МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2011
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2011
Proceedings of the Fortieth Jubilee Spring Conference
of the Union of Bulgarian Mathematicians
Borovetz, April 5–9, 2011

A LANGUAGE FOR COMPOSITIONAL PROGRAMMING: A RATIONALE AND DESIGN*

Boyko B. Bantchev

A rationale and description of a language for exploratory and educational programming in a compositional style is presented. By 'compositional' a functional programming style is meant where the computation is a hierarchy of function compositions and applications. One of the datatypes of the language is that of the geometrical figures that can be obtained by simple rules of spatial correlation, thus, too, forming hierarchical compositions. The language is strongly influenced by GeomLab, but differs from it substantially in many respects. The paper discusses the main features of the language; the detailed description along with the picture construction facilities will be presented in an accompanying publication.

Whither educational computing? The programming language landscape is changing. One of the most noticeable tendencies is functional programming becoming more and more popular and available. Once the province of computing scientists studying programming language semantics or searching for the ever evading artificial intelligence, now functional programming is becoming ubiquitous. Gone are the days when by 'functional language' one had no other option to consider but Lisp or some of its dialects. Virtually any programming language that has emerged in the last two decades is equipped with facilities for functional programming, such as dynamically created functions, closures, and ability to carry out computations directly on bulk data (as opposed to piecewise processing). Mainstream languages like C++, C#, and Java are also rapidly growing such features. The pressure for more generic, more reliable, and amenable to parallelizing software is at the root of the readiness with which functional programming is embraced.

It is natural to expect that the education in computing is evolving in a similar direction, but that is not so. Particularly disturbing is the growing gap between the evolution trends in the modern understanding of computing and the modern languages and software, on the one hand, and what is being taught in schools at pre- college and university level, on the other. If programming is in the curriculum at all, BASIC, Logo and Pascal continue to be the dominating languages. These days, C++ or Java are a possible option taken by those who want to make a stress on object orientation – a disputable choice in itself.

*2000 **Mathematics Subject Classification:** 68N15, 68N18.
Key words: programming languages, functional programming.

A number of programming environments for young pupils – e. g. Alice, Scratch, ToonTalk, StarLogo TNG, Sugar, eToys, Kodu, Stagecast – have been created that aim at removing the need to cope with syntax in programming by replacing the writing of text with manipulation of correspondingly interacting images. The program is guaranteed to be syntactically correct by allowing only meaningful compositions of visually represented commands. The pupils are able to run animations, do simulations, 'tell stories', and play games. Sound and video may be available as building blocks. However, such systems tend to offer only too primitive programming abstractions, of imperative nature, such as variables and explicit loops. For example, in Scratch [4] – which seems to be particularly popular in this class – there are no functions or procedures, no recursion or other kind of self-reference, and no operations on bulk data. Therefore, although allowing superficially complex behaviour to be achieved by little effort, as programming systems such visual environments are plain basic, cannot scale beyond very small scripts, and from this perspective cannot be really attractive.

On the other hand, the immediate feedback and the mere appeal of the visual are something worth making use of, possibly in other kinds of programming environments. Can we, in particular, meet the mathematical purity, expressiveness, solidity, and elegance that characterize a functional programming style with the richness and ease of perception uniquely available to us through representing information visually?

Successful findings on this track would be beneficial to educational and exploratory programming at the least.

Years ago, P. Henderson described [2] a 'functional geometry': a set of operations through which pictures could be composed from simpler ones. A picture could be rotated and laid above, beside, or over another one. By defining picture-valued functions and making use of recursion, functional geometry is a system of describing pictures – including instructing a computer how to draw such pictures – which is purely declarative and sufficiently powerful. Declarative, as one writes denotations of what is to be constructed, rather than procedural descriptions of how to perform the construction. Powerful, as it proved to elegantly cope with pictures as complicated as some of the M. Escher's drawings. The approach was considered successful enough to deserve redoing of the original paper twenty years later [3].

An important characteristic of this functional geometry, accented by Henderson, is it being a formal algebraic system: an algebra of pictures, in which they are computed according to operational specifications, the latter serving as both a most clear description and practical implementation.

Henderson himself never implemented his idea in a program, but in the recent years the functional geometry found its realization in GeomLab [5]. The latter was conceived as a programming system for education of, and exploration by, students up to secondary-school level, but it is considered generally useful to beginner-level programmers, including people only wishing to acquaint themselves with the subject of programming, such as humanists.

GeomLab is a simple functional programming language and environment where the result of evaluating an expression is displayed immediately, without explicit action: numeric and textual data is printed, and pictures are shown in a separate window. Complex pictures are considered a metaphor for the complex behaviours seen in computer programs. Links with mathematics and computing science can be made and accordingly

elaborated while immersing in GeomLab programming.

Learning of GeomLab, and experimenting for some time with it, resonated with the author's long-standing conviction that functional programming could and should be used in the school curricula for students of all ages. The author has himself argued before on the latter topic [1]. The gained experience with GeomLab, and the realization of some of its inconveniences, was felt as an urge towards searching for a similar, but even more expressive language.

The following sections describe the result of this effort.

Evolving a language. A language suitable for exploration and education in computing is bound to have, in author's opinion, the following essential properties:

- expressivity: ability to convey a wide range of data and computational structures;
- straightforwardness: ability to do so directly, with as little deviation and bureaucracy as possible;
- readability;
- dynamic typing;
- an interactive, interpreted system.

The first two properties ensure that the language is an adequate modelling tool for a large number of programming problems and domains. Readability is especially important in the educational context. Dynamic typing is generally preferable to static typing: the flexibility that usually comes with dynamic typing does a better service to an exploring or teaching or learning programmer than the additional level of control from the language processor that is enabled through static typing. Similarly, an interactive, interpreted system is greatly advantageous (again, for the said class of programmers) over compiled languages.

Other properties, notably those related to "system(s) programming", such as execution speed or elaborate modular facilities, are immaterial.

Simplicity, though hard to define for its elusiveness, is a feature of paramount importance that should permeate all other properties of an exploratory or educational language. It concerns the structuring of both data and computation. Solutions to intuitively simple problems should be achieved easily and expressed succinctly. There should be relatively few language constructs and rules for combining them – although the possible outcomes of those combinations should be sufficiently diverse, lest versatility is compromised.

The above bill of essential properties can be fitted perfectly by a functional programming language. The functional style provides a single but highly expressive and direct program structuring concept: compositions of functions and their applications. In GeomLab, the functional approach combines with capabilities for generating images. Differently from other programming environments, GeomLab's drawing is strictly subordinate to purpose. Pictures are created through composition, a process that usually requires careful planning, inductive reasoning, and building layered abstractions – all these are characteristically programmer's virtues. Along the way, a student or explorer can also get in touch with geometry, group theory, combinatorics, formal logic, and other mathematical disciplines. It is the intellectual curiosity and sense of challenge that drives the work with this system, rather than merely the desire of getting a picture drawn.

However, GeomLab has certain deficiencies as a language. For one thing, GeomLab's only data structure is the heterogeneous, unidirectional linked list. Being asymmetric,

such lists can only be augmented or reduced from one of their ends, which limits their usefulness. To many, they are also less intuitive, compared to, e. g., random access sequences.

Another limitation is the lack of direct support for some constructs and techniques that are commonplace in functional programming. Neither function application nor composition are built-in operators. Function calls, as well as function definitions, suffer certain restrictions. No mechanism for partial application is provided. Missing are library functions for expressing a number of well known computational patterns and idioms.

Yet another set of problems in GeomLab is related to datatype handling, such as the inability for reliable discrimination between different types and underdevelopment of some of them, viz. Booleans, strings, and pictures. For example, pictures are structureless data: once a picture is constructed, it becomes an atomic value. Its constituting components and their transformations are not preserved in it. Not only one cannot programmatically obtain them, but even if they are known, they are inaccessible.

Some of the said limitations can be overcome by employing user-defined operators, anonymous functions and lexical closures. However, this leads to seriously bending the language without solving all the problems.

GeomLab also has some syntactic and lexical issues pertaining to binding, function definition, and expression structure. Circumventing these misfeatures is only possible at the cost of bloating or twisting the user program.

Consideration of the weak points of GeomLab initially lead the present author to building a layer of improvement over it, within GeomLab itself, which then was used in a number of sample programs to test how the programming was influenced by the changes. Eventually, the accumulated observation and reflection in the process of doing this brought to the belief that a completely new design was needed.

The resulting language is partially and informally presented in the following section. In many ways, it is a strong departure from GeomLab, carrying influences from a number of other languages, and also exhibiting several novel features of its own. The language was given the working name FunCo, for 'functional composer' and will be described in more detail in a publication to follow the current one.

The FunCo language. FunCo is a purely functional language, in which primitive datatypes are floating-point numbers, Booleans, strings, and symbols. The principal compound datatype is that of random-access sequences. The individual entries within a sequence are referred to by indices, and each can be a value of any type, including a sequence. Pictures can be primitive or compound. There are some predefined primitive pictures, but the programmer can also create new ones as needed. A compound picture is one composed as in GeomLab (and the original functional geometry), but in FunCo such a picture can be decomposed into parts by pattern matching much like sequences can.

Sequences can be used to model datatypes from other languages, such as tuples, structures, associative tables, hierarchical and (through the reference semantics of argument passing) linked structures.

The set of operators in FunCo is larger than that of most languages. Many of the operators are ambivalent, having both a monadic and dyadic cases. Extensive use is made of type polymorphism. For example, ~ denotes logical negation, reversal, or argument

commutation, depending on whether the operand is a Boolean, a sequence, or a function. Similarly, the dyad of `|` denotes logical disjunction of Booleans or the 'beside' operator on figures, while the monad is the magnitude operator on numbers.

FunCo strives for being a hospitable environment for programming in different domains, not just picture composition. To this end, it offers many operators for computing with numbers and sequences. Many of the list manipulation functions known from a typical functional language, such as e. g. Haskell's *head*, *tail*, *map*, *filter*, *zip*, *find*, *lookup*, *foldl*, etc. are implemented in FunCo as operators on sequences. For example, if `q` is a sequence and `p` a predicate, the expression `f!(10<(p#q))` applies a function `f` to each member of the sub-sequence of the first 10 items of `q` for which `p` is true. (An expression in Haskell with a similar meaning would be `map f (take 10 (filter p q))`.)

All operators on sequences can be used on strings, so that programming with text takes full advantage of the variety of ways to process sequences.

As the said functions correspond to the most frequently occurring patterns of recursion on sequences, a FunCo programmer would seldom have to resort to explicitly recursive definitions, mostly composing their programs as simple expressions. Implementing these functions as operators brings the additional benefit of having a terser and more convenient (infix) syntax available in place of the prefix-form function applications in most other languages. Moreover, as some of the operators denote higher-order functions, it is straightforward to obtain function-valued sub-expressions and then apply the respective functions, all within the uniform syntax of the infix expressions. Thus, a function-level programming style is strongly supported.

An important distinguishing feature of FunCo is that function application itself is an infix operator, denoted with a dot sign, so to apply a function `f` to an argument `x` one writes `f.x`.

Furthermore, each function has a single argument which is a sequence. Because of that, no special syntax is needed for denoting argument tuples in function calls. There are no tuples at all (no multiple arguments), but the single argument being a sequence means that a function can actually be defined to accept as many arguments as needed, including none or a varying number of them. If an argument is not a sequence, it is implicitly put into one. Thus, functions become syntactically and semantically simpler, acquiring a greater potential for flexibility along with that.

Monadic operators also apply only through the `.` operator, e. g. the arithmetic negation of `x` is written `-.x`. Therefore, syntactically, functions and monadic operators are arguments to the 'apply' operator.

Ultimately, expressions become richer and yet syntactically simpler. The richness comes from the variety of useful operators and their combination. Simplicity is achieved by admitting only a single construct from which all expressions are eventually formed: application of an infix operator to its arguments. Semantic structure is still imposed on expressions by rules of precedence.

An operator or a function can be partially applied, producing a more specific operator or function. Partial applications of operators preserve type polymorphism wherever the latter is present in the operator. For example, `_>x` applies partially `>` to `x` as a second argument, producing a function of one argument, `>`'s left one. If `x` is a number, `>` is interpreted as 'greater-than' and the resulting function's argument must be a number to compare `x` to. If `x` is a sequence, `>` denotes either 'along-from-left' (see the exam-

ples below) or 'take/drop' (as seen above), which remains ambivalent until the resulting function `_>x` is applied to a function or a number, respectively.

Another way in which simplicity is achieved in FunCo is not providing a syntactic construct for function definition, other than a statement for binding a name to a value of arbitrary type: in order to define a function, one simply binds a name to a function value. This approach works well due to being possible to construct function-valued expressions in many ways – using higher-order operators, partial applications, and explicitly parametrized anonymous functions. For instance, `sum == +>_` defines `sum` as a function that sums up a sequence, and that function is obtained by partially applying `>` to `+`. Similarly, `*>_` and `/\>_` are expressions for anonymous functions computing the product and the maximum of a sequence of numbers, and `-<_` is a function computing an alternating sum $x_1 - (x_2 - (x_3 - (\cdots))) = x_1 - x_2 + x_3 - \cdots$, applying '`-`' backwards along a sequence. As with `sum`, each of the expressions could be given a name if that is needed.

Another example of binding a name to a function is `all == @[p;xs]::&>(p!xs)`: here, `@` denotes an anonymous-function expression where the argument is matched against a sequence of two elements, `p` and `xs`. The predicate `p` is applied to all members of `xs` and the resulting Boolean sequence is '*and*-ed up' (`&` denotes conjunction).

Here are some more examples of expressions and function definitions in FunCo. Lack of space prohibits giving examples with pictures; that will be done elsewhere.

• Similarly to the above mentioned sum, product, etc. functions, the following one computes a polynomial, specified as a sequence of coefficients `cs`, at an argument `x`:

```
poly == @[cs;x] :: (@[c;v]::c*x+v) > cs
```

For example, `poly.[[2;3;-5;1];2]` would yield 19. Computation is done by applying the Horner's rule, implemented by the function `@[c;v]::c*x+v` which is passed to `>`. (For the sample sequence `[2;3;-5;1]`, $2\,x^3 + 3\,x^2 - 5\,x + 1$ is computed at 2, as $((2 \cdot 2 + 3) \cdot 2 - 5) \cdot 2 + 1$.)

• The dot operator with its left argument a sequence denotes indexing, the right argument being an integer; e. g. `q.3` yields the entry at index 3 in `q`. Partial application of `.`, passed as an argument to `!` ('apply to all'), such as in `q._![2,3,2,5]`, yields a sequence of `q`'s entries under the given indices.

• An expression for the factorial function can be defined in any of the ways:

```
@[n] n>0:: n*@.(n-1)
^    n=0:: 1
```

,

```
@[n]:: {? n=0; 1;
          n*@.(n-1) }
```

,

```
@[n]:: *>(1..n).
```

The first is an explicitly parametrized function of two clauses, where the parameter's value is guard-checked in order to choose one of the two expressions `n*@.(n-1)` and `1`, the value of which is to be returned. Note that `@` is a formal name referring to the function being defined, i. e. denoting a recursive call. The second form makes use of a conditional expression rather than guarded clauses, but otherwise specifies the same computation. The third expression computes the product of the arithmetic sequence $1, 2, \ldots, n$.

• The following:

```
@[p;xs] :: #.(p#xs)
```

is a function that finds the number of those elements of a sequence `xs` that satisfy a predicate `p`. It actually finds the sequence of those items first (`p#xs`), and then computes its length (`#`).

• The function `insert`, defined:

```
insert == @[x;[]]        :: x
          ^[x;y\ys] x=<y :: x\y\ys
          ^         "t   :: y\@.[x;ys]
```

puts a number, `x`, in a non-decreasing sequence, preserving the order in the sequence. The definition includes three clauses: one for inserting in an empty sequence, and two for the cases when `x` does not exceed the first element `y` of the sequence and when it does. It should be noted that the last two clauses share the pattern for a non-empty sequence (`[x;y\ys]`), but the last clause is guarded by an explicitly stated condition that is constantly true. (The guard of the first clause is also true, but there is no need there to write that explicitly.)

• Having defined `insert`, sorting a sequence in ascending order can be done by the function

```
@xs :: insert<(xs/[])
```

(insertion sort), which applies `insert` backwards (`<`) along `xs`, after appending an empty sequence to the rear of `xs` to ensure that the insertion process starts properly. The same function can be defined entirely at function-level, leaving its argument implicit:

```
(_/[]) >> (insert<_)
```

The latter expression is a composition (`>>`) of `_/[]`, which reads 'the function that appends an empty sequence', and `insert<_`, the 'function that does `insert`, scanning a sequence from end to start'. Both functions are partial applications.

• Sorting a sequence using the 'quicksort' method:

```
@[]     :: []
^(x\xs) :: @.((_=<x)#xs) + [x] + @.((x<_)#xs)
```

where the partitioning is done using the already seen 'filter' operator (`#`) twice, with suitable comparison functions, themselves partial applications. The '`+`' here denotes concatenation.

• Here is a function computing the Cartesian product of a sequence of sequences, each one arbitrarily long. It uses 'apply-to-each' (`!`) twice, and 'along-from-left' (`>`) with concatenation (`+`):

```
cprod == @[]       :: [[]]
         ^(xs\xss) :: + > ((@[x] :: (x\_)!@@.xss) ! xs)
```

The function `@[x]::(x\_)!@@.xss` is responsible for performing a single step in accumulating the product and is mutually recursive with the main one, hence it makes use of `@@` for calling it: `@@` means 'the outer function' just as `@` is 'this function'.

Implementation and further development. By the moment of writing this, an implementation of FunCo has just started. A small research was carried out for choosing an implementation language, with respect to which the following requirements were considered important:

- portability across the several most popular operating systems;
- support for automatic memory management;
- support for lexical closures;
- portable GUI building capabilities across the targeted operating systems;
- portable vector graphics capabilities across the targeted operating systems;
- ability to produce a compiled executable (an interpretive implementation was considered undesirable, as the user would have have to install the implementation

language along with FunCo to be able to use the latter);
– sufficiently good performance.

Somewhat surprisingly, it was found out that, however modest the requirements, they were all indisputably answered by only one language, Lua. Most languages do not have standardized GUI and drawing facilities, and compiled ones are mostly weak at dealing with closures. C++, in addition, has no automatic memory management. Interpretive and semi-compiled languages also suffer some of these deficiencies, along with making it very hard or impossible to create a stand-alone executable (Ruby, Python, Lisp, Scheme, Perl, Factor, ...). If Lua were absent, Java would have been an acceptable compromise, but its imitation of closures is clunky, and its use places the prerequisite of having a JVM installed.

Exceptional performance of the FunCo implementation is not a goal, so it is acceptable to translate FunCo to Lua and compile the resulting code into the Lua virtual machine using the compiler built-in in that language. As Lua is an embeddable language, it can be called by a program in C, thus eventually obtaining a single executable file.

Once an implementation of FunCo, along with a suitable user environment for it, is complete, several directions of future development are possible, such as: studying the user acceptance, and accordingly adding, changing or removing language features, if needed; improving the user interface; growing libraries in the language, e. g. for different application domains.

Certain facilities, such as for general I/O, tracing, and exception handling are considered useful but of secondary importance, and could be added once the language stabilizes.

REFERENCES

[1] B. Bantchev. Functional programming for the high-school curriculum. *Math. and Education in Math.*, **32** (2003) 305–311.

[2] P. Henderson. Functional geometry. Proc. 1982 Symp. on Lisp and functional programming, 1982, 179–187.

[3] P. Henderson. Functional geometry. *Higher Order and Symbolic Computation*, **15** (2002), 349-365.

[4] Scratch programming language website: `http://scratch.mit.edu`.

[5] M. Spivey. GeomLab – exploring computer science. `http://web.comlab.ox.ac.uk/geomlab`.

Boyko B. Bantchev
Institute of Mathematics and Informatics
Bulgarian Academy of Sciences
Acad. G. Bontchev Str., Bl. 8
1113 Sofia, Bulgaria
e-mail: bantchev@math.bas.bg

ЕЗИК ЗА КОМПОЗИЦИОННО ПРОГРАМИРАНЕ: ОБОСНОВКА И КОНСТРУКЦИЯ

Бойко Бл. Банчев

Представена е обосновка и описание на език за програмиране в композиционен стил за опитни и учебни цели. Под "композиционен" имаме предвид функционален стил на програмиране, при който пресмятането е йерархия от композиции и прилагания на функции. Един от данновите типове на езика е този на геометричните фигури, които могат да бъдат получавани чрез прости правила за съотнасяне и така също образуват йерархични композиции. Езикът е силно повлиян от GeomLab, но по редица свойства се различава от него значително. Статията разглежда основните черти на езика; подробното му описание и фигурноконструктивните му възможности ще бъдат представени в съпътстваща публикация.