understanding in several areas. The problem was that writing `(list (moveLeft) (moveRight))` produces a list that contains the *result* of invoking `moveLeft` and `moveRight`; Racket’s eager evaluation means that both arguments to the function list are evaluated before it is applied. The functions have the side effect of printing to the screen. The contents of the list are the void values returned by the two functions (because `printf` returns void), and as a result `list-ref` chooses a random value from a list of voids. The solution provided at the end of the discussion is to build a list of references to the functions `moveLeft` and `moveRight`, rather than applications of them, by removing the brackets around them:

```

1 (list-ref (list moveLeft moveRight) (random 2))

```

This code will sometimes return a reference to `moveLeft`, and sometimes a reference to `moveRight`. To execute this reference, we need to surround the `list-ref` command with another pair of brackets.

```

1 ((list-ref (list moveLeft moveRight) (random 2)))

```

The point is that this idea was generated by the students’ own desire to make their robot do something interesting. Without this motivation, it is unlikely that they would have explored the concepts in sufficient detail to produce their proposed solution and, in turn, stimulate further discussion about how to make it work, which deepened their understanding of side effects and the difference between a symbol and its evaluation.

3.2 Using higher order functions

There are a number of instances in the projects where students may apply the programming concepts they have learned. One concept that many students find challenging is higher order functions. There are a number of possible ways to deploy higher order functions in controlling a robot with Racket, which

enable students to see the concept applied in practical situations. As an example, we will consider how the Racket client for ASIP can be used to process Arduino analog input pins. The relevant input message received from the Arduino is a string of the following form:

```
@I,a,3,{0:320,1:340,2:329}
```

This indicates that these are analog pins, 3 of which are set, pin 0 to 320, pin 1 to 340 and pin 2 to 329. A vector `ANALOG-IO-PINS` is defined to hold the values of the pins:

```
1 (define MAX_NUM_ANALOG_PINS 16)
2 (define ANALOG-IO-PINS (make-vector MAX_NUM_ANALOG_PINS))
```

and the code to update the vector is as follows:

```
1 (define (process-analog-values input))
2
3 (define analogValues (string-split (substring input
4                                     (+ (str-index-of input "{") 1)
5                                     (str-index-of input "}") ) ","))
6
7 (map (lambda (x) (vector-set! ANALOG-IO-PINS
8                             (string->number (first (string-split x ":")))) ;; the pin
9                             (string->number (second (string-split x ":")))) ;; the value
10                            ) ) ;; end of lambda
11    analogValues) ;; end of map
12 (printf "The current value of analog pins is: ~a \n" ANALOG-IO-PINS)
13 ) ;; end process-analog-values
```

First we obtain the substring of the input message between the braces (`str-index-of` is defined below) and split to obtain the list `analogValues` of the form ("`0 : 320`" "`1 : 340`"...). We then map a function to set an analog pin to a given value, over the list of pin/value pairs. `str-index-of` is a utility function to find the index of a character x in a string str ; x needs to be a string although we only look for its first character.

```
1 (define (str-index-of str x)
2 (define l (string->list str))
3 (for/or ([y l] [i (in-naturals)] #:when (equal? (string-ref x 0) y)) i))
```

Working with the above gave students further practice both with the imperative features of Racket and with higher order functions and string processing. Some students would develop a deep understanding of the code, while others might gain a superficial understanding sufficient to use the code. The main benefit was for students to get used to working with and taking advantage of code that they hadn't written and that tested their ability to learn and to understand; in other words, to get a feel for programming in the real world.

As a further example, here is some code using `map` in a simple control loop to print the value of the robot's IR sensors every 3 seconds and to print when the bump sensors are pressed or released:

```
1 (define previousTime (current-inexact-milliseconds))
2 (define currentTime 0)
3
4 ;; How often should we print?
5 (define interval 3000)
6
7 ;; The list of IR sensors (numbered 0,1,2 and used in map below)
8 (define irSensors (list 0 1 2))
```

```

9
10 (define previousLeft #f)
11 (define previousRight #f)
12
13 (define (controlLoop)
14
15   (set! currentTime (current-inexact-milliseconds))
16
17   ;; Print IR values
18   (cond ( (> (- currentTime previousTime) interval)
19         ;; We use map to print the value of each sensor
20         (map (lambda (i) (printf "IR sensor ~a -> ~a; " i (getIR i)))
21              irSensors)
22         (printf "\n"))
23         (set! previousTime (current-inexact-milliseconds))
24         )
25   ) ;; end of print IR
26
27   (cond ( (not (equal? (leftBump?) previousLeft))
28         ;; Something has changed for the left bump
29         ;; Just two cases: either it has been pressed, or released
30         (cond ((leftBump?) (printf "Left bump pressed\n"))
31               (else (printf "Left bump released\n"))
32               )
33         )
34   ) ;; end of cond for left bump changed
35
36   (cond ( (not (equal? (rightBump?) previousRight))
37         ;; Something has changed for the right bump
38         (cond ((rightBump?) (printf "Right bump pressed\n"))
39               (else (printf "Right bump released\n"))
40               )
41         )
42   ) ;; end of cond for right bump changed
43
44   ;; Set the state before iterating
45   (set! previousLeft (leftBump?))
46   (set! previousRight (rightBump?))
47
48   (sleep 0.02)
49
50   ;; A little trick to exit when both bump sensors are pressed
51   (cond ((not (and (leftBump?) (rightBump?)))
52         (controlLoop)
53         )
54   )
55 )
56
57 (define (minimalLoop)
58   (open-asip)
59
60   ;; let's take things easy...
61   (sleep 0.2)
62   (enableIR 100)
63   (sleep 0.2)
64   (enableBumpers 100)

```

```

65
66 ;; half a second to stabilise
67 (sleep 0.5)
68
69 (controlLoop)
70 (close-asip)
71 )

```

The students also become familiar with the trial and error aspects of programming with real-time systems, such as the need for the sleep commands in the above code.

Other higher-order functions can also be employed by students in robotic applications. For example, an application might log a list of the moves a robot makes in exploring an environment under some algorithm such as that in Section 3.1. Filter functions might then be used with predicate arguments to extract interesting data, such as the number of right turns or the number of straight paths taken for more than a given time before hitting a wall. Fold functions might be used to process the data in a number of ways. The following examples show how students might use `map`, `filter` and `foldr` in working with the robots.

Firstly, let us suppose that we want to read some Arduino input pins and find out how many of them are set to high. The following code fragments illustrate this.

```

1 ;defined in AsipMain.rkt
2 (define HIGH 1)
3 (define LOW 0)
4
5 (define INPUTPINS (list 2 3 4))
6
7 ;replace pin numbers with pin values
8 (map (lambda (i) (digital-read i)) INPUTPINS)
9
10 ;count HIGH values
11 (length (filter (lambda (i) (= i HIGH)) INPUTPINS))
12
13 ;alternative count using foldr
14 (foldr + 0 (filter (lambda (i) (equal? i HIGH)) INPUTPINS))

```

As a further example of the use of `foldr`, we return to the code that printed the values of the IR sensors at 3 second intervals and modify it so that instead of printing the IR values, they are accumulated in a list:

```

1 ;list of IR values, initially empty
2 (define IRlog (list))
3
4 ;; snippet modified to Log IR values
5 (cond ( (> (- currentTime previousTime) interval)
6         ;; We use map to add IR values to a list
7         (map (lambda (i) (cons (getIR i) IRlog)) irSensors)
8         (set! previousTime (current-inexact-milliseconds))
9         )
10 )

```

The list could then be processed with `foldr` to find out things such as the sum of those IR readings greater than some threshold value.

```

1 (define sumIRgreaterthan (lambda (threshold)
2 (foldr

```

```

3         (lambda (x y) (cond
4             ((> x threshold) (+ x y))
5             (#t y)))
6         0 IRlog)))

```

3.3 Contracts

A contract in Racket is a promise that a developer makes about a piece of code. Racket contracts are typically defined for modules [11], collections of definitions that are then used by other Racket programs using the construct `(require modulename.rkt)`. The `(provide [...])` block is used to specify the definitions that are accessible when the module is included with a `(require)` statement. The role of contracts is explored by students first in a non-physical context (the creation of a bank account module), and then in the cyber-physical context of MIRTO to determine requirements on the robot's behaviour, for example its speed with respect to the hardware specification:

```

1 (provide (contract-out ;; Begin of contract
2         [speed (and/c number? exact-nonnegative-integer?)]
3         [added_speed (-> checkSpeed any)]
4         [current_speed (-> number?)]
5         ) ;; End of contract
6     )
7
8 ;; We start from an initial speed of 0
9 (define speed 0)
10
11 ;; added speed takes a value and adds it to the initial speed
12 (define (added_speed value) (set! speed (+ value speed)))
13
14 ;; current speed returns the new value of speed
15 (define (current_speed) total)
16
17 (define checkSpeed
18     (lambda (a)
19         (and (number? a) (integer? a) (exact? a)
20              (and (>= (+ a total) -255) (<= (+ a total) +255))
21              )
22     )

```

We shall justify briefly in Section 5 the value of this specific construct for the purposes of learning programming.

4 Robotic examples by students

The final project work for the year consisted of the design of interesting applications for the robots, which some students tackled with much skill and imagination. One team had their robot 'race' against falling dominos, following identical paths (<https://www.youtube.com/watch?v=RnzDDdNOB14>). Another team implemented a PID algorithm that used the values of the robot's three IR sensors to follow lines drawn on a surface (<https://www.youtube.com/watch?v=VKXLM4av54o>); another team created two robots of their own and entered them in the Eurobot national championships in April 2015, coming 4th out of 17 teams (<https://youtu.be/o8b63XqIg5Y>). Such achievements were unheard of in the predecessor of the current CS programme, and much of this success is the result of the motivation instilled

in the students by the opportunity to apply their developing knowledge and skills to real-world problems through using the robots.

In the line-following project, students started studying the design principles of open- versus closed-loop systems, to understand how to feedback values from sensors in the code for other actuators. This was followed by the study of mathematical principles to design first a “bang-bang” line-following algorithm, then improved to a proportional controller to change the speed of the wheels, finally extended to a proportional-integral-derivative controller. At least one team of students did extended testing, both of the code and of its execution on MIRTO to find the optimal setting for various tracks, see <https://www.youtube.com/watch?v=VKXLM4av54o>. Here below we present their code, construed around the various functions of the IR-sensors and PID-controller that we helped them define. This code was developed autonomously by the team, without any external help from the tutors.

```

1 (define previousTime ( current-inexact-milliseconds ))
2 (define currentTime 0)
3
4 (define interval 10)
5
6 ;; The list of IR sensors (used in map below)
7 (define irSensors (list 0 ))
8 (define irSensors1 (list 1))
9 (define irSensors2 (list 2))
10
11 (define (irLoop)
12   (set! currentTime ( current-inexact-milliseconds ))
13   (cond ( (> (- currentTime previousTime ) interval )
14         (set! previousTime ( current-inexact-milliseconds ))
15         )
16   )
17   (irLoop))
18
19 (define (IRsweg a b c)
20   (define curRightCount (getCount 0))
21   (define curLeftCount (getCount 1))
22   (define (searchLoop)
23     (set! curRightCount (getCount 0))
24     (set! curLeftCount (getCount 1))
25     (cond ( (or (< 45 (getIR a)) (< 45 (getIR b)) (< 45 (getIR c))) (stopMotors))
26           ( (or (>= curRightCount 16) (<= curLeftCount -16)) (stopMotors)
27             (sleep 0.1)
28             (setMotors -115 -115)
29             (sleep 0.1)
30             (cond ((or (< 45 (getIR a)) (< 45 (getIR b)) (< 45 (getIR c))) (stopMotors))
31                   (#t (searchLoop) )
32                   )
33             )
34     (#t (printf "~a ~a\n" (getCount 0) (getCount 1)) (searchLoop))
35     )
36   )
37
38 (define (search)
39   (resetCount 0)
40   (resetCount 1)
41   (setMotors 115 115)
42   (sleep 0.1)

```

```

43   (searchLoop)
44   )
45
46   (cond
47     ((and (> 45 (getIR a)) (> 45 (getIR b)) (> 45 (getIR c))) (search))
48     ((and (< 45 (getIR a)) (< 45 (getIR b)) (< 45 (getIR c))) (setMotors -115 115))
49     ((and (< 45 (getIR a)) (< 45 (getIR b)) (> 45 (getIR c))) (setMotors 0 115))
50     ((and (> 45 (getIR a)) (< 45 (getIR b)) (< 45 (getIR c))) (setMotors -115 0))
51     ((and (< 45 (getIR a)) (> 45 (getIR b)) (> 45 (getIR c))) (setMotors 0 115))
52     ((and (> 45 (getIR a)) (> 45 (getIR b)) (< 45 (getIR c))) (setMotors -115 0))
53     (#t (setMotors 0 0)))
54   (IRsweg a b c)
55   )
56
57   (define cIR
58     (lambda (i)
59       (cond ( (> (getIR i) 45)
60             (getIR i)
61             )
62             (else 0)
63             )
64       )
65     )
66
67   (define oldError 0)
68   (define speed 150)
69   (define sumError 0)
70   (define (IRsweggier a b c)
71     (define Kp 0.05)
72     (define Kd 0.045)
73     (define Ki 0.007)
74     (define currentError 0)
75     (cond [ (> (+ (cIR a) (cIR b) (cIR c)) 0)
76           (set! currentError (/ (+ (mult 0 (cIR a)) (mult 2000 (cIR b)) (mult 4000 (cIR c))))
77           [else
78             (cond [ (> oldError 2000)
79                   (set! currentError 4000)]
80                 (else (set! currentError 0))
81                 )
82             ]
83           )
84     (define correction (inexact->exact (round (+ (mult Kp (- currentError 2800))
85                                                  (mult Kd (- currentError oldError)) )
86                                                  (mult Ki sumError)
87                                                  ))))
88     )
89     (displayln currentError)
90     (displayln correction)
91
92     (cond
93       ((< correction 0) (setMotors (- (+ speed correction)) speed)
94        )
95       ((> correction 0) (setMotors (- speed) (- speed correction))
96        )
97       (#t (setMotors (- speed) speed))
98     )

```

```

99   (sleep 0.02)
100  (set! oldError currentError)
101  (cond
102    ((and (> currentError -400) (< currentError 400)) (set! sumError 0))
103    (else (set! sumError (+ sumError (- currentError 2000))))
104  )
105  (IRsweggier a b c)
106  )

```

5 Related literature

The principles at the basis of our First Year BSc in Computer Science highlighted in Section 1 reflect much of the current literature in pedagogy, where we broadly follow a fine-grained, outcome-based learning path model. The theoretical implications remain to be assessed in their full meaning, especially for the pedagogical support; see [15] for a recent overview. However, this approach also follows professional guidelines and advice from industry. For example, the ACM/IEEE 2013 CS 2013 Curricula [1, p.28] in section 4.1 discourages

“to associate each Knowledge Area with a course [...] even though many curricula will have some courses containing material from only one Knowledge Area or, conversely, all the material from one Knowledge Area in one course. We view the hierarchical structure of the Body of Knowledge as a useful way to group related information, not as a structure for organizing material into courses. Beyond this general flexibility, in several places we expect many curricula to integrate material from multiple Knowledge Areas”.

The structure of our course reflects precisely this principle and, albeit we could have used a purely imperative approach to obtain the same final results in terms of robotic applications, we have chosen to emphasise functional programming, a topic represented as one of the first three Knowledge Units in the ACM/IEEE CS Curricula. In our approach, we try to cover all the Core-Tier1 and Core-Tier2 Topics from the Curriculum, while our assessment methodology (see [2]) focuses explicitly on all of the Core-Tier1 and Core-Tier2 Learning Outcomes.

However, the essential role of functional programming in teaching is highlighted not only in the academic context. In a very recent contribution [5], it is stressed how programmers should be exposed as early as possible to functional programming also as a way to gain exposure in declarative language abstractions, and how this principle is highly appreciated in the industry. This nicely complements the richness of functional constructs (with their imperative flavour mentioned at the beginning of Section 2) that Racket allows us to teach to our students.

Another important recommendation from [5] for teaching programming concerns “design by contracts” as a way to refer to annotations made in the program to express what the program (or part of it) is supposed to accomplish, as opposed to how it should compute. This technique falls in the larger and more essential issue of educating the future generation of computer scientists and programmers with

“a grounding in logic, its application in design formalisms, and experience the creation and debugging of formal specifications” [5, p.31].

Besides our coverage of formal topics (including functions, relations, set theory, regular expressions, propositional and predicate logic) and our design workshops (in which essential topics such as UML and (extended) finite state machines are introduced), we implement directly the design by contracts principle in Racket, as illustrated in section 3.3.

There are other works dealing with the subject of robots and functional programming. The approach used in [7] and [14] is quite similar to ours, though the former uses functional programming to teach robot operation, rather than the other way around. The approach in [9] is more advanced and uses functional reactive programming. This is a concept that, while interesting, we feel is not a topic for a basic first year programming course.

6 Conclusion

We have introduced an overview of how the use of Racket to drive the MIRTO robot lends itself to teaching functional patterns to first year students, and the role of these in our first year Computer Science curriculum.

Our chief goals in using robots in the curriculum were fivefold.

- To teach some real-time robotics programming.
- To reinforce the learning of functional and imperative programming
- To help students to develop their practical group-working and project skills
- To enable students to reinforce and integrate the knowledge and skills acquired in other parts of the curriculum
- To encourage the students to explore beyond the confines of their programme of study, by competing with other students within the university and from other universities.

The programming ranged from the application of functional programming concepts, such as higher-order functions, lists of functions and vectors, to the use of low-level imperative programming such as insertion of delays. The latter could have been hidden behind abstractions but exposing the students to such concepts directly gave them a broader appreciation of the many facets of practical problem solving in programming.

In essence, we have used functional programming as a tool not only to reinforce the teaching of most aspects of the curriculum, but also to introduce students to robotics applications. On the other hand, we have also used robotics to deepen students' knowledge and skills in functional programming. It is true that, in terms of final results for the implementation of robotic applications, all our functional patterns could have been converted to imperative ones. However, students are excited by working with the robots, and this excitement tends to make them engage more and thereby to achieve more in a topic (functional programming) that is normally considered "theoretical". The ability to motivate is undoubtedly one of the most important aspects of the robotics projects, especially when compared with the rather dry applications normally included in first-year and functional programming courses. At the time of writing, two cohorts of students have passed through the first year of the programme, and pass rates for those students completing the year were over 90% in each case. We believe this success to be due in no small part to the motivating influence of the practical projects undertaken by the students.

References

- [1] ACM/IEEE-CS Joint Task Force on Computing Curricula (2013): *Computer Science Curricula 2013*. Technical Report, ACM Press and IEEE Computer Society Press, doi:10.1145/2534860. Available at <http://dx.doi.org/10.1145/2534860>.

- [2] K. Androutsopoulos, N. Gorogiannis, M. Loomes, M. Margolis, G. Primiero, F. Raimondi, P. Varsani, N. Weldin & A. Zivanovic (2014): *A Racket-Based Robot to Teach First-Year Computer Science*. In: *Proceedings of the 7th European Lisp Symposium*, pp. 54–62.
- [3] *Racket Asip Client Library*. Available at <https://github.com/fraimondi/racket-asip>.
- [4] *Atmel Studio*. Available at http://www.atmel.com/microsite/atmel_studio6/.
- [5] Thomas Ball & Benjamin Zorn (2015): *Teach Foundational Language Principles*. *Communications of the ACM* 58(5), pp. 30–31.
- [6] Mirco Bordoni, Michele Bottone, Bob Fields, Nikos Gorogiannis, Michael Margolis, Giuseppe Primiero & Franco Raimondi (2015): *Towards Cyber-Physical Systems as Services: the ASIP Protocol*. Workshop Paper, Ocado Group & Middlesex University. Available at <http://www.rmnd.net/pubs/sescps15.pdf>. To be presented at International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS) Collocated with ICSE 2015.
- [7] Thomas Paul Carlson: *Robotlang-A Concurrent Functional Programming Language For Robots*. *Integers* 1, p. 42.
- [8] David Harel & P.S. Thiagarajan (2003): *Message Sequence Charts*. In Luciano Lavagno, Grant Martin & Bran Selic, editors: *UML for Real*, Springer US, pp. 77–105.
- [9] Paul Hudak, Antony Courtney, Henrik Nilsson & John Peterson (2003): *Arrows, Robots, and Functional Reactive Programming*. In: *Summer School on Advanced Functional Programming 2002, Oxford University, Lecture Notes in Computer Science* 2638, Springer-Verlag, pp. 159–187.
- [10] *The Middlesex Robotic platfOrm (MIRTO)*. Available at <https://github.com/fraimondi/myrtle>.
- [11] *The Racket Guide*. Available at <http://docs.racket-lang.org/guide/module-basics.html>.
- [12] Creative Robotics: *HUB-ee*. Available at <http://www.creative-robotics.com/About-HUBee-Wheels>.
- [13] Hans-Christoph Steiner (2009): *Firmata : Towards Making Microcontrollers Act Like Extensions of the Computer*. In Noel Zahler, Roger B. Dannenberg & Tom Sullivan, editors: *Proceedings of the International Conference on New Interfaces for Musical Expression*, Pittsburgh, PA, United States, pp. 125–130. Available at http://www.nime.org/proceedings/2009/nime2009_125.pdf.
- [14] David Wakeling (2008): *A Robot in Every Classroom: Robots and Functional Programming Across the Curriculum*. In: *Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education, FDPE '08*, ACM, New York, NY, USA, pp. 51–60, doi:10.1145/1411260.1411268. Available at <http://doi.acm.org/10.1145/1411260.1411268>.
- [15] Fan Yang, Frederick W. B. Li & Rynson W. H. Lau (2014): *A Fine-Grained Outcome-Based Learning Path Model*. *IEEE T. Systems, Man, and Cybernetics: Systems* 44(2), pp. 235–245, doi:10.1109/TSMCC.2013.2263133. Available at <http://dx.doi.org/10.1109/TSMCC.2013.2263133>.