A denotational semantics of a concatenative/ compositional programming language

Jurij Mihelič, William Steingartner, Valerie Novitzká

University of Ljubljana, Faculty of Computer Science and Informatics,
Večna pot 113, 1000 Ljubljana, Slovenia;
jurij.mihelic@fri.uni-lj.si

Technical University of Košice, Faculty of Electrical Engineering and Informatics,
Letná 9, 042 00 Košice, Slovakia;
{william.steingartner, valerie.novitzka}@tuke.sk

Abstract: A distinctive feature of concatenative languages is that a concatenation of their programs corresponds to a composition of meaning functions of these programs. At first programming in such languages may resemble assembly language programming. In spite of this, they also exhibit many similarities to high-level functional programming languages. We start our presentation with the definition of the language syntax. The main part of the paper consists of the definition of a meaning of programs in the language. To do this we employ a well-known method based on denotational semantics. We also informally introduce the language and its meaning as well as present its background and provide motivation for the work. Our exposition is accompanied by many examples and in the last part of the paper, we also discuss various language extensions and identify several proposals for further research.

Keywords: concatenative, compositional, denotational semantics, function, programming language, syntax, stack

1 Introduction

In this paper, we focus on defining the denotational semantics for a new concatenative/compositional language. We begin the paper with a review of the area and related work. First, in the next subsection, we review approaches to define a meaning of programs. Since the presented programming paradigm is not well known, we present its background with a related work. Finally, we will give an informal introduction to the discussed programming language.

1.1 Short overview of the area

Each programming language should have its own formal definition. This definition consists of formal syntax and formal semantics. The former can be concrete

or abstract where a concrete syntax serves for syntax analysis and abstract syntax is suitable for defining the semantics of a program. The latter expresses the meaning of a program. For programs written in purely functional languages, a value of a term is a meaning of the program [8]. In contrary, the semantics of an imperative program is defined as a change of memory states (storage).

Our goal is to define a semantics for our language KKJ[1]. Since there are several well-known approaches to semantics that are reciprocally equivalent and they are used for different purposes, we briefly look over them. One of the most popular methods for defining the semantics of programming languages is structural operational semantics (also known as small-step semantics). This method models, in details, computations explicitly in particular steps of execution and describes the effects of program constructs on program states and each step is expressed as the transition relation [17]. Operational semantics specifies programming languages in terms of program execution on abstract machines which provide an intermediate language stage for compilation. They bridge the gap between the high level of a programming language and the low level of a real machine [4]. Structural operational semantics represents computation by means of deductive systems that turn the abstract machine into a system of logical inferences [19]. An alternative approach in operational semantics is known as natural semantics or big-step semantics. In natural semantics [10], the relationship between the initial and the final state of an execution is constructed. This method is mostly used for imperative languages but it has nice application also in area of domain specific languages, e.g. [2].

A further particular type of small-step semantics is Reduction Semantics with Evaluation Contexts (RSEC) [5], also known as contextual semantics. This method models an execution as a sequence of atomic rewrites of state, between each of which some small amount of time passes [3, 6].

Another approach to semantic methods is axiomatic semantics which models the relationship between pre- and post-conditions on program variables – it describes properties of program state, using the first-order logic [11, 14].

Action semantics [13, 24] is considered as a hybrid of denotational and operational semantics. Action semantics uses English phrases for defining the meaning of syntactic constructs (still being formal). It serves mostly as a very illustrative framework for teaching semantics [24].

In this approach, we present how to formulate and define the denotational semantics. However, denotational semantics is one of the oldest semantic methods [14], where only the contribution of each construct to the computational meaning of the enclosing program is modeled. This method defines the meaning of a program using functions/mappings defined over sets and/or lattices, respectively [18]. The intermediate states during the execution of the constructs are generally of no relevance and are thus not represented. One of its main aims is to provide a proper mathematical foundation for reasoning about programs and for understanding the fundamental concepts of programming languages. Therefore, denotational semantics plays an important *rôle* in the language design process.

---

[1] KKJ – konkatenacijski/kompozicijski jezik in Slovene or konkatenatívny/kompozičný jazyk in Slovak, both meaning concatenative/compositional language.

1.2 Background

In this paper, we focus on a simple programming language introducing only a handful of programming constructs while still offering a flexible and useful programming environment. The simplicity of the language, which for the purposes of this presentation we call KKJ, spurs a plethora of possibilities such as an option to clearly define semantics, to straightforwardly implement an interpreter or a compiler, and to design program analysis tools for the purpose of optimization or verification, etc. The so called concatenative/compositional nature of the language offers a great flexibility for decomposing a program into units than can be executed in parallel, and, as such, have a potential to be suitable for modern multi-core computer architectures as well as to serve as an intermediate code representation in compilers and interpreters. We provide a brief discussion on this later in the paper.

The proposed language abstracts away all the intricate details of any possible underlying hardware and processor architecture. Nevertheless, implementing programs in KKJ may occasionally resemble programming in an assembly language (without architectural details) since a programmer uses only simple basic "instructions" to arrive at a solution for a particular programming task.

On the other hand, the language also offers a programming construct representing first-class anonymous function which is usually not present in low-level languages and as such a programmer's perspective is raised to a level usually occurring in higher-level programming languages. In particular, programming in KKJ becomes similar to programming in functional programming languages without using variables, e.g., the *tacit* or *point-free* style of programming which is oftentimes aspired.

Many ideas found in this paper already appear in some similar form throughout the scientific literature. Already in 1977, John Backus in his Turing award lecture argued for simplification of conventional programming languages as well as proposed functional programming systems as an alternative. He explicated a *function-level* programming where new programs are written by putting together existing programs rather than by manipulating values and then abstracting from those values to produce programs. The result of his efforts is the FP language proposed in the lecture, which through many improvements involving several researchers evolved later into the FL [1], and other programming languages.

In contrast, conventional well-known *functional* programming languages such as Haskell and Lisp mostly base on the *lambda calculus* which puts forth *value-level* programming where new values are constructed from existing ones until the final result is obtained. The development of these has already greatly advanced from their inception and many modern functional programming languages offer both high expressiveness as well as execution speed.

Nevertheless, even though both function-level and value-level functional programming adopt somewhat different views on object manipulation their fundamental programming concept is still a function. In several other successful programming paradigms, such a concept may also be an object (i.e., object-oriented programming) or a relation (i.e., logic programming).

Our other source of inspiration is the area of *concatenative* programming languages. The term arises from the property that a (syntactic) concatenation of programs corresponds to the (semantic) composition of functions. Indeed, both aspects actually

represent *monoid* algebraic structure. In particular, an empty program is a unit for program concatenation and program concatenation is clearly associative while an identity function is a unit for function composition and composition is associative as well.

The paradigm of concatenative languages is valuable for fundamental software engineering research (ideally for language experimentation and worth to be applied in software engineering because of their unique features) and might prove to be a suitable foundation for future programming [7, 21, 22] .

Unfortunately, the area received little attention from a scientific community and thus its treatment lack theoretical rigor and strictness. Moreover, many of the concatenative languages exist only as a prototype implementation, are not actively developed, and they lack financial support as well as any serious development environment and support. A list of several examples including brief descriptions is available on the http://concatenative.org website. Nevertheless, there are some prominent examples worth mentioning.

The Forth language [12] appeared in 1970 and is considered as a flexible, extensible, stack-based, procedural, concatenative programming language with many applications. Another is the Joy language [23] from 2001 which is considered as a purely functional, of theoretical interest and has established the term concatenative and had influenced many other concatenative languages. Finally, the Factor language [15, 25] which was conceived in 2003 and is an actively developed, dynamically typed, garbage collected language with a self-hosting compiler, interactive development environment, and large standard library. It supports both functional and object-oriented programming paradigm and is well used in practice.

Finally, we mention also several stack-based application virtual machines which often display many similarities with concatenative paradigm and they have already proved themselves to be successful and efficient in practice. Probably the most well-know examples are the Java Virtual Machine and Common Language Runtime as well as Erlang's BEAM runtime.

The language proposed in this paper resembles many of the above-mentioned languages in the way function composition is used, but it is simpler in order to enable theoretical consideration using formal methods. Our simplifications are partially in syntax (e.g., less syntactic domains) and also in semantics (e.g., no modules and information hiding), but the main concatenative features are present (e.g., function combinators).

The goal of the paper is two fold: first, to serve as a presentation of concatenative (compositional) programming constructs, and, second, to establish a firm basis for understanding the meaning of concatenative programs which is currently missing in the scientific literature. Indeed, current implementations of such programming languages are based on *ad hoc* definition of the concepts.

1.3 Informal description of KKJ

Since the corresponding concatenative programming paradigm of the proposed language is not well-known we start with a brief informal description of KKJ followed by a demonstration of the evaluation of an example program.

Syntactically a program in KKJ is just a sequence of *words*, i.e., numerals represent-

ing numbers and names representing functions, where words are separated with a whitespace. Additionally, there is also a programming construct called a *quotation* (i.e., abstraction) which encapsulates another program with brackets and it represents a definition of an anonymous function. The term quotation is also used in the Lisp programming language for a similar construct.
Observe the following two examples of programs in KKJ. The first one is without quotations and it consists of ten words:

```
3 4 add dup ispos 5 6 swap choose mul,
```
(1)

and the second one contains two quotations:

```
14 {dup dup} {add add} compose apply.
```
(2)

As we will see later, both programs evaluate to the same value, i.e., they are semantically equivalent. We notice also, that the defined syntax exhibits such a great simplicity that it is consequently very straightforward to implement an efficient parser to perform syntax analysis.
Now we focus our attention to the meaning of the above two programs. In what follows we show two different approaches for program evaluation. The first approach is based on term-rewriting (i.e., reduction semantics) where specific patterns in the program are found and being replaced until a *normal form* of the program is obtained. An example evaluation of the program 1 is shown in Table 1. We omit rule specification and rely on a reader's intuitive understanding.

Table 1
Evaluation of a program based on term-rewriting.

program	substitutions
`3 4 add dup ispos 5 6 swap choose mul`	`3 4 add` $\rightarrow$ `7`, `5 6 swap` $\rightarrow$ `6 5`
`7 dup ispos 6 5 choose mul`	`7 dup` $\rightarrow$ `7 7`
`7 7 ispos 6 5 choose mul`	`7 ispos` $\rightarrow$ `true`
`7 true 6 5 choose mul`	`true 6 5 choose` $\rightarrow$ `6`
`7 6 mul`	`7 6 mul` $\rightarrow$ `42`
`42`	

Based on the example one can clearly see that the language is functional in a way that basic expressions represent built-in (or primitive) functions and a sequence of expressions forms a new expression representing a composition of functions; in other words, a computation is carried out entirely through the evaluation of expressions. Even though a functional point of view is a more appropriate one, built-in functions may also represent constructs usually found in the imperative paradigm of programming.
Another approach to evaluation is based on the stack data structure (i.e., state-transition semantics), where a program successively transforms the stack. Each word represents a function operating on the stack, where numerals represent functions pushing a number onto the stack. Again, an example evaluation of the program (1) is shown in Table 2.

Table 2
Evaluation of a program based on a stack.

program	stack
`3 4 add dup ispos 5 6 swap choose mul`	
`4 add dup ispos 5 6 swap choose mul`	3
`add dup ispos 5 6 swap choose mul`	3 4
`dup ispos 5 6 swap choose mul`	7
`ispos 5 6 swap choose mul`	7 7
`5 6 swap choose mul`	7 **true**
`6 swap choose mul`	7 **true** 5
`swap choose mul`	7 **true** 5 6
`choose mul`	7 **true** 6 5
`mul`	7 6
	42

Indeed, many concatenative programming languages are stack-based (i.e., operations manipulate the implicit stack), which may suggest an imperative view. However, in imperative languages the state is implicit, but it is explicitly manipulated (e.g. via assignments) whereas in stack-based concatenative languages the stack manipulation is considered implicit. Moreover, operations in these languages may also be regarded as unary transformations from stack to another stack.
Consider now the program (2). What value does it evaluate to? What is the meaning of a quotations {`dup dup`} and {`add add`}? We intuitively know that the first one is a function that duplicates the top element twice and the second one is a function that pops and adds three top values on the stack. Now, composing these two functions, i.e., the meaning of {`dup dup`} {`add add`} `compose`, results in a new function which given a number returns its triple. So, we simply calculate 3×14 here. Nevertheless, to clearly answer these questions we have to formally define a meaning function and that is the goal of the rest of the paper.

2 Syntax and semantics

In this section, we present a foundations of KKJ– we start with the definition of syntax. Then we define an abstraction of computer memory as memory states. After this step, we are ready to define a semantics of KKJ.

2.1 Syntax

Before delving into the semantics of KKJ, we define abstract syntax if its expressions of KKJ. First, we introduce syntactic domains:

- $i \in$ **IntNum** – integer numerals, i.e., strings of digits,
- $n \in$ **Name** – names, i.e., strings of alphanumeric characters,
- $E \in$ **Expr** – expressions.

Here, the elements $i \in$ **IntNum** represent integer numbers and they have no internal structure from the semantic point of view, but syntactically they can be represented with a regular expression $[0,\ldots,9]^+$. Similarly $n \in$ **Name** represent function names without any internal structure significant to defining semantics and its internal syntax is described with a regular expression $[a,\ldots,z][a,\ldots,z,0,\ldots,9]^*$.

To describe the syntax of expressions $E \in$ **Expr** in the programming language KKJ, we use the well-known BNF-notation:

$$E ::= \varepsilon \mid i \mid n \mid \{E\} \mid E\,E. \tag{3}$$

Here, ε stands for an empty expression, a numeral $i \in$ **IntNum** and a name $n \in$ **Name** are considered as expressions as well. Additionally, one can form a new expression by quoting (with brackets) an existing expression, i.e., a quotation $\{E\}$, to represent an anonymous function defined by the expression E. And, finally, a new expression can be formed by concatenating (in juxtaposition using only whitespace as a delimiter) two expressions, i.e., $E\ \ E$, simply to represent their composition.

We also note here that a name $n \in$ **Name** is considered to represent a built-in (also called primitive) operation such as `add`, `sub`, `pop`, `dup`, `compose`, and `apply`. We exactly specify these names in the following sections while specifying semantics. We assume that undefined names are not allowed in correct programs, whereas in practice they would cause the program to crash or raise an exception. However, we extend later the basic syntax of the language with a construct which allows us to assign functions to names.

2.2 Representation of states

The state is, in general, an abstraction of a computer memory (a kind of memory snapshot) and in our case, it is actually represented by a stack. In what follows we show how the stack is defined. First, we introduce a new semantic domain **Int** for integer values and **Bool** for Boolean values. They are defined as

$$\textbf{Int} = \mathbb{Z} \quad \text{and} \quad \textbf{Bool} = \mathbb{B} = \{\textbf{false}, \textbf{true}\}.$$

Second, we introduce a semantic domain **Stack** for representing the stack as well as a semantic domain **Fun** (a function space) for representing functions manipulating the stack, i.e.,

$$\textbf{Fun} = \textbf{Stack} \to \textbf{Stack}.$$

Now to define **Stack**, we first introduce a new domain **Value**, which represents values (i.e., elements, members) residing on the stack, i.e.,

$$\textbf{Value} = \textbf{Int} \cup \textbf{Bool} \cup \textbf{Fun}.$$

Finally, the type stack is represented with the following semantic domain

$$\textbf{Stack} = \textbf{Value}^* \cup \{\bot\},$$

where X^* represents Kleene's closure (or iteration) over X. Observe that, a stack $s \in \textbf{Stack}$ represents an abstraction (a snapshot) of the actual memory and thus represents a state in our semantics. A particular content of the stack may also be represented with an ordered sequence, i.e., $(x_1, x_2, \ldots, x_n) \in \textbf{Stack}$, where $x_i \in \textbf{Value}$ for each $1 \leq i \leq n$ and x_n is a topmost element of the stack.

Notational remark: We mostly use symbols $i, j \in \textbf{Int}$ for integers and $b, d \in \textbf{Bool}$ for Boolean values, $f, g, h \in \textbf{Fun}$ for functions, $x, y, z \in \textbf{Value}$ for value of any type, and $s, t \in \textbf{Stack}$ for stacks.

2.3 Semantics

Now let us describe denotational semantics for the language KKJ. To do this we specify a semantics of expressions $E \in \textbf{Expr}$, where their meaning can be summarized by a function from **Stack** to **Stack**, i.e.,

$$\mathscr{S} : \textbf{Expr} \to (\textbf{Stack} \to \textbf{Stack}). \tag{4}$$

The function $\mathscr{S}$ will be defined inductively in the following sections. In this paper, we often omit the symbol $\mathscr{S}$ and use the semantic bracket $[\![E]\!]$ around the syntactic parameter $E \in \textbf{Expr}$, when the notation is clear, e.g., $[\![E]\!] \equiv \mathscr{S}[\![E]\!]$.

When providing inductive definitions, we define semantic clauses for various syntactic constructs. Doing this we use several auxiliary functions defined as follows:

- newstack: $\to$ **Stack**,
- id: **Stack** $\to$ **Stack**,
- push: **Value** $\to$ **Stack** $\to$ **Stack**.

Here, the newstack is an initial function which *ex nihilo* creates a new empty stack, i.e., $\mathsf{newstack} = ()$, the id function (identity) leaves a stack unchanged, i.e., $\mathsf{id}\ s = s$, while the push function appends an element to a stack, i.e.,

$$\begin{aligned} \mathsf{push}\ x\ (x_1, x_2, \ldots, x_n) &= (x_1, x_2, \ldots, x_n, x), \\ \mathsf{push}\ x\ \bot &= \bot, \end{aligned}$$

where $x, x_i \in \textbf{Value}$ and $1 \leq i \leq n, n \geq 0$.

In the rest of the paper, we often write $s = (x_1, x_2, \ldots, x_n)$ and use the symbol $\cdot$ as an infix operator representing push, i.e., $s \cdot x \equiv \mathsf{push}\ x\ s$. Moreover, we also use $\cdot$ in pattern matching, and thus define $\mathsf{pop}\ s \cdot x = s$ as a function that returns the stack without its top element, and $\mathsf{top}\ s \cdot x = x$ as the function that returns the top element of the stack. Notice also, that we define push as a curried function.

Generally, we define function $[\![E]\!]$ by defining it on each expression from eq. (3) as follows:

$$\begin{aligned} [\![\varepsilon]\!]\ s &= s, \\ [\![i]\!]\ s &= s \cdot \mathbf{i} && \forall i \in \textbf{IntNum} \\ [\![\{E\}]\!]\ s &= s \cdot [\![E]\!], \\ [\![E_1 E_2]\!]\ s &= ([\![E_2]\!] \circ [\![E_1]\!])\ s. \end{aligned}$$

The semantics of an expression $[\![n]\!]\ s$ depends on concrete name $n \in \textbf{Name}$: the language KKJ uses concrete names for arithmetic operations, Boolean operations, operations for stack manipulation, functions, condition and loop expressions. We explain the details of these definition in sections that follow.

2.4 Expression concatenation

Let us begin with a presentation of the semantics for expression concatenation. The semantics of an empty program is the identity function and the concatenation of two programs corresponds to a composition of the semantic functions corresponding to the programs. The semantic clauses are given in Table 3.

Table 3
Semantics of the empty expression and expression concatenation

$$[\![\varepsilon]\!] = \mathsf{id} \qquad [\![E_1\ E_2]\!] = [\![E_2]\!] \circ [\![E_1]\!]$$

where

$$f \circ g\ s = \begin{cases} \bot, & \text{if } s = \bot \vee g\ s = \bot; \\ f\ (g\ s), & \text{otherwise.} \end{cases}$$

The latter rule also gives a rationale for the term *concatenative* for naming this sort of programming languages, since concatenation of valid programs results in a new valid program. Moreover, the new program semantics is defined as a composition of the semantics of the original programs. Hence, the term *compositional* languages may also be used [9] analogously to the term *applicative* which is sometimes used for conventional functional programming languages.

Now, consider a sequence of expressions, we state that the exact order of how the expression concatenation rule is applied is not important from the viewpoint of semantics. First we write the following theorem.

Theorem 1. Let $E_1, E_2, E_3 \in \textbf{Expr}$. We have

$$[\![E_3]\!] \circ [\![E_1\ \ E_2]\!] = [\![E_2\ \ E_3]\!] \circ [\![E_1]\!].$$

Proof. First, observe that the function composition as defined in Table 3 is associative. Then, on the left-hand side of the equation we have

$$[\![E_3]\!] \circ [\![E_1\ \ E_2]\!] = [\![E_3]\!] \circ ([\![E_2]\!] \circ [\![E_1]\!]) = [\![E_3]\!] \circ [\![E_2]\!] \circ [\![E_1]\!],$$

and on the right-hand side we have

$$[\![E_2\ \ E_3]\!] \circ [\![E_1]\!] = ([\![E_3]\!] \circ [\![E_2]\!]) \circ [\![E_1]\!] = [\![E_3]\!] \circ [\![E_2]\!] \circ [\![E_1]\!]$$

which are both obviously equal. □

When there are three or more concatenated expressions the rule can be applied in multiple ways. For example, having three expressions we may decompose $E_1\ E_2\ E_3$ into either $[\![E_1\ \ E_2\ \ E_3]\!] = [\![E_3]\!] \circ [\![E_1\ \ E_2]\!]$ or $[\![E_1\ \ E_2\ \ E_3]\!] = [\![E_2\ \ E_3]\!] \circ [\![E_1]\!]$, yet still obtaining the same semantical result. In a more general case with n concatenated expressions, we $n-2$ times use Theorem 1. We can state the following corollary.

Corollary. *Let $E_1, E_2, \ldots, E_n \in$ **Expr** be n expressions. We have*

$$[\![E_1\ \ E_2\ \ \ldots\ \ E_n]\!] = [\![E_n]\!] \circ \cdots \circ [\![E_2]\!] \circ [\![E_1]\!].$$

2.5 Arithmetic and Boolean operations

In this section, we consider several functions representing arithmetic and Boolean operations. In particular, these functions deal only with integer or Boolean values, which may during the operation be consumed from the stack or produced and pushed onto it.

See semantic clauses listed in Table 4 for semantic definitions of the selected basic arithmetic and Boolean operations. In the specification, the meaning of $[\![i]\!]\ s$ is to push the number $\mathbf{i} \in \mathbf{Int}$ corresponding to the numeral i on the stack s. For details on how to define an additional semantic function determining the number for a given numeral see [14].

Table 4
Semantics of arithmetic and Boolean operations

$$[\![i]\!]\ s = s{\cdot}\mathbf{i} \qquad [\![\texttt{sub}]\!]\ s{\cdot}i{\cdot}j = s{\cdot}(i-j)$$

$$[\![\texttt{add}]\!]\ s{\cdot}i{\cdot}j = s{\cdot}(i+j) \qquad [\![\texttt{mul}]\!]\ s{\cdot}i{\cdot}j = s{\cdot}(i \times j)$$

$$[\![\texttt{false}]\!]\ s = s{\cdot}\mathbf{false} \qquad [\![\texttt{not}]\!]\ s{\cdot}b = s{\cdot}\begin{cases}\mathbf{true}, & \text{if } b = \mathbf{false};\\ \mathbf{false}, & \text{otherwise.}\end{cases}$$

$$[\![\texttt{true}]\!]\ s = s{\cdot}\mathbf{true} \qquad [\![\texttt{and}]\!]\ s{\cdot}b{\cdot}d = s{\cdot}\begin{cases}\mathbf{true}, & \text{if } b = d = \mathbf{true};\\ \mathbf{false}, & \text{otherwise.}\end{cases}$$

$$[\![\texttt{cmp}]\!]\ s{\cdot}i{\cdot}j = s{\cdot}\mathrm{sgn}(i-j) \qquad [\![\texttt{isneg}]\!]\ s{\cdot}i = s{\cdot}\begin{cases}\mathbf{true}, & \text{if } i < 0;\\ \mathbf{false}, & \text{otherwise.}\end{cases}$$

$$[\![\texttt{ispos}]\!]\ s{\cdot}i = s{\cdot}\begin{cases}\mathbf{true}, & \text{if } i > 0;\\ \mathbf{false}, & \text{otherwise.}\end{cases}$$

The table also includes basic arithmetic operations such as addition (`add`), subtraction (`sub`), and multiplication (`mul`) as well as operations to produce Boolean values (`true` and `false`) together with operations for logical negation (`not`) and conjunction (`and`).

Additionally, we also include the operation for comparing (`cmp`) two integer numbers producing -1,0, or 1 on the stack if the first number is lower, the numbers are equal, or the second number is lower, respectively. And finally, operations to determine whether the top number on the stack is negative (`isneg`) or is positive (`ispos`).

Observe also, that the listed operations are defined only when the "input types match" as indicated by the use of variable names in the stack notation. For example, the operation $[\![\texttt{add}]\!]\ s{\cdot}i{\cdot}j$ is only defined when the top two elements i and j of the stack belong to the **Int** domain while it is not defined in all other cases, e.g., when the top element $j \in$ **Bool**, etc. Hence, the meaning of `7 true add` is evaluated as

$$[\![\texttt{7 true add}]\!]\ s = [\![\texttt{add}]\!]\ s{\cdot}7{\cdot}\textbf{true} = \bot.$$

In this paper, we do not delve into details of type checking issues; we assume that the expressions are always correct regarding types. See also [16] for an in-depth discussion on types.

Several other important arithmetic and integer comparison operations can easily be formed using the basic operations from Table 4. To demonstrate this we give some examples in Table 5. Note: In the examples, we also use the operations `dup` (top of stack duplication) and `swap` (exchange of the top two elements) which are both defined in the next subsection.

Table 5
Several derived arithmetic and logical operations

`pred`	$\equiv$ `1 sub`	...	predecessor
`succ`	$\equiv$ `1 add`	...	successor
`neg`	$\equiv$ `0 swap sub`	...	negation
`iszero`	$\equiv$ `dup isneg not swap ispos not and`	...	is it zero?
`lt`	$\equiv$ `cmp isneg`	...	$<$
`le`	$\equiv$ `cmp dup isneg swap iszero or`	...	$\leq$
`eq`	$\equiv$ `cmp iszero`	...	$=$
`ne`	$\equiv$ `eq not`	...	$\neq$
`ge`	$\equiv$ `lt not`	...	$\geq$
`gt`	$\equiv$ `le not`	...	$>$
`or`	$\equiv$ `not swap not and not`	...	logical disjunction
`square`	$\equiv$ `dup mul`	...	square

2.6 Stack manipulation

In a programming language based on the function application parameters given to a function are explicitly specified by a programmer, but in a language based on function composition, parameters are implicitly set on a data stack and must there also be put into a specific order as required by the corresponding operation. Consequently, the programmer must be able to explicitly manage the stack by using special oper-

ations for manipulating the values on the stack. Such operations are occasionally also called *rewiring* operations.

We present definitions of semantic clauses for several stack manipulation operators in Table 6. Here, `clear` empties the stack, `id` represent the identity function, `pop` removes the top element, `dup` duplicates the top element, `over` duplicates the element just below the top of stack, `swap` exchanges the top two elements, and `rotl` rotates the top three element to the left.

Table 6
Semantics of basic stack manipulation operators

$[\![\texttt{clear}]\!]\ s = \mathsf{newstack}$	$[\![\texttt{over}]\!]\ s{\cdot}x{\cdot}y = s{\cdot}x{\cdot}y{\cdot}x$
$[\![\texttt{id}]\!] = \mathsf{id}$	$[\![\texttt{swap}]\!]\ s{\cdot}x{\cdot}y = s{\cdot}y{\cdot}x$
$[\![\texttt{pop}]\!] = \mathsf{pop}$	$[\![\texttt{rotl}]\!]\ s{\cdot}x{\cdot}y{\cdot}z = s{\cdot}y{\cdot}z{\cdot}x$
$[\![\texttt{dup}]\!]\ s{\cdot}x = s{\cdot}x{\cdot}x$	

Using the operations in Table 6 we can clearly perform the removal of arbitrary number of top stack elements, e.g., `pop2` $\equiv$ `pop pop`, `pop3` $\equiv$ `pop pop pop`, etc. We can also duplicate the top two elements, e.g., `dup2` $\equiv$ `over over`, but we cannot duplicate three or more top elements of the stack.

Let us notice also that `rotl` operation enables us to obtain any permutation of the top three stack elements. See Table 7 for definitions of operations which, given a stack $s{\cdot}x{\cdot}y{\cdot}z$ with top three elements x, y, and z, produce a particular permutation.

Table 7
Several derived stack manipulation operations

`id`		$\ldots\ s{\cdot}x{\cdot}y{\cdot}z$
`swap`		$\ldots\ s{\cdot}x{\cdot}z{\cdot}y$
`swapOver`	$\equiv$ `rotl swap`	$\ldots\ s{\cdot}y{\cdot}x{\cdot}z$
`rotl`		$\ldots\ s{\cdot}y{\cdot}z{\cdot}x$
`rotr`	$\equiv$ `rotl rotl`	$\ldots\ s{\cdot}z{\cdot}x{\cdot}y$
`mirror`	$\equiv$ `rotl rotl swap`	$\ldots\ s{\cdot}z{\cdot}y{\cdot}x$

2.7 Functions

In this subsection, we continue with a semantic clause for a quotation which represents an anonymous function. We also define some useful operations for manipulating functions; such operations are usually called *combinators*. See Table 8 for the list of quintessential semantic clauses appearing in concatenative programming languages.

Here, the semantics of quotation is to push the enclosing function on the stack. Operation `apply` takes an existing function from the stack and applies the function

Table 8
Semantics of quotations and function operations

$[\![\{E\}]\!]\ s = s \cdot [\![E]\!]$	$[\![\texttt{compose}]\!]\ s \cdot f \cdot g = s \cdot (g \circ f)$
$[\![\texttt{apply}]\!]\ s \cdot f = f\ s$	$[\![\texttt{applyOver}]\!]\ s \cdot f \cdot x = (f\ s) \cdot x$
$[\![\texttt{quote}]\!]\ s \cdot x = s \cdot \mathsf{push}\ x$	

on the remaining stack, `quote` takes a value from the stack and produces a function that pushes that value on the stack. Next, we have the `compose` operation which takes two functions and produces their composition, and `applyOver` which acts similarly to `apply` but it preserves the top element of the stack.

As an example, let us now evaluate the program (2) from the introduction.

$$
\begin{aligned}
&[\![\texttt{14 \{dup dup\} \{add add\} compose apply}]\!]\ s = \\
&\quad = [\![\texttt{apply}]\!] \circ [\![\texttt{compose}]\!] \circ [\![\{\texttt{add add}\}]\!] \circ [\![\{\texttt{dup dup}\}]\!] \circ [\![\texttt{14}]\!]\ s \\
&\quad = [\![\texttt{apply}]\!] \circ [\![\texttt{compose}]\!] \circ \mathsf{push}\ [\![\texttt{add add}]\!] \circ \mathsf{push}\ [\![\texttt{dup dup}]\!] \circ [\![\texttt{14}]\!]\ s \\
&\quad = [\![\texttt{apply}]\!] \circ [\![\texttt{compose}]\!]\ s \cdot 14 \cdot [\![\texttt{dup dup}]\!] \cdot [\![\texttt{add add}]\!] \\
&\quad = [\![\texttt{apply}]\!]\ s \cdot 14 \cdot ([\![\texttt{add add}]\!] \circ [\![\texttt{dup dup}]\!]) \\
&\quad = [\![\texttt{add add}]\!] \circ [\![\texttt{dup dup}]\!]\ s \cdot 14 = [\![\texttt{add}]\!] \circ [\![\texttt{add}]\!] \circ [\![\texttt{dup}]\!] \circ [\![\texttt{dup}]\!]\ s \cdot 14 \\
&\quad = [\![\texttt{add}]\!] \circ [\![\texttt{add}]\!]\ s \cdot 14 \cdot 14 \cdot 14 = s \cdot 42
\end{aligned}
$$

Now consider a function $\texttt{twice} \equiv \texttt{dup compose apply}$ which applies a function twice. We have $[\![\texttt{twice}]\!]\ s \cdot f = (f \circ f)\ s$. However, maybe contrary to the intuition, the input and output arity of the function f need not match. For example, let $f = \{\texttt{dup}\}$ which takes zero elements from the stack and produces one, thus $[\![\{\texttt{dup}\}\ \texttt{twice}]\!]\ s \cdot x = [\![\texttt{dup}]\!] \circ [\![\texttt{dup}]\!]\ s \cdot x = s \cdot x \cdot x \cdot x$.

2.8 Conditional expression

Now let us introduce a simple conditional expression. It consumes three elements from the stack: one Boolean value and two more elements. The Boolean value represents a condition, based on which one of the other two elements is pushed back to the stack. We call this operation `choose` and its semantic definition is given in Table 9.

To define `choose`, we also introduced a special utility function cond which has the following signature

$$(\textbf{Stack} \to \textbf{Stack}) \times (\textbf{Stack} \to \textbf{Stack}) \times (\textbf{Stack} \to \textbf{Stack}) \to (\textbf{Stack} \to \textbf{Stack}).$$

The function cond takes three stack manipulating functions and combines them into a new one. Here, the idea of applying $\mathsf{cond}(f, g, h)$ to the given stack s is as follows. First, the function f is applied to s, and the top element of the resulting stack s' is checked: if it is $\textbf{true} \in \textbf{Bool}$ or $\textbf{false} \in \textbf{Bool}$ then g or h is applied on s', respectively.

Table 9
Semantics of the `choose` conditional operator

$$[\![\texttt{choose}]\!]\ s{\cdot}b{\cdot}x{\cdot}y = \mathsf{cond}(\mathsf{push}\ b, \mathsf{push}\ x, \mathsf{push}\ y)\ s$$

where

$$\mathsf{cond}(f,g,h)\ s = \begin{cases} g\ s', & \text{if } f\ s = s'{\cdot}\textbf{true}; \\ h\ s', & \text{if } f\ s = s'{\cdot}\textbf{false}; \\ \bot, & \text{otherwise.} \end{cases}$$

Notice that, the function cond may not be defined when the function f does not leave a Boolean value on the top of the stack; however, we are only interested in cases when it does. Now, let us define a total function and consider cases when cond is total.

Definition 1. *A function f is total if $f\ s = \bot$ if and only if $s = \bot$.*

Lemma 1. *Let f, g, and h be total functions on **Stack**, $f,g,h : \textbf{Stack} \to \textbf{Stack}$. If the application of f always produces a Boolean value from **Bool** on the top of stack, then the function $\mathsf{cond}(f,g,h)$ is also total.*

Proof. Consider the definition of cond from Table 9. Since, f is total we have $s \neq \bot \implies f\ s \neq \bot$, and, by assumption, either $f\ s = s'\ {\cdot}\textbf{true}$ or $f\ s = s'\ {\cdot}\textbf{false}$. In the former, we have $\mathsf{cond}(f,g,h)\ s = g\ (\mathsf{pop}\ (f\ s))'$, and, in the latter, we have $\mathsf{cond}(f,g,h)\ s = h\ (\mathsf{pop}\ (f\ s))'$. □

Since, the function push b (used in the definition of `choose`) always pushes a Boolean on the stack, we have the following corollary.

Corollary. *The semantic clause $[\![\texttt{choose}]\!]\ s{\cdot}b{\cdot}x{\cdot}y$ is a total function.*

Proof. Observe, that all push b, push x, and push y are total functions. Moreover, push b leaves $b \in \textbf{Bool}$ on the top of the stack. Now, use Theorem 2.8. □

We can also observe this if we simplify the definition of the `choose` operation like this:

$$[\![\texttt{choose}]\!]\ s{\cdot}b{\cdot}x{\cdot}y = \mathsf{cond}(\mathsf{push}\ b, \mathsf{push}\ x, \mathsf{push}\ y)\ s = \begin{cases} s{\cdot}x, & \text{if } b = \textbf{true}; \\ s{\cdot}y, & \text{otherwise.} \end{cases}$$

Obviously, the cond function is quite versatile and consequently used as a basis in other conditional and looping programming constructs. For example, let us introduce an operation $\texttt{if} \equiv \texttt{choose apply}$ and its semantics as

$$\begin{aligned}[\![\texttt{if}]\!]\, s{\cdot}b{\cdot}f{\cdot}g &= [\![\texttt{choose apply}]\!]\, s{\cdot}b{\cdot}f{\cdot}g = [\![\texttt{apply}]\!] \circ [\![\texttt{choose}]\!]\, s{\cdot}b{\cdot}f{\cdot}g \\ &= [\![\texttt{apply}]\!] \circ \mathsf{cond}(\mathsf{push}\ b, \mathsf{push}\ f, \mathsf{push}\ g)\ s = \mathsf{cond}(\mathsf{push}\ b, f, g)\ s \\ &= \begin{cases} f\ s, & \text{if } b = \textbf{true}; \\ g\ s, & \text{otherwise.} \end{cases}\end{aligned}$$

We also observe similar corollary.

Corollary. *The semantic clause* $[\![\texttt{if}]\!]\, s{\cdot}b{\cdot}f{\cdot}g$ *is a total function if* f *and* g *are total functions.*

Proof. Let $b \in \textbf{Bool}$ be a Boolean expression. From the semantics of $[\![\texttt{if}]\!]$ we see that either $[\![\texttt{if}]\!]\, s{\cdot}b{\cdot}f{\cdot}g = f\ s$ and $[\![\texttt{if}]\!]\, s{\cdot}b{\cdot}f{\cdot}g = g\ s$, which are both total if f and g are total. □

2.9 Iteration

In imperative languages, one of the more general programming constructs supporting iteration is a while loop. In this section, we consider a similar construct for our language.
Intuitively the `while` operation expects two functions on the stack: a loop condition followed by a loop body. It then executes the condition, which should push a Boolean value on the stack. The top of the stack is then checked and consumed: if it equals **false** the iteration ends, otherwise if it equals **true** the body is executed and the process is repeated.

To define a semantics of the `while` operation we employ similar idea as is presented in [14]. In particular, we first rewrite `while` using `if` operation, i.e.,

$$\{C\}\ \{B\}\ \texttt{while} = C\ \{B\ \{C\}\ \{B\}\ \texttt{while}\}\ \{\}\ \texttt{if}$$

and proceed as follows

$$\begin{aligned}[\![\{C\}\ \{B\}\ \texttt{while}]\!]\, s &= [\![C\ \{B\ \{C\}\ \{B\}\ \texttt{while}\}\ \{\}\ \texttt{if}]\!]\, s \\ &= [\![\texttt{if}]\!] \circ [\![\{\}]\!] \circ [\![\{B\ \{C\}\ \{B\}\ \texttt{while}\}]\!] \circ [\![C]\!]\, s \\ &= [\![\texttt{if}]\!]\ ([\![C]\!]\, s){\cdot}[\![B\ \{C\}\ \{B\}\ \texttt{while}]\!]{\cdot}\mathrm{id} \\ &= [\![\texttt{if}]\!]\ ([\![C]\!]\, s){\cdot}([\![\{C\}\ \{B\}\ \texttt{while}]\!] \circ [\![B]\!]){\cdot}\mathrm{id} \\ &= \mathrm{cond}([\![C]\!], [\![\{C\}\ \{B\}\ \texttt{while}]\!] \circ [\![B]\!], \mathrm{id})\ s\end{aligned}$$

Unfortunately, we cannot use this equation as a denotational clause because it is not a compositional definition, but if we denote $h = [\![\{C\}\ \{B\}\ \texttt{while}]\!]$, we can rewrite it as

$$h\ s = \mathrm{cond}([\![C]\!], h \circ [\![B]\!], \mathrm{id})\ s$$

and see that h is a least fixed point of a functional F defined by

$$F\ h = \mathrm{cond}([\![C]\!], h \circ [\![B]\!], \mathrm{id}).$$

Now note that $[\![\{C\}\ \{B\}\ \texttt{while}]\!]\ s = [\![\texttt{while}]\!]\ s \cdot [\![C]\!] \cdot [\![B]\!]$ and summarize the semantics of `while` in Table 10. The functionality of the auxiliary function FIX is

$$\mathsf{FIX} : ((\mathbf{Stack} \to \mathbf{Stack}) \to (\mathbf{Stack} \to \mathbf{Stack})) \to (\mathbf{Stack} \to \mathbf{Stack}).$$

The FIX F function thus returns the least fixed point of F, i.e., $F(\mathsf{FIX}\ F) = \mathsf{FIX}\ F$ and if $F\ g = g$ then FIX F is smaller than g. We refer the reader to [14] for the details.

Table 10
Semantics of the `while` operation

$$[\![\texttt{while}]\!]\ s \cdot f \cdot g = \mathsf{FIX}\ F$$

where

$$F\ h = \mathsf{cond}(f, h \circ g, \mathsf{id})$$

3 Extensions and discussion

In this subsection, we discuss some features of and possible extensions to the proposed programming language. Our language only includes values of three types, i.e., **Int**, **Bool**, and **Fun**. However, an additional type could straightforwardly be added similarly as we introduced these three by defining a new semantic domain and corresponding primitive operations.

For example, to support a list data structure in KKJ, we can define a new semantic domain **List** = **Value*** with additional primitive operations such as `head`, `tail`, `cons` defined similarly as in many functional programming languages.

Instead of this, we would rather propose another research direction where functions take the role of lists. In particular, the content of the list represented by a function are the elements which are pushed to the stack if the function is applied. Different functions may represent the same list, e.g., {`1 2`} and {`1 dup dup add`} both push 1 and 2 on the stack.
Concatenation of lists thus corresponds to a composition of functions, i.e., the `compose` operation. Furthermore, to prepend an element to a list we define

`cons` ≡ `swap quote swap compose`.

Its semantics is evaluated as

$$\begin{aligned}
[\![\texttt{cons}]\!]\, s{\cdot}x{\cdot}f &= [\![\texttt{swap quote swap compose}]\!]\, s{\cdot}x{\cdot}f \\
&= [\![\texttt{quote swap compose}]\!]\, s{\cdot}f{\cdot}x \\
&= [\![\texttt{swap compose}]\!]\, s{\cdot}f{\cdot}\mathsf{push}\ x \\
&= [\![\texttt{compose}]\!]\, s{\cdot}\mathsf{push}\ x{\cdot}f \\
&= s{\cdot}(f \circ \mathsf{push}\ x)
\end{aligned}$$

Clearly, the resulting list (function on the stack) first pushes the prepended element x and afterwards also the elements represented by f.
Another useful programming operation is to take one or more elements from the stack and store them in a list for later manipulation. We refer to such operations as *stack packing* and we already introduced one such operation, i.e., `quote`.
Using functions as lists we can easily build operation which packs an arbitrary number of the top stack elements. The idea lies in the repeated use of `cons`. For example: `quote2` $\equiv$ `quote cons` and `quote3` $\equiv$ `quote cons cons` pack two and three elements from the stack into a list, respectively. Again let us evaluate the semantics of `quote2`

$$\begin{aligned}
[\![\texttt{quote2}]\!]\, s{\cdot}x{\cdot}y &= [\![\texttt{quote cons}]\!]\, s{\cdot}x{\cdot}y \\
&= [\![\texttt{cons}]\!]\, s{\cdot}x{\cdot}\mathsf{push}\ x \\
&= s{\cdot}(\mathsf{push}\ y \circ \mathsf{push}\ x)
\end{aligned}$$

Besides that, we can also access an arbitrary element of the stack and push it on the top. We refer to this as *stack picking* and we already have operations to pick the top and the element below the top, i.e., `dup` and `over`, respectively. To construct the operation which picks the element two positions below the stack top, we simply use `pick2` $\equiv$ `quote2 over applyOver`, and similarly `pick3` $\equiv$ `quote3 over applyOver` to pick the element three positions below the stack top. Here, the semantics of `pick2` is

$$\begin{aligned}
[\![\texttt{pick2}]\!]\, s{\cdot}x{\cdot}y{\cdot}z &= [\![\texttt{quote2 over applyOver}]\!]\, s{\cdot}x{\cdot}y{\cdot}z \\
&= [\![\texttt{over applyOver}]\!]\, s{\cdot}x{\cdot}(\mathsf{push}\ z \circ \mathsf{push}\ y) \\
&= [\![\texttt{applyOver}]\!]\, s{\cdot}x{\cdot}(\mathsf{push}\ z \circ \mathsf{push}\ y){\cdot}x \\
&= s{\cdot}x{\cdot}y{\cdot}z{\cdot}x
\end{aligned}$$

Most of the programming languages also support the assignment of values to names to ease the task of programming to the users. A somewhat standard technique to do this is to extend the state to contain also the definitions of assignments, i.e., **State** = **Stack** $\times$ **Defs** where **Defs** = **Name** $\to$ **Fun**. Similarly, the semantic clauses of existing programming constructs must be extended to work on the **State**, i.e., on its first component, while a new programming construct to support assignment must

also be defined. We refer the reader to [20] for a more detailed presentation on the technique supporting also variable scoping in an imperative language.
An interesting further research direction is to exploit the parallelism which is inherently present in concatenative programs. Both concatenation and composition are associative, hence, any order of execution, as well as parallel execution, produce the same result. For example

$$\begin{aligned}&[\![\texttt{3 4 add dup ispos 5 6 swap choose mul}]\!]\, s = \\ &\quad = [\![\texttt{5 6 swap choose mul}]\!] \circ [\![\texttt{3 4 add dup ispos}]\!]\, s\end{aligned}$$

Clearly, all operations are considered to be pure, i.e., without side effects. Associativity may also be exploited for program optimization and refactoring.

Finally, the implementation of a developer toolchain for concatenative language is quite straightforward. The syntax exhibits great simplicity and thus support easy parsing. Here, an interesting example is the Forth programming language parser which supports constructs to change itself. Additionally, stack-based evaluation is also straightforward as the stack is a standard structure directly supported by almost all modern computer architectures. Moreover, even the anonymous functions syntactically represented by quotations are easily represented and manipulated by pointers. To support these claims we developed a parser and evaluation engine of the KKJ language with some extensions (e.g. more primitive operations and support for function definitions) in the Haskell programming language. The corresponding source code is freely available under a permissive license at `https://www.github.com/jurem/kkj-lang`.

Conclusions

In this paper, we proposed and examined a simple yet powerful programming language which exhibits properties of both low-level, e.g., assembly, and, high-level, e.g., functional languages. Additionally, the term-rewriting view on the evaluation puts the language among functional ones while stack-based view puts it among imperative ones. We strongly believe that these dichotomies represent a great advantage to such languages and many features and properties are yet waiting to be discovered.
The discussed language syntactically and semantically belongs to a group of concatenative and compositional programming languages, respectively. We formally defined the syntax as well as the semantics of the language, and, hence, formed a basis for further theoretical investigation of concatenative programming languages whose formal treatment received little attention in the scientific literature.
Despite this, various variants of such languages are already present (however, usually missing the quotation construct) in the mainstream industry mostly as intermediate languages or as virtual-machine byte-code. In particular Java Virtual Machine (JVM), Common Language Infrastructure (CLI) and the Python virtual machine (PVM) are typical examples of common stack-based virtual machines. Even, until version 5.0, Lua's virtual machine was a stack-based machine; however, 5.0's virtual machine is a register machine. Another example is the GraalVM (virtual machine allowing polyglot features for JVM, Python and other languages).

Our main goal in the paper was to select the prominent features of concatenative languages and formally describe their meaning using tools and techniques from the field of denotational semantics. As we feel that the area deserves more scientific attention we also identified and proposed several possible research directions.

Acknowledgement

The first author was supported by the Slovak Academic Information Agency under the National Scholarship program during his research stay in 2018. The second and third authors were supported by the project KEGA 011TUKE-4/2020: "A development of the new semantic technologies in educating of young IT experts".

References

[1] A. Aiken, J. H. Williams, and Wimmers. "The FL project: The design of a functional language", 1991.

[2] M. Benčík and L. Dedera. "Natural semantics of battle management languages". In *2019 Communication and Information Technologies (KIT)*, pp. 1-4, 2019.

[3] O. Danvy and L. R. Nielsen. "Refocusing in reduction semantics". BRICS, Department of Computer Science, University of Aarhus, 2004.

[4] S. Diehl, P. Hartel, and P. Sestoft. "Abstract machines for programming language implementation". *Future Generation Computer Systems*, 16(7):739-751, 2000.

[5] A. K. Wright and M. Felleisen. "A syntactic approach to type soundness". *Journal Information and Computation*, 115(1):38-94, 1994.

[6] P. Haller and H. Miller. "A reduction semantics for direct-style asynchronous observables". *Journal of Logical and Algebraic Methods in Programming*, 105:75-111, 2019.

[7] D. Herzberg and T. Reichert. "Concatenative programming – An Overlooked Paradigm in Functional Programming". In *Proceedings of the 4th International Conference on Software and Data Technologies*, pp. 257-262, 2009.

[8] J.M.E. Hyland and C.-H.L. Ong. "On full abstraction for PCF: I, II, and III". *Information and Computation*, 163(2):285-408, 2000.

[9] T. Jones and M. Homer. "The practice of a compositional functional programming language". In *Proceedings of the 16th Asian Symposium on Programming Languages and Systems*, 2018.

[10] G. Kahn. "Natural semantics". In *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '87, page 22-39, Berlin, Heidelberg, 1987. Springer-Verlag.

[11] R. A. Kemmerer. "Hoare's axiomatic semantics". In *roceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis*, Clearwater Beach, Florida, 1997. ACM Press.

[12] C. Moore. “Forth: a new way to program a mini-computer”. *Astronomy and Astrophysics Supplement*, Vol. 15, pp. 497-511, 1974.

[13] P. D. Mosses. “Theory and practice of action semantics”. In *MFCS ’96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science*, pages 37-61. Springer-Verlag, 1996.

[14] H. R. Nielson and F. Nielson. *“Semantics with Applications: An Appetizer”*. Springer-Verlag London, 2007.

[15] S. Pestov, D. Ehrenberg, and J. Groff. “Factor: A dynamic stack-based programming language”. In *Proceedings of the 6th Symposium on Dynamic Languages*, DLS ’10, ACM, New York, NY, USA, pp. 43-58, 2010.

[16] B. C. Pierce. *“Types and Programming Languages”*. The MIT Press, 1st edition, 2002.

[17] G. D. Plotkin. “The origins of structural operational semantics”. *J. Log. Algebr. Program.*, Vol. 60-61, pp. 3-15, 2004.

[18] D. A. Schmidt. *“Denotational semantics. Methodology for Language Development”*. Allyn and Bacon, 1986.

[19] K. Slonneger and B. Kurtz. *“Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach”*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1995.

[20] W. Steingartner, V. Novitzká, M. Bačíková, and Š. Korečko. “New approach to categorical semantics for procedural languages”. *Computing and Informatics*, 36(6):1385-1414, 2017.

[21] S. Szymoniak. “Security protocols analysis including various time parameters”. *Mathematical Biosciences and Engineering*, 18(2): 1136-1153, 2021.

[22] O. Siedlecka-Lamch, S. Szymoniak, M. Kurkowski, I. El Fray. Towards Most Efficient Method for Untimed Security Protocols Verification. In: *Proceedings of the 24th Pacific Asia Conference on Information Systems: Information Systems (IS) for the Future, PACIS 2020*. 2020

[23] M. von Thun. “Joy: Forth’s functional cousin”. In *In Proceedings from the 17th EuroForth Conference*, 2001.

[24] D. A. Watt. “Action Semantics in Retrospect”. In Palsberg J. (eds) *Semantics and Algebraic Specification. Lecture Notes in Computer Science*, Vol. 5700, Springer, Berlin, Heidelberg, 2009.

[25] A. L. Zackery, S. Perugini. “An Introduction to Concatenative Programming in Factor”. *J. Comput. Sci. Coll.* 35(5):70-77, 2019. Consortium for Computing Sciences in Colleges, Evansville, IN, USA.