Abstrakt In this note we critically examine the current state of teaching of Discrete Simulation at our university. We consider the case study of one of the projects that students are expected to solve in the course. Three approaches are compared - nonmodular resulting from the current approach, using Haskell, and using Reactive framework from C#. We conclude that functional approach can address problems that a teacher may encounter, although a full-fledged functional language may be unnecessary. This is in agreement with general trends in software industry.

1 Introduction

University of Žilina is situated at an industrial hub of northernwestern Slovakia. Historically, its goals were to prepare engineers for transportation and manufacturing industries. With decreasing economic role of heavy industries in the neighbouring region, its education plans became more varied and software engineering is one of the newer and fast-growing specialties of the university. In this note we will concentrate on students enrolled in our “Information systems” program.

Software simulation is considered by our department an important area of application of software engineering skills. Our program incorporates courses both at the undergraduate level and the engineering level. At the undergraduate level students attend a one term course about simulation models and learn to use Arena software to develop models. At the engineering level students are expected to master writing simulation models, first event based and then agent based models. Implementation is complemented by planning and execution of experiments, and preparation of preliminary and final analysis. The final problem that students solve is sufficiently complex to demonstrate the complete lifecycle of simulation analysis.

The engineering level course is considered challenging by many students. The course completion rate is about 70%, despite the fact that the course is typically taken in the fourth year of study.

In this note we examine the possibilities of using functional programming in Discrete Simulation course.

2 Functional programming

Theoretical basis of functional programming can be traced to work of Alonso Church in 1930’s. He developed λ calculus which was shown to be equally powerful as the Turing machine model of computation. LISP, introduced in 1958, is the oldest widely used functional programming language. Its dialects (e.g. Scheme) are still commonly used in computer science undergraduate courses.

Functional languages have been consistently significantly less popular than imperative languages (Fortran, Cobol, Pascal, Modula-2, C, C++, Java, C#), perhaps due to its heavy dependence on efficient garbage collection. With the advent of multicore computing many guessed that this would change, because functional programming appears more suitable for developing parallel programs. However, alternative technologies, from CUDA through OpenMP to MapReduce have so far kept the most widely used languages equally suitable for developing parallel software.

Yet, functional languages scored some indisputable victories. Erlang programming language used by Ericsson allows the company to manufacture telecommunication systems with 99.999% uptime. Popular open source statistics software R uses the functional language R for programming. Functional constructs have started to appear in mainstream imperative languages. For instance anonymous functions, a λ calculus construct, can be now found in C#, Python, PHP, Matlab, Mathematica and are to appear in later versions of Java and C++. LINQ, a popular Microsoft.NET feature, is a also functional programming construct. Finally, we should also mention that JavaScript is considered to be a functional programming language. Let us briefly summarize main features of functional programming.

2.1 Features of functional programming

Functions are first-class values. That means that they can be assigned, passed as arguments and returned as values from functions. This is stronger than just passing and returning C-style function pointers, because functions can use variables from their originating
context. Thus returning a function from a function in effect means returning both a function pointer and state of some variables.

Immutability of variables. Variables are expected not to change, much like the behavior of C++ variables with const modifier. The obvious benefit is that it is much easier to argue about value of variables, since they are not affected by time. Obviously, exceptions are allowed, but as a rule it is much easier to have variable bindings that do not change their values (exactly unlike C++, where it is much easier to have non-const values than const ones).

Pure functions. Similarly, functional programming prefers pure functions, that is functions, that return the same values for the same inputs. (In relational databases such functions are called deterministic). Pure functions allow for more powerful compiler optimizations.

Recursion. With variables being constant it is impossible to carry out typical iteration loops. An obvious alternative is using the recursion, the disadvantage being possible stack overrun. When the recursion call is the last call in the function then it can be replaced by a simple jump that reclaims the previous state. This optimization, called tail recursion is crucial for efficiency of functional programming and many languages provide explicit guarantees to its availability. Functional programming also uses higher order constructs to replace recursion.

Sophisticated type system. Algebraic data types, commonly needed in compiler development, are a hallmark of functional programming languages. They allow for precise specification of types unlike the classical C union. Their usage allows the compiler to warn about missing pattern matching cases.

Moreover, compilers for functional languages are characteristic by being able to infer correct types of variables. This significantly shortens functional programs, since type annotations account for most of lines of a typical imperative source code. Some argue, this increases programmers’ productivity. It is however indisputable that type systems in functional languages are quite advanced, being able to express constructs not found in most of widely used programming languages.

3 A case study

Let us describe a mid-term problem given this year to our students. The goal is to simulate a supermarket with three service departments. A customer enters the market and spends time shopping for some freely displayed items. Then she goes to vegetable and fruit department, waits in a line, if necessary, and buys some fruits and vegetables. Then she proceeds to meat department, again possibly waits in a line, and buys some meat items. Finally she proceeds to cashiers, waits in a final line, scans the items pays and goes out. For each of the activities a known probability distribution is given that governs its behavior. The goal is to investigate aggregate statistics depending on the number of service agents in various departments (vegetables & fruits, meats and cashiers).

Let us remark that both data structures and algorithms needed are of moderate difficulty. Obviously, a student will need to use arrays of queues to model customer lines at various departments. The most advanced data structure needed is the priority queue for holding events in sorted order. Two classical pedagogical problems may be expected to occur:

Problem A For students that struggle, it may be hard to find out exactly where they make errors.

Problem B For students experienced in programming, the problem may be too easy, and they may lose interest in participation in class.

The first problem typically affects a bigger proportion of students. Since students work on the assignment individually, their designs, implementations and even programming languages vary. It thus appears the biggest payoff would be in providing “behavior assertions” that would allow students to determine the location of their error. In is easy to see that the proposed problem is quite amenable to such hints. If a student errs only in implementing meat processing department behavior, any aggregate statistics for fruit and vegetables would still be valid. It is only later stage statistics (meat department, cashiers) that would be incorrect. This guidance is of course only available if students are able to readily modify their programs to produce a wide variety of statistics. It is a problem in itself how to structure programs so that they are versatile and can be easily adapted to production of various statistics. We state it as

Problem A’ Separation of simulation itself from collection and maintenance of statistics.

We have examined these three problems from several perspectives.

3.1 Student’s solutions

During lectures and exercises students were taught about how to program event based simulation in Pas-
cal. To judge their experience developing solutions we have administered a survey at the end of the course. The sample included 35 students. Figure 1 shows languages chosen by students to solve the assignment.

Obr. 1. Language used by students to implement their solutions

In the survey, more than 2/3 students indicated that the biggest issue for them was to discover why the simulation produces incorrect results. Moreover, again 2/3 of students were unable to uncover the problem by themselves, and they had to consult their classmates to compare code and results. This indicates that Problem A is acute, and is not addressed by the current approach to instruction.

On the other end of the spectrum, a few (less than 10%) students indicated they were able to produce a solution in less than 3 days. This is a big difference from the majority of students (more than 60%) taking more than 7 days to produce a solution. This is an indication that Problem B also occurs.

3.2 Haskell

Haskell language is a functional language popular in academic sphere. We have decided to use this language because it is used by the leading functional programming researchers. For instance, in the past 15 years there has been significant research interest in developing Haskell libraries for functional reactive programming (FRP) starting from [1], [2]. This paradigm can be used to express both push and pull based models [3], for both continuous time functions (signals) and events.

We have decided to eschew using a specific FRP library, partly because of unsettled standard, and partly because we felt Haskell itself would be sufficiently challenging to pick up for students in our program. In fact, the latter reason makes Haskell a good choice for addressing Problem B. Many programmers feel that learning Haskell was the most challenging endeavour in their skills development.

Let us consider how to address Problem A'. In principle the are two ways to go around doing it. The first is to pass an aggregation or processing function to the simulation core. The core then has the responsibility to call this function whenever event occurs. The function would typically maintain some state, although functions that simply trace out events are also quite handy. Another solution is to expose event stream to a processing operator. The former approach would result in signature for the main simulation function as follows:

$$sim :: (\text{Int}, \text{Int}, \text{Int}) \rightarrow (a, a \rightarrow \text{Event} \rightarrow a) \rightarrow \text{IO Simulation a},$$

where the triple indicates number of people servicing different departments. The latter approach could result in less general type signature

$$sim :: \text{System.Random.RandomGen} g \Rightarrow (\text{Int}, \text{Int}, \text{Int}) \rightarrow g \rightarrow [\text{Event}]$$

Note that Haskell’s lazy evaluation would be crucial for efficient implementation of this approach, because the event list should not ever be fully evaluated as it can be too large to store in memory.

Although the latter approach potentially yields pure function code, we decided to implement the former approach. The main factor was that available priority queue implementation required monadic code, and thus monadic approach seemed more natural. (We should note that Arrowized FRP (AFRP) was proposed [5] and has certain advantages over monadic approaches.) The resulting code had about 320 lines, which was less than half of code needed for solutions written by students. Number of textual symbols used in Haskell was 181, slightly less than average of 228 used by students.

Despite that we feel that Haskell failed our (possibly too high) expectations. The resulting code was significantly slower. It used too much memory, most likely leaking at some place. We have been unable to profile the code because standard installation failed to profile priority queue implementation. Also Haskell lacks good standard graphics libraries. Taken altogether, the Haskell Platform [4] appears immature compared to commercially used environments. The undisputable benefits were concise code, that was easy to verify and almost directly reflected the requirements. Finally Haskell’s type system proved its superb reputation, allowing to concisely express the program’s logic. The code was very flexible allowing for instance easy change of time from integral to float type.
Our department has a few years back settled on Java as the main programming language for students. It is thus quite surprising that C# proved to be, by a wide margin, the most popular choice for solving the assignment. It is probably caused by confluence of multiple factors: excellent quality of developer tools provided by Microsoft, Microsoft’s licensing for students and academic users, and popularity of Microsoft’s technologies in industry.

Microsoft’s Visual Studio supports multiple languages, C#, F# and Visual Basic.Net. Visual Basic appears to assume legacy rôle, and is being surpassed by C#. F# is a functional language, derived from O’Caml. It may be somewhat surprising but looking for functional approach to discrete simulation we have opted for C# instead of F#.

The reason is that C# has been regularly extended to support latest language research developments. Moreover, Microsoft strives to make .NET languages interoperable, and to a certain degree interchangeable. Microsoft also appear to strive to learn from the best academic research results, employing for instance both C. Elliott, the leading researcher in FRP, and Simon Peyton Jones, a leader in Glasgow Haskell Compiler development. It is thus no surprise that functional reactive programming, originally conceived in Haskell, found its way to Reactive C# framework [6].

This framework is sometimes referred as LINQ for events. Let us recall the motivation for LINQ. There is a well-known impedance mismatch between OOP and relational databases. One of its manifestation is in a wall separating SQL and application code, which makes it impossible for the compiler to comprehend data communication. This can lead for instance to SQL injection attacks. Microsoft realized that using functional constructs could solve this problem. It allowed to write declarative commands in their applications that could be parsed and combined by the compiler. SQL queries were suddenly exposed and any errors in them could be detected in compile time. The core behind LINQ is IEnumerable interface.

```java
public interface IEnumerable<T>
    : IEnumerable
{
    new IEnumerator<T> GetEnumerator();
}

public interface IEnumerator<T>
    : IDisposable, IEnumerator
{
    new T Current { get; }
    bool MoveNext();
}
```

By dualizing these interfaces one arrives at IObservable and IObserver interfaces, and this is precisely what Reactive framework uses.

```java
public interface IObservable<T>
{
    void Subscribe(IObserver<T> observer);
}

public interface IObserver<T>
{
    void OnCompleted();
    void OnNext(T value);
    voidOnError(Exception error);
}
```

A developer now can write declarative statements that process stream of events. Since LINQ is a technology that is well known to our students (over 40% used it, despite it not being in curriculum), we feel that Reactive framework is well within grasp of majority of our students and will allow us to address Problem A’.

Moreover, since students themselves prefer C# by a large margin, this should also mean that Problem A will be addressed.

4 Conclusion

We have examined the current methodology for teaching Discrete Simulation at our university. Based on our research we’ve found that functional programming language in itself is not the answer to improve our teaching. But functional programming techniques, such as Reactive framework in C#, have the potential to adequately address the problems we identified. For the very brightest students, learning a functional programming language may be a challenging and worthwhile undertaking.

Unfortunately, we did not find a comparable solution for Java. But functional constructs are to appear in future versions of Java, and functional languages derived from Java and JVM are being developed (Scala, Clojure, Ceylon) thus we may expect similar functionality to be available on Java platform as well.

Literatúra

6. Get Started Developing with the Reactive Extensions