DE GRUYTER OPEN
RIGA TECHNICAL UNIVERSITY
Applied Computer Systems
ISSN 2255-8691 (online)
ISSN 2255-8683 (print)
December 2017, vol. 22, pp. 47–53
doi: 10.1515/acss-2017-0017
https://www.degruyter.com/view/j/acss

Implementation of a MIX Emulator: A Case Study of the Scala Programming Language Facilities

Ruslan Batdalov[1], Oksana Ņikiforova[2]
[1, 2] *Riga Technical University, Latvia*

***Abstract* – Implementation of an emulator of MIX, a mythical computer invented by Donald Knuth, is used as a case study of the features of the Scala programming language. The developed emulator provides rich opportunities for program debugging, such as tracking intermediate steps of program execution, an opportunity to run a program in the binary or the decimal mode of MIX, verification of correct synchronisation of input/output operations. Such Scala features as cross-compilation, family polymorphism and support for immutable data structures have proved to be useful for implementation of the emulator. The authors of the paper also propose some improvements to these features: flexible definition of family-polymorphic types, integration of family polymorphism with generics, establishing full equivalence between mutating operations on mutable data types and copy-and-modify operations on immutable data types. The emulator is free and open source software available at www.mix-emulator.org.**

***Keywords* – MIX, program correctness verification, Scala.**

I. Introduction

MIX is an imaginary computer invented by Donald Knuth for his book series "The Art of Computer Programming" [1], which became one of the most famous treatises in computer science, in general, and in the analysis of algorithms, in particular. Despite the necessity to program in the assembler language and the old-fashioned architecture, typical of the computers of the 1960s, MIX still preserves its value for studying algorithms and their analysis. Therefore, an opportunity to have a software emulator of MIX is helpful in mastering "The Art of Computer Programming".

Due to its outdated architecture, MIX is planned to be replaced by a newer MMIX computer in future editions of Knuth's masterpiece [1], [2]. However, while this work is in progress, emulators of MIX still preserve their value. The approaches developed for them can also form the basis for developing newer MMIX emulators.

Existing MIX emulators typically allow running and debugging programs for MIX and counting the elapsed time [3]–[11]. However, their capabilities in checking program correctness are limited. According to Knuth's description, MIX can work as either a binary or a decimal computer, and a correct program should not depend on the byte size [1]. The existing emulators work in the binary mode only. Furthermore, the existing emulators typically do not check correctness of input/output operation synchronisation (for example, whether a program waits correctly for an operation to be completed before reusing memory). Since modern hardware allows emulating these operations in the synchronous manner, possible errors in a program may remain unnoticed. In the authors' opinion, these checks are useful in mastering how to write correct programs because similar errors often occur in a modern program despite all changes in hardware and software technologies. Therefore, it would be helpful if an emulator supported running programs in different modes and allowed checking that the execution result was the same in all cases.

The programming language chosen by the authors for the implementation of an emulator supporting these features is Scala. This choice is arbitrary to some extent and rather dictated by the authors' interest in the features of this language. Scala is a modern programming language, which combines the object-oriented and functional paradigms [12] and significantly advances the state of art in programming language research [13]. Novel features of Scala are extensively used in the authors' implementation of the emulator as discussed further. Therefore, the current research is also a case study of the mentioned features and insufficiencies in their support in Scala.

The goal of the research is to perform a case study of implementation of a MIX emulator in Scala and analyse benefits and drawbacks of the chosen approach. The main areas of interest in this analysis are the opportunities provided by the emulator for studying algorithms using Knuth's work, as well as the features of the Scala programming language.

The remainder of the paper is organised as follows: Section II discusses related approaches and similar tasks. Section III presents the description of the emulator developed in this study. Section IV is devoted to the analysis of the Scala programming language facilities with respect to this project, and Section V concludes the paper.

II. Background and Related Research

There are a number of different MIX emulators [3]–[11]. They differ in the target platform (e.g., native compilation for UNIX, Java Virtual Machine, .NET, and web), the subset of supported instructions and other properties. Typically, these emulators provide the common functionality of software debuggers: they allow a user to compile a program, execute it, set execution breakpoints, and inspect memory and register content. Obviously, this functionality is also required in the emulator under consideration.

Typically, debugging of a program (both in common debuggers and in the mentioned MIX emulators) is execution of a program with an opportunity to stop it at an arbitrary point and execute the program step by step. An interesting

©2017 Ruslan Batdalov, Oksana Ņikiforova.
This is an open access article licensed under the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0), in the manner agreed with De Gruyter Open.

alternative approach to debugging is proposed in Online Python Tutor [14]. The software stores the memory state after execution of each program statement and thus allows not only running the program in the forward direction, but also returning to the previous states. This feature facilitates debugging and understanding of the operation of the program. Therefore, it would also be useful for the discussed emulator.

Another related group of software consists of the virtual machines designed for executing intermediate code (bytecode), for example, Java Virtual Machine [15] or .NET Common Language Runtime [16]. However, these machines are primarily designed for performance and do not provide rich opportunities for running programs in different modes. Since the main goals of the considered emulator are related rather to careful studying of how a program works, the primary focus should be placed on the functionality instead of the performance.

Summarising, the main functionality of the emulator should be similar to the one of a typical debugger with an additional opportunity to return to the previous states. Verification of program work in the decimal mode and of input/output synchronisation correctness is not supported in the existing MIX emulators, but can be helpful in understanding of the program execution flow. In future, some extra functionality (for example, additional checks performed by advanced debuggers) may also be added to the product, but it is not strictly necessary.

III. Emulator Implementation

This section describes the authors' implementation of a MIX emulator according to the goals discussed above. First, the overall picture of the architecture is given, and then the main problematic issues of the implementation are discussed in further subsections.

A. Architecture Overview

Knuth described two main elements required to execute programs for MIX: the MIX computer itself and MIX assembly language (MIXAL) [1]. Therefore, it is logical to implement a virtual machine and an assembler in separate related modules. The assembler uses the virtual machine (because translation of MIXAL programs is defined in terms of MIX operations [1]), but not vice versa.

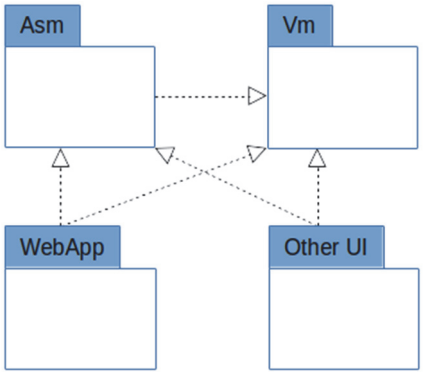

Fig. 1. Emulator building blocks.

Both the assembler and the virtual machine are used by the user interface, through which users work with the virtual MIX. The assembler and the virtual machine are independent of the user interface and may be used by multiple user interfaces. Moreover, different user interfaces can work in different run-time environments. At present, Scala officially supports two of them: Java Virtual Machine (JVM) and JavaScript. The opportunity to cross-compile the same source code for different platforms allows using the same assembler and virtual machine on both. Currently, the only user interface for the emulator is a web interface written in Scala and transpiled to JavaScript with Scala.js (Scala plugin for Google Web Toolkit is a possible alternative for this task). At the same time, the unit tests work in JVM due to the restrictions of the used testing library. JVM-based user interfaces for the emulator are planned in future.

The overall composition of the emulator is depicted in Fig. 1. Implementation of the assembler (provided that the virtual machine is already implemented) and the user interface is rather straightforward and is not discussed in this paper in detail. However, implementation of the virtual machine supporting the desired features is not trivial. The main problems and decisions related to the implementation are discussed in the following subsections.

B. Basic Data Types

According to Knuth's description, MIX consists of [1]:

- 4000 memory words of five bytes and a sign each,
- two registers (A and X) of five bytes and a sign each,
- six index registers (I1 to I6) of two bytes and a sign each,
- one register (J) of two bytes without a sign,
- overflow flag (Boolean value),
- comparison indicator (less, equal or greater),
- peripheral devices.

The following abstract data types are used in the authors' implementation of the emulator to represent these data and operations on them:

- `MixByte` to represent one byte,
- `MixIndex` to represent a memory address, the content of a two-byte register (I1 to I6, J) or the program counter (the address of the next instruction to execute),
- `MixWord` to represent the content of a five-byte register (A, X) or of a memory cell,
- `MixDWord` to represent the result of multiplication of two `MixWord`s.

However, implementation of these data types should take into account that the size of a byte in MIX is variable. If MIX is functioning as a binary computer, a byte can hold 64 different values, and, in the decimal mode, a byte can represent 100 different values [1]. A correct program must also support any intermediate byte size (for example, 81 different values for a ternary computer) [1]. The difference in the byte size requires using different Scala data types for storing data in different MIX modes.

Furthermore, the result of elementary machine operations also depends on the mode. For example, the sum of 10^9 and 10^9 fits into a MIX word in the decimal mode ($2 \times 10^9 < 100^5$), but triggers an integer overflow in the binary mode (because $2 \times 10^9 > 64^5$). Therefore, operations on the basic data types should also be implemented separately for each machine mode.

Thus, the set of basic data types (byte, index, word and double word) should be implemented for each MIX byte size. Scala offers a mechanism for implementing families of interdependent data types called family polymorphism [17]. Implementation of this mechanism requires [17]:

1. Declaring an abstract class that contains declarations of the abstract data types (abstract classes and/or traits) forming the abstraction of the type family. Martin Odersky et al. do not define a separate term for this class [17]. For convenience, it is referred to as the abstract model in the present article.
2. Declaring type variables that represent not yet defined concrete data types inside the abstract model. The abstract model declares that the types represented by the type variables inherit from the abstract data types, but does not specify them any further.
3. Defining common operations in terms of the declared abstract data types and type variables.
4. Declaring an object that contains concrete data types, implementing the previously defined abstract data types. In a manner similar to the abstract model, this object is further referred to as the concrete model. The concrete model assigns the concrete data types to the type variables declared in Step 2.
5. Implementing the declared operations on the concrete data types to complete the implementation.

For example, application of this scheme to MIX operation `add` (adding a memory word to register A) in the considered emulator looks as follows:

1. The abstract model (`DataModel`) declares, among other things, data types representing MIX words (`AbstractMixWord`), memory state (`AbstractMemoryState`) and register state (`AbstractRegisterState`). Memory and register state are discussed in more detail later.
2. The abstract model declares type variables `W`, `MS` and `RS`, which represent concrete subtypes of `AbstractMixWord`, `AbstractMemoryState` and `AbstractRegisterState`, respectively.
3. The abstract model defines operation `add` in terms of word retrieval from variables of type `MS` and `RS`, summation of two variables of type `W` and storing the result by means of a call to a variable of type `RS`. All these operations are declared in the corresponding abstract types and must be supported by types `W`, `MS` and `RS`, whatever these types are.
4. The concrete models (`binary` and `decimal`) declare concrete classes `{binary, decimal}.MixWord`, `{binary, decimal}.MemoryState` and `{binary, decimal}.RegisterState`. Type variables `W`, `MS` and `RS` are assigned with types `{binary, decimal}.MixWord`, `{binary, decimal}.MemoryState` and `{binary, decimal}.RegisterState`, respectively.
5. The concrete classes implement the operations declared in the abstract data types (word retrieval, word summation, and storing a word), so that the implementation of `add` is complete in each concrete model.

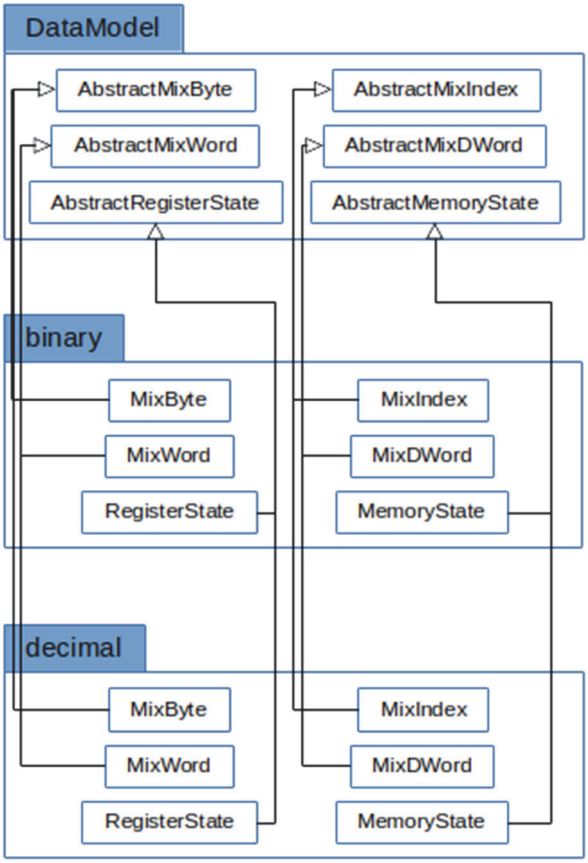

Fig. 2. Relationships between abstract and concrete data types.

The main benefit of this pattern in comparison with more common forms of object-oriented polymorphism is the opportunity to establish relationships between classes within a family. For example, it is known that the result of multiplication of two binary MIX words is a *binary* MIX double word, not just *some* double word. Therefore, family polymorphism in Scala is extremely useful for implementation of different modes of the MIX emulator.

The resulting relationships between the mentioned classes are shown in Fig. 2.

C. Implementation of Concrete Data Types

Since machine words essentially just represent numbers, an idea to emulate a MIX value with the numerically equal value of a modern data type is appealing. For example, a machine

word containing value 1 would be represented by a Scala integer containing value 1, etc. However, this scheme is problematic due to architectural differences. Unlike modern computers, which typically use the two's complement representation of negative numbers, MIX stores the sign in a separate bit of a word [1]. It means that `+0` and `−0` are separate (although numerically equal) values. Since there is no way to store value `−0` in a Scala integer variable so that it remained distinguishable from `+0`, it is necessary to use a representation that is closer to the one used in MIX itself. Therefore, a single bit of a `MixIndex` or a `MixWord` is used to encode the sign of a value (its value is 0 for positive values and 1 for negative ones) and the remaining bits are used for storing the absolute value of a number. The particular bits assigned for these purposes depend on the mode as discussed further.

In the binary mode, a byte can contain 64 different values, so six bits are enough to represent a `MixByte`. A `MixIndex` can be represented with $1 + 2 \times 6 = 13$ bits, a `MixWord` with $1 + 5 \times 6 = 31$ bits, and `MixDWord` with $1 + 10 \times 6 = 61$ bits. Therefore, standard Scala types `Byte` (8 bits), `Short` (16 bits), `Int` (32 bits) and `Long` (64 bits) can be used to represent the four basic data types in the binary mode. Each MIX byte occupies six consecutive bits. The resulting storage scheme is shown in Table I, where X – unused; U – used for storing byte value; S – sign bit; 0 to 9 – bit of byte 0, 1, etc. (starting from the most significant byte of the data type).

TABLE I

MIX DATA TYPE REPRESENTATION IN THE BINARY MODE

MIX Data Type	Scala Data Type	Bits Usage							
		7	6	5	4	3	2	1	0
`MixByte`	`Byte`	X	X	U	U	U	U	U	U
`MixIndex`	`Short`	X	X	X	S	0	0	0	0
		0	0	1	1	1	1	1	1
`MixWord`	`Int`	X	S	0	0	0	0	0	0
		1	1	1	1	1	1	2	2
		2	2	2	2	3	3	3	3
		3	3	4	4	4	4	4	4
`MixDWord`	`Long`	X	X	X	S	0	0	0	0
		0	0	1	1	1	1	1	1
		2	2	2	2	2	2	3	3
		3	3	3	3	4	4	4	4
		4	4	5	5	5	5	5	5
		6	6	6	6	6	6	7	7
		7	7	7	7	8	8	8	8
		8	8	9	9	9	9	9	9

In the decimal mode, a byte can contain 100 different values, so `MixByte` requires at least 7 bits. `MixIndex` requires $1 + \lceil \log_2 100^2 \rceil = 15$ bits, `MixWord` requires $1 + \lceil \log_2 100^5 \rceil = 35$ bits, and `MixDWord` requires $1 + \lceil \log_2 100^{10} \rceil = 68$ bits. Therefore, the same standard data types `Byte` (8 bits) and `Short` (16 bits) are used for implementation of `MixByte` and `MixIndex`, but `MixWord` is emulated with `Long` (64 bits), and `MixDWord` with a pair of `Longs` (128 bits in total). Unlike the binary mode, `MixBytes` in decimal `MixIndex`, `MixWord` and `MixDWord` are not positioned in separate groups of bits because it would significantly complicate calculations. Instead, their representation consists of a sign bit and the necessary number of bits for storing the absolute value. The absolute value of a `MixDWord` is stored in two parts: 10 most significant decimal digits of a 20-digit number are stored in the first `Long`, and 10 least significant digits are stored in the second `Long`. The resulting storage scheme is summarised in Table II, where bits are numbered starting from the least significant bit and the least significant bit is bit 0.

TABLE II

MIX DATA TYPE REPRESENTATION IN THE DECIMAL MODE

MIX Data Type	Scala Data Type	Sign Bit No.	Absolute Value in Bits
`MixByte`	`Byte`	–	6–0
`MixIndex`	`Short`	14	13–0
`MixWord`	`Long`	34	33–0
`MixDWord`	Pair of `Longs`	34 of the first `Long`	33–0 of both `Longs`

Operations on MIX data types (arithmetic operations, comparison, conversions, etc.) are implemented by means of similar operations on the underlying Scala data types. Implementation requires adapting the values to a differing representation, but it is a rather technical difficulty.

Other modes (intermediate byte sizes) are not supported currently, but can be added in a similar manner. Adding a mode requires designing a representation of the MIX data types with Scala data types and implementing the operations declared in the abstract data types.

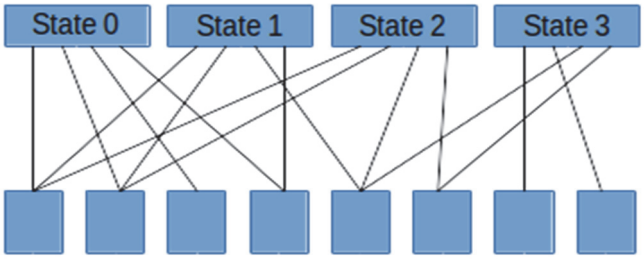

Fig. 3. Shared memory storage (schematically).

D. Machine State and State Sequence

Values stored in all memory cells and registers at a given moment form the state of the virtual machine at this moment. As discussed in Section 2, storing the states encountered in the course of program execution is helpful in program debugging. The amount of memory necessary to store all states can be big, but using immutable data structures allows reducing it significantly. Immutable data structures typically use shared memory storage, in which the common part of two data structures is stored only once. This concept is schematically

shown in Fig. 3. The actual storage model is typically more efficient than in Fig. 3 because storing a separate reference to each memory cell would create an excessive storage overhead. Instead, larger blocks and some duplication of unchanged data may be used for optimisation, but the general idea of the shared data storage is the same.

Being a functional language, Scala provides rich support for immutable data structures. Scala's standard library contains a variety of immutable data structures for different purposes (either immutable equivalents of conventional mutable data structures or new ones specifically designed as immutable) [18]. The language itself supports various operations that facilitate using immutable data structures, for example, the automatically defined method `copy` on case classes and expansion of the shortcut assignment operators (+=, −=, etc.) so that they work in the equivalent manner for both mutable and immutable data structures [12]. These operations provide a simple and concise syntax that is similar to the syntax of customary operations on immutable data structures, but instead of mutating a data structure, creates another one that contains the mutated data. It is exactly what is needed in the MIX emulator because MIX processor instructions mutate the machine state, but the objective to store all states requires emulating this mutation with a sequence of immutable states. Therefore, the mentioned Scala operations are widely used in the authors' implementation of the emulator.

The whole state of the virtual machine is represented by a Scala case class (a class with a few generated methods, typical for working with immutable data structures [12]) consisting of the register state, the memory state, the program counter (the index of the next instruction to execute), the time counter (the time elapsed since the beginning of the program execution), the flag indicating that the machine is halted and the state of the connected devices (discussed in the next subsection). The register state is represented by a case class consisting of the values of all registers and flags. The memory state is represented by a case class consisting of a vector (`scala.collection.immutable.Vector`, an immutable analogue of an array) of the values of all memory cells and two lists of memory locks (discussed in the next subsection).

Implementation of MIX processor instructions generates new states based on the preceding ones according to the specification of these instructions in [1]. In the tracking mode, the virtual machine builds a sequence of states starting from the initial state and finishing at the halted state. This sequence of states is stored in a linked list (`java.util.LinkedList`) and can be traversed in both directions (either stepwise or straight to the next breakpoint) during debugging. In order to save memory, the emulator also supports the non-tracking mode, in which only the current state and no state history are stored.

E. Emulating Input / Output Operations

Input/output operations violate the simple sequence of states described in the previous subsection. The reason is that these operations are performed asynchronously and the main execution flow may continue before an input/output operation has been completed [1]. Therefore, the machine state at a given moment is not fully determined by the program execution flow, but also depends on the speed of input/output operations.

A correct program should consider this factor and synchronise the main execution flow with the input/output operations. Improper synchronisation (for example, an attempt to read from a memory area without verifying that an input operation has finished storing data to this area [1]) may lead to unpredictable results.

Improper synchronisation conditions are hard to determine within an emulator, especially when a program is executed step by step. Due to the high performance of modern computers, any input/output operation will likely have been completed before a user proceeds to the next step. Therefore, synchronisation errors may be unnoticeable during debugging.

For the described reasons, the authors' implementation of a MIX emulator tries to ensure that a program gives the same result independently of the speed of input/output operations. In order to achieve it, memory locks, inspired by SQL data locks [19], are used. When a program initiates an input operation, which reads data from a device and writes them to memory, any attempt to write to the same memory area or read from it before the operation has been completed, causes unpredictable results. Prohibition on such operations is an analogue of the SQL exclusive lock [19]. Similarly, when a program initiates output of memory data to a peripheral device, any attempt to write to the same memory area before output completion leads to an unpredictable state. However, reading from the same memory area does not cause any problems itself. Therefore, the applied lock is an analogue of the SQL shared lock [19].

The emulator applies corresponding exclusive or shared locks when an input/output operation is initiated and stores them in the memory state. If an action prohibited by a lock is attempted before the operation is guaranteed to be completed, the emulator raises an exception telling that the machine state is unpredictable. After a program has been waiting for the operation completion, the emulator changes the memory state (for an input operation) and releases the lock.

However, detection of the situation when a program performs synchronisation with input/output operations is problematic itself. Unlike multi-threading environments in modern programming languages, MIX does not have a special synchronisation command. The emulator recognises the 'busy wait' synchronisation pattern: a conditional jump to the current address when the addressed device is busy (in MIXAL notation, `JBUS *(`*`DEVICE_NUMBER`*`)`) means that the program waits until all input/output operations on this device have been completed. Currently, it is the only recognised synchronisation mechanism, which may be too restrictive. Support for other patterns may be added in future.

Input/output devices themselves are typically emulated with files, as in other emulators. However, the emulator does not depend on this assumption and allows using other objects implementing one of the defined input/output device interfaces. In order to support traversing the sequence of state

back and forth, device states should also be stored. For the file-emulated devices, state storage is implemented by means of creation of a new file copy on each output operation.

IV. Case Study of Scala Features

The approach described in the previous section allows implementing an emulator that satisfies the requirements stated in Sections 1 and 2. This section is devoted to the discussion of the Scala programming language facilities used in this implementation, how they are helpful, and where they are insufficient.

Such novel features of Scala as cross-compilation to different platforms (thanks to the high level of abstraction in Scala), family polymorphism and rich support for immutable data structures significantly facilitate implementation of the described requirements. Cross-compilation allows supporting different platforms and using interfaces with the same core system. Family polymorphism provides an opportunity to implement different modes of MIX functioning remaining the most part of the code unchanged. Immutable data structures and easy operations on them simplify storing and traversing the sequence of states of the virtual machine without excessive memory requirements. These features are not as well developed (or not supported at all) in many other mainstream programming languages as in Scala, so their support is a significant benefit of Scala.

However, there are also some drawbacks in the current support for these features in Scala. First, the implementation of the family polymorphism pattern is rather heavyweight. The concrete data types forming a family have to be defined within one object [17]. It means that the whole concrete model (in terms of Section III. B) should be placed in one file. For example, in the case of the binary model it led to creation of a file of over 500 lines containing related, but quite different classes. It violates the principle of single responsibility [20] and hinders code reading. An opportunity to split the concrete model into different files could give a cleaner code.

Second, a similar requirement exists for the abstract model, too. In this case, the requirement is less restrictive because the abstract model *may* be split into several classes (and therefore, several files) provided that these classes share the same self type [17]. Scala's opportunity to explicitly define the self type of a class, which may differ from the class itself, allows separating the logic of the abstract model to some extent. However, this logic is still tightly coupled within a self type. An opportunity to separate part of this logic into a generic type, whose parameter is the used model (binary or decimal), could make the code cleaner and more understandable. Currently, such an approach is not supported in Scala.

Finally, the equivalence between mutable and immutable data structures does not seem complete. As mentioned in Section III.D, many Scala's operations on immutable data structures are functional equivalents of conventional operations on mutable ones. However, not all customary mutating operations have such equivalents. For example, method `copy` of case classes has no equivalent of the shortcut assignment. If case class `A` has a member `num` of type `Int`, the call `a.copy(num = a.num + 1)` is legal, and its mutable equivalent is `a.num = a.num + 1`. However, in the mutable case, the shortcut assignment `a.num += 1` can be used instead, which has no counterpart for immutable data structures. Similarly, there is no direct equivalent of mutating a class member by means of calling a method of this member. The full set of such operations would allow working with an immutable data structure in completely the same way as with mutable ones and, in some cases, could make the code shorter and simpler.

V. Conclusion and Future Research

The direct result of the research is an emulator of the MIX computer. The emulator can be used in studying and teaching algorithms and their performance. Since the emulator supports both binary and decimal mode of MIX, and checks correctness of input/output operations synchronisation, it can be used in training of avoidance of typical related errors.

The next step of the research is developing an emulator of MMIX, the newer version of MIX. It will provide the same opportunities for studying "The Art of Computer Programming" with a modern hardware architecture as the described MIX emulator does for an older one. Development of a MMIX emulator may help in adaptation of the algorithms described in Knuth's treatise for MMIX.

Another possible future direction of the research is generalisation of the implemented approach to more complex cases of synchronisation. Due to multi-threading, modern run-time environments have much more complex requirements to asynchronous operation correctness (e.g., the definition of a well-formed execution in Java [21]). Therefore, the approach implemented in this study cannot be applied directly in these cases. However, further research of this approach can be useful because multi-threading errors are often difficult to catch and debug [20].

Another result of the study is the analysis of Scala facilities used to implement the emulator. Such features as cross-compilation, family polymorphism, and rich support of immutable data structures are extremely helpful in implementing the described approach.

At the same time, in some cases, Scala rules make the code using these features rather heavy. Some possible improvements could facilitate using these features: an opportunity to define the set of concrete classes in family polymorphism in several separate files, an opportunity to use this concrete family as a parameter of a generic type, and creation of equivalents for immutable data structures for all conventional operations mutating mutable data structures. A possible future research direction is studying how these features can be implemented and combined with other language facilities.

References

[1] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, 1997.

[2] D. E. Knuth, *The Art of Computer Programming, Volume 1, Fascicle 1: MMIX – A RISC Computer for the New Millennium*. Addison-Wesley, 2005.

[3] "GNU MIX Development Kit (MDK)," [Online]. Available: https://www.gnu.org/software/mdk/mdk.html
[4] "JMixSim," [Online]. Available: https://sourceforge.net/projects/jmixsim
[5] "MixIDE," [Online]. Available: http://mixide.sourceforge.net
[6] "MIX Builder," [Online]. Available: http://www.menees.com/MIXBuilder.htm
[7] "The Expandable MIX Emulator (EMIX)," [Online]. Available: http://dandrade.tripod.com
[8] "MIX Assembler and Simulator," [Online]. Available: http://puszcza.gnu.org.ua/software/mix/
[9] "MIX," [Online]. Available: http://web.archive.org/web/20080805212106/http://swiss.csail.mit.edu:80/~adler/MIX
[10] "Dan's MIX Simulator and MIXAL Compiler," [Online]. Available: http://www.recreationalmath.com/mixal
[11] "MixEmul," [Online]. Available: https://rbergen.home.xs4all.nl/mixemul.html
[12] M. Odersky, P. Altherr, V. Cremet, G. Dubochet, B. Emir et al., "Scala Language Specification: Version 2.12," [Online]. Available: http://www.scala-lang.org/files/archive/spec/2.12
[13] JetBrains, "Kotlin Language Documentation," [Online]. Available: http://www.kotlinlang.org/docs/kotlin-docs.pdf
[14] P. J. Guo, "Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education," *Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE'13)*, pp. 579–584, 2013. https://doi.org/10.1145/2445196.2445368
[15] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java virtual machine specification: Java SE 8 edition*, 2015 [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf
[16] Microsoft, "Common Language Runtime (CLR)," [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/clr
[17] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Gubochet et al., "An Overview of the Scala Programming Language," Technical Report LAMP-REPORT-2006-001, École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland, 20 p., 2006.
[18] M. Odersky and L. Spoon, "Scala Collections," [Online]. Available: http://docs.scala-lang.org/overviews/collections/introduction.html
[19] ISO/IEC 9075-2:2016, "Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation)," 2016.
[20] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 431 p., 2009.
[21] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java language specification: Java® SE 8 edition*, 2015 [Online]. Available: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf

Ruslan Batdalov received the Specialist degree in Applied Mathematics and Informatics from Kazan State University, Russia, in 2003 and the Master degree in Computer Systems from Riga Technical University, Latvia, in 2017.
He is a first-year Doctoral student at the Department of Applied Computer Science, Riga Technical University, and a developer at Rietumu Banka, Riga, Latvia. Previously, he worked as a Business Analyst, System Analyst, Activity-Based Costing Specialist. His current research interests include programming languages, their capabilities, structure and design.
He has been an ACM member since March 2010.
E-mail: Ruslan.Batdalov@rtu.lv

Oksana Ņikiforova received the Doctoral degree in Information Technologies (system analysis, modelling and design) from Riga Technical University, Latvia, in 2001.
She is a Professor at the Department of Applied Computer Science, Riga Technical University. Her current research interests include object-oriented system analysis, design and modelling, especially the issues in model driven software development.
E-mail: oksana.nikiforova@rtu.lv