Functional Block Programming and Debugging

Dimitri Racordon
University of Geneva
Computer Science Department
Switzerland
dimitri.racordon@unige.ch

Emmanouela Stachtiari
University of Geneva
Computer Science Department
Switzerland
emmanouela.stachtiari@unige.ch

Damien Morard
University of Geneva
Computer Science Department
Switzerland
damien.morard@unige.ch

Didier Buchs
University of Geneva
Computer Science Department
Switzerland
didier.buchs@unige.ch

ABSTRACT

Recent years have seen the inclusion of introductory programming courses as early as in elementary schools, as an answer to the onminpresence of computer technology. While some curriculums rely on traditional text-based languages, such as Java or Python, others advocate for more accessible visual programming environments, such as Scratch or Microsoft MakeCode. Most of these tools adopt an imperative paradigm in which programs are expressed using pre-defined instruction blocks. This setup is particularly well suited to interactive debugging environments, using the common stepping model to control execution. However, as they focus on control rather than data, they often lack appropriate mechanisms to manipulate non-trivial data structures.

In an effort to provide an alternative approach, we are developing FunBlocks, a hybrid programming environment based on algebraic data types and term rewriting. Rewriting systems emphasis on data rather than control, enabling students and novices to create a better understanding of data structures, while letting them define elaborate transformation strategies without the need to get familiar with the ad-hoc semantics of pre-defined control-flow statements. Instead, terms describe the program's state syntactically, while rewriting rules describe its semantics. Furthermore, terms and rewriting rules also provide a simple way to adopt the traditional watch expressions and step-by-step execution, respectively, whereas such techniques are often difficult to apply on declarative programs. This paper gives an overview of the design principles of FunBlocks, with a particular focus on its visual editor and debugger.

KEYWORDS

visual programming

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Dimitri Racordon, Emmanouela Stachtiari, Damien Morard, and Didier Buchs. 2020. Functional Block Programming and Debugging. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The ever increasing influence of computer technology on everyday lives has motivated the inclusion of introductory courses on computer science in the classrooms. Most curriculums focus heavily on programming, often by the means of a visual environment to facilitate media manipulation for novices and young students. Popular environments include Scratch [16], Microsoft MakeCode [3] and Lego Mindstorms [6]. These tools typically offer to write programs using pre-defined instruction blocks, in an attempt to free users from the burden of mastering a textual syntax. The rationale supporting this approach is based on the observation that syntax errors are reliably identified by compilers and are usually better understood than errors related to a program's semantics [19].

Most learning tools adopt an imperative paradigm, in which a program is expressed as a sequence of instructions. While such a choice seems obvious, as it reminisces mainstream structured programming languages (e.g., Python or Java), it forces a strong emphasis on control structures while eliding data structures. As a consequence, users are encouraged to focus practically exclusively on the way they are supposed to solve a problem (i.e., "how") rather than thinking about the nature of the problem (i.e., "what"). We argue that this asymmetry hinders one's ability to create mental representations of complex problems [24], which is an essential skill to write high quality software programs and be able to efficiently identify and repair errors in source code [30]. Furthermore, the imperative paradigm itself comes with several challenging subtelties. Three issues stand out in particular:

- The *state* of the program is an abstract notion that is not visible in its syntax. Hence, one has to build a mental model of the program's execution to understand how its state interacts with each instruction.
- State modifications require the use of assignment statements, whose semantics can vary a lot from one language to the

other, especially in the presence of aliasing [14]. This impedes one's ability to form correct assumptions about the precise meaning of an instruction.
- The imperative paradigm is unable to express its own meta-theoretical model of computation and therefore cannot elegantly describe control structures. Instead, it assumes the existence of numerous "built-in" statements (such as *if then else*, *while*,...), whose semantics must be defined ad hoc.

As an alternative, we propose to adopt a declarative paradigm, in which a program is expressed as a set of rules that can be used to transform some initial data into a satisfying solution. To this end, we are developing FunBlocks (for **Fun**ctional **Blocks**), a web-based programming environment that offers to write programs in the form of rewriting rules [7]. Although such an approach may not seem as intuitive as the more traditional imperative paradigm [18], it provides an arguably better support to explore fundamental aspects of computability theory. In fact, programs in FunBlocks are reminiscent of structural operational semantics [1] (a.k.a. small-step semantics). FunBlocks provides both a graphical and textual representation of a program and allows the user to switch seamlessly from one to the other. The textual syntax heavily borrows from Haskell's while the graphical syntax allows the user to write rewriting rules using nested blocks to describe terms and variables. The tool also comes equipped with a debugger that lets the user to finely control the evaluation of a program with step-by-step execution and rewinding. To the best of our knowledge, FunBlocks is the first educational tool to offer a hybrid programming environment based on a declarative paradigm that supports such debugging features.

This paper reports on the core design principles of FunBlocks and elaborates on ideas to refine its user experience, with a particular focus on its visual editor and debugger.

2 RELATED WORK

Our work is heavily inspired by the large collection of visual programming environments aimed at computer science education. Most popular entries in this category include Microsoft MakeCode [3], Scratch [16], Blockly [11] and Lego Mindstorms [6]. All provide the user with a visual language in which pre-defined blocks can be dragged and dropped to create sequences of instructions. Control structures (e.g., conditions, loops, etc.) are typically represented by blocks featuring "holes" which should be filled with other blocks. User interfaces generally consist of a web-based editor, paired with some sort of area that depicts the objects being manipulated by the program (e.g., a character moving in a 2D maze).

All these educational tools, including FunBlocks, are based on the assumption that visual, block-based programming can prove beneficial to help novices getting started with programming. This intuition has been validated to various extents in different studies which have notably shown that such educational tools lower the entry barrier to programming [23, 29] and may improve its attractiveness [17, 31]. However, some studies also reported that block-based environments did not lead to significant differences in understanding in performance once learners moved beyond basic mechanisms [2, 13, 28], suggesting a need for better tools to convey more complex concepts, in particular pertaining to semantics.

Although most environments provide tools to translate a program developed with their visual language into its equivalent in a traditional textual language (e.g., Javascript), they generally offer a very limited support to *edit* programs in both its visual and textual representations in a truly hybrid fashion. Notable counter-examples include Microsoft MakeCode [3], which can switch back and forth between its visual block-based language and TypeScript, a typed extension of JavaScript. Pencil code [5] and Tiled Grace [15] both offer a similar feature, but rely on dedicated textual languages rather than targeting a mainstream option.

While visual programming environments for education overwhelming favor the imperative paradigm, some tools have been built on top of functional data-flow languages. Camaleon [10] and Luna [20] are two good representative of this approach, although not aimed at education. A program is defined as a directed graph whose vertices represent operators (or functions) and edges denote how data is *flows* between them. Data flow graphs present an intuitive view of transformations, as they eliminate the need to deal with the abstract notion of state related to an imperative program's execution. However, they are not well suited to describe iteration and recursion, requiring complex mechanisms to break dependency cycles [9]. Furthermore, they do not intrinsically address the issue of data structure representation.

One line of work leaning toward declarative programming propose to extend existing languages with graphical interpretations. An early representative of this idea is VEX [8], which extends the lambda calculus with a graphical syntax. More recent works include TypeBlocks [27], based on on standard ML, and Haskell block expressions [22]. These approaches provide a better support to represent and manipulate non-trivial data structures, as they leverage the expressiveness of the underyling (functional) language. Unfortunately, they have yet to materialize into modern implementations of interactive, educational programming environments. FunBlocks can be seen as an attempt to fill this void.

3 FUNBLOCKS

FunBlocks is a programming environment designed to support teaching activies related to the fundamental principles of programming and computability theory. It presents itself as a web-based application, compatible with most modern browsers, which offers users to write, run and debug computer programs.

3.1 Model of Computation

Unlike most educational programming environments, FunBlocks adopts a declarative (or functional) paradigm, based on term rewriting [7]. At the tool's core is the notion of *term*, which is used to represent structured data. Terms are finite recursive structures, composed of constant symbols, variables and function symbols. For example, the arithmetic expression $\pi \times r^2$ can be represented by a term of the form $mul(pi, square(\$r))$. Here, the term is built from the constant π, the variable $\$r$ and the function symbols *mul* and *square*. A program is expressed as a set of rewriting rules, which describe how to transform (i.e., *rewrite*) a term to another. A rule is essentially defined as a pair of terms, whose first element denotes a

pattern for the rule's inputs and second element specifies how the input should be modified. For example, a rule that describes the area of a disk could be expressed as $area(disk(\$r)) \rightarrow mul(\pi, square(\$r))$.

Figure 1: Rewriting rule computing a disk's area.

FunBlocks offers a textual and visual syntax to write programs, as well as the ability to switch from one representation to the other at any moment. The visual syntax allows terms and rewriting rules to be defined by nesting blocks representing constant and variables, which can be moved around using drag-and-drop. An example is depicted in Figure 1, which corresponds to the graphical representation of the area computing rule mentioned earlier. Rounded rectangles represent constants and function symbols, whereas hexagons represent variables. A block graphically nested within an other is interpreted as a sub-term of the latter or, in other words, as one argument of a function symbol.

The left part of a rule, called its *pattern*, specifies the form of the terms that can be rewritten. In Figure 1, for example, the rule matches any term of the form $area(disk(\$r))$, where the variable $\$r$ acts as a placeholder for any term. For instance, the rule's pattern would match the term $area(disk(five))$ as well as the term $area(disk(add(four, two))$. However, it would not match the term $area(square(five))$. The right part of a rule, called its *conclusion*, is a recipe that specifies how matched terms should be transformed. The rewriting process simply consists of substituting variables matched from the pattern in the conclusion, if any. For instance, applying the rule on the term $area(disk(add(four, two))$ would yield the term $mul(pi, square(add(four, two)))$, because the variable $\$r$ is matched to $add(four, two)$ in the pattern. From there, further transformations, expressed by additional rules, could eventually reduce the term to a final, canonical form.

3.2 Type System

At a more abstract level, a rewriting rule can be understood as the description of a function's semantics for a particular set of inputs. FunBlocks' type system captures this intuition and offers to type a set of rewriting rules as if all were "cases" of some function. This approach prescribes that rule patterns be function symbols, i.e., terms of the form $f(a_1, \ldots, a_n)$, where f is interpreted as a function name. It follows that a function's domain describes acceptable subterms, whereas its codomain describes the terms produced after a rule application. For example, let $area : Shape \rightarrow \mathbb{R}$ denote a function that computes the area of a shape. Then the rewriting rule from Figure 1 is a "case" of this function, relevant when the input is a disk. This suggests that the type *Shape* would be defined as the union of different shapes, for instance, disks and squares.

Typing can be a relatively difficult concept to master, in particular for novice users with little to no prior programming experience. Furthermore, due to its emphasis on data structures, FunBlocks requires a relatively elaborate type system, featuring sum types (a.k.a. tagged unions), product types (a.k.a. records) and polymorphic (a.k.a. generic) types. Understanding the theoretical fundations behind these concepts require a solid background in first-order logic, while appreciating their value in practice requires experience. Hence, to minimize the cognitive overload on learning users, FunBlocks implements a *gradual* type system [25]. A gradual type system supports both untyped and typed terms to coexist in the same program and only applies static type checking for the latter ones. This approach lets advanced typing concepts to be introduced progressively or be eluded altogether.

At the moment, types can only be defined using FunBlocks' textual syntax. Nonetheless, type-checked terms are visually differenciated from untyped expressions in the visual editor. Each defined type is associated with a color which is used to draw typed terms. On the other hand, untyped terms are drawn in gray.

3.3 Program Evaluation

The state of a program is represented by a single term. Executing a program amounts to successively applying rewriting rules on this term until it reaches a final, canonical form. FunBlocks offers two different ways to run a program, an automated and an interactive one. The former consists of automatically selecting and applying an appropriate rule (i.e., a rule whose pattern matches the current state) until the program's state cannot be rewritten further (i.e., when the current state cannot be matched by any rule pattern). This evaluation mechanism departs from the traditional sequential interpretation of a program. Since FunBlocks does not assume any reduction strategy (e.g., innermost, outermost, etc.), any (sub-)term can be rewritten at any execution step. Furthermore, since rewriting rules are defined without any particular order and their patterns are not required to be mutually exclusive, there is no guarantee on which specific rule will be used to rewrite a specific term. All the above design choices provides the practical foundations to discuss and experiment with key properties in computability theory, such as termination, confluence and determinism.

The interactive evaluation method lets the user manually choose which rule they wish to apply at each execution step. This feature corresponds to FunBlocks' debugger. Figure 2 shows a screenshot of the tool, in the middle of a program execution. The bottom area of the interface shows all the rewriting rules defined in the program, which can be selected with a click or tap. Once a rule has been selected, the user may try to apply it on the (sub-)term of their choice by clicking or tapping on it from the program's current state. If the selected (sub-)term matches the pattern of the selected rule, it is rewritten and the current state is updated. The completion of an evaluation step is visually signaled by the addition of a circular button in the history area. If the selected (sub-)term does not match, then the tool provides a visual cue that the desired operation is illegal and does not progress further. Clicking on any button from the history lets the user jump backward or forward to a specific point in the program's evaluation. The user is then allowed to inspect the state of the program and optionally to choose a different execution path, either by selecting and applying a different rewriting rule or a different (sub-)term.

4 MAKING FUNBLOCKS ALIVE

One future direction we envision for FunBlocks is to provide users with more hints about the computability properties of their program, as it is being edited and/or executed. For example, the user

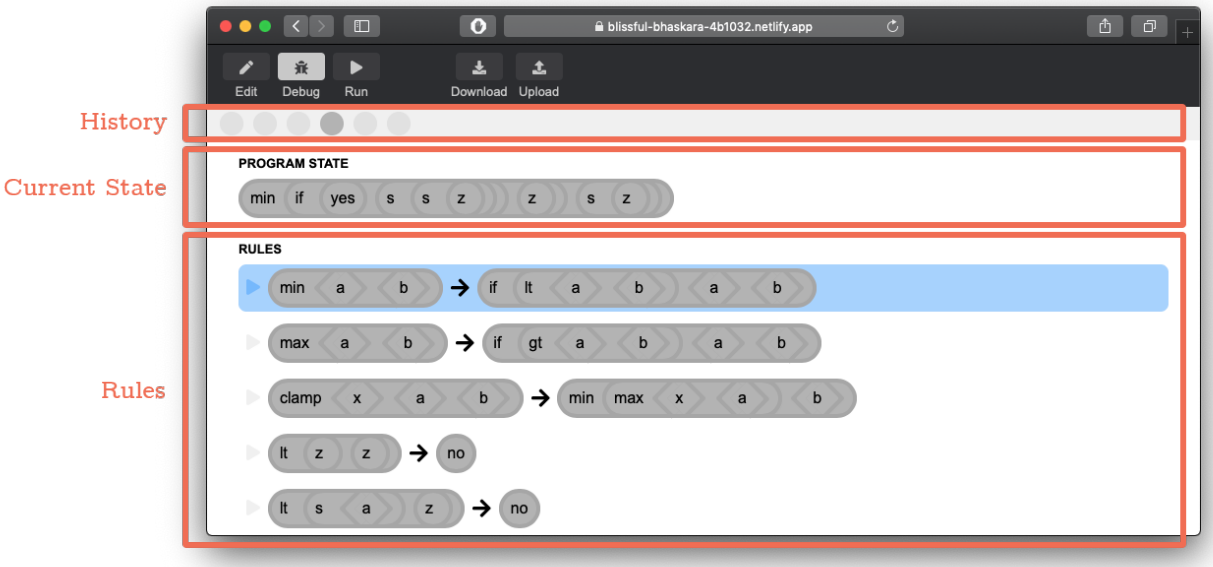

Figure 2: Screenshot of FunBlocks' debugger interface.

could be informed that the addition of a specific rewriting rule has made the program non-deterministic (static hint), or that a particular evaluation step has led to a non-terminating execution (runtime hint). Hints could also illustrate concrete examples of the mechanisms which are used by compilers and runtime systems to check for a program's correctness. This section discusses the directions we plan to explore on computing such hints.

While determining computability properties such as termination and confluence is undecidable in the general case, weaker results (e.g., yes/no/maybe answers) could still provide users with valuable insights about their program. It would also lay out a practical framework to discuss decidability in itself.

4.1 Static Evaluation

The metamodel of FunBlocks' evaluation mechanism being based solely on rewriting rules, programs can be analyzed by the means of any classical framework associated with term rewriting systems. In particular, we can exploit the concept of *critical pairs* to establish semi-decisions on determinism, confluence and termination. In a rewriting system, a situation in which two rewriting rules overlap but yield different terms is called a critical pair. For instance, consider the following program, composed of two rules:

$$\texttt{case } add(succ(\$r), zero) \longrightarrow succ(\$r)$$
$$\texttt{case } add(\$r, zero) \longrightarrow \$r$$

The program contains a single critical pair $\langle succ(\$r), \$r\rangle$, as both these terms can be derived from the application of different rules to the term $add(succ(\$r), zero)$. The fact that two program rules can match the same aforementioned term, leads to a random choice of which rule to apply and which term we will have in a single evaluation step.

The presence or absence of critical pairs in a program determines whether or not its evaluations are deterministic. However, it is insufficient for determining its confluence, as both elements of a critical pair might converge. This is the case in the above program, because both rules are in fact converging. Nonetheless, highlighting possible divergence points in the program editor could help the user conclude that their program is actually confluent, in cases where the critical pair analysis produces a false negative, or determine which rules invalidate the property.

Some recursive patterns without termination can be statically detected by unfolding backward and forward a set of rewriting rules (based on sureduction). Only loop based non-terminating programs can be detected this way. Other principles should be considered for diverging non-termination in general, typically based on complexity measures determined by heuristics [21].

4.2 Leveraging Types

Thanks to the use of algebraic data types, providing type annotations for a particular set of rewriting rules enables FunBlocks' internal type checker to determine its completeness. Given a function $f : A \longrightarrow B$, the set of rewriting rules defining f's semantics is *complete* if for each element of $a \in A$ there exists at least one rule whose pattern matches a. For example, consider the following FunBlocks program:

$$\texttt{type } Tree\ \$t :: empty \mid leaf\ \$t \mid node\ (Tree\ \$t)\ (Tree\ \$t)$$
$$\texttt{func } depth\ \$t :: Tree\ \$t \longrightarrow \texttt{Nat}$$
$$\texttt{case } depth(empty) \longrightarrow zero$$
$$\texttt{case } depth(node(\$a, \$b)) \longrightarrow succ(max(depth(\$a), depth(\$b)))$$

The generic type *Tree* defines binary trees as a union of three type constructors: *empty* which denotes an empty tree, *leaf* $t which denotes leaf nodes containing an element of type $t and *node* (*Tree* $t) (*Tree* $t) which denotes branch nodes with two children. The function *depth* is expected to accept a tree and return its depth. However, none of the rewriting rules describing its semantics can match leaf nodes. Thus, the set is actually incomplete, turning *depth* into a partial function.

In general, whether incompleteness constitutes an error depends on the desired semantics of the program (or function). In the example above, it would be reasonable to interpret it as the unintended omission of a rule that deals with leaf nodes. Nonetheless, there exist cases where using incomplete definitions to define partial functions would be sensical. For instance, the function that returns the position of an element in an array could be intentionally partial, being defined only for the elements that are present in the array.

The formalization of partial functions in FunBlocks remains an open question from a pedagogical point of view. In its current state, incompletess is interpreted as a failure, since it prevents the evaluation mechanism from reaching a normal form, composed only of the values defined by a type. Instead, partial operations can be typed by expliciting the partial nature of the function, e.g. by using option types or by simulating exceptions. However, future developments of FunBlocks may bring a more elaborate mechanism to deal with partial domains, perhaps based on customizable evaluation strategies [4] or dependent types.

4.3 Highlighting Optimal Evaluation Paths

As mentioned in Section 3.3, FunBlocks does not assume any particular reduction strategy, allowing any term to be rewritten at any execution step. This lets the user play with different strategies, to help them understand their implication in terms of semantics and form a more accurate mental representation of a program's execution. In this context, hinting the user about the consequences of a rule application could provide valuable insights.

Given a terminating program, the tool could indicate that choosing one rule over the other requires less further evaluation steps to reach a normal form. For instance, consider the following pair of rules, describing the semantics of a conditional expression:

$$\texttt{case}\ if(yes, \$a, \$b) \rightarrow \$a$$
$$\texttt{case}\ if(no, \$a, \$b) \rightarrow \$b$$

Assume the user wishes to reduce a term $if(yes, f(x), g(y))$, where both $f(x)$ and $g(y)$ denote expensive computations (i.e., long evaluation paths). Clearly, any step involved in the evaluation of the third sub-term (i.e., $g(y)$) is worthless in this situation.

Although termination is undecidable in general, there exist several techniques that can handle specific cases. For instance, if we can determine that the only rule matching a given term eventually leads the system to rewrite the same term (e.g., in the simplest case, if $f(\$x) \rightarrow g(f(\$x))$ is the only rule matching $f(\$x)$), then it is trivial to show that the program diverges. Another more general but weaker approach is to search for a normal form with a bound on the number of evaluation steps. Of course, as the choice for the bound's value is necessarily heuristic, this process is unable to establish negative answers definitively (i.e., the execution does not diverge). Nonetheless, highlighting (potentially) diverging rules could help the user shed light on the reason why their program does not terminate, in particular when non-determinism is involved.

5 MAKING FUNBLOCKS FUN

While FunBlocks can be used to illustrate theoretical concepts in a course, it is not suitable in a self-learning context. Unlike other equivalent educational tools, FunBlocks does not provide any guidance as to what the user is supposed to do, nor does it provide specific goals to reach. As it stands, it essentially presents itself as a compiler and interpreter for a hybrid visual and textual language.

One way to solve this issue would be to turn a program and its evaluation into a puzzle. This approach is reminiscent of the concept of *serious gaming* [12], which consists of exploiting elements of (video) game design for other purposes than pure entertainment. For instance, the user could be tasked to complete a set of rules so that the resulting program is able to evaluate its initial term to some specific final value, or to reach such final value in a maximum number of evaluation steps. Other, more complex puzzles would leverage the hinting system we discussed in the previous section and task the user to identify which rules cause a program to be non-confluent, divergeant and/or non-deterministic.

Another approach is to create logic games directly within FunBlocks that are played using "step-by-step" evaluations. For instance, we implemented a simplified variant of Conway's game of life. The state of the program represents a 1-dimensional cellular automaton, while the rewriting rules describes the lifecycle of a single cell. With this setup, the user can experiment with FunBlocks' semantics to create interesting patterns.

6 CONCLUSION

We presented FunBlocks, a visual and textual programming environment aimed at computer science education. Unlike most other pedagogical tools, FunBlocks adopts a declarative rather than imperative paradigm, based on term rewriting. Its language is equipped with a type system for algebraic data types which can be applied gradually on top of untyped programs. Both features place a great emphasis on data structures, providing novice users with a framework to reason not only about the way to solve a problem (i.e., "how"), but also about the nature of the problem (i.e., "what"). We argue that both notions are paramount to the development of so-called "computational thinking" skills. Despite its reliance on declarative programming, FunBlocks supports step-by-step evaluations, providing the user with a simple tool to debug their program. This feature can be also leveraged to play with different properties of computability, such as non-determinism, termination and confluence.

We explored different directions in which FunBlocks could be improved to provide the user with hints about the properties of their program. These hints would help the user develop a deeper understanding of their program's semantics and provide insights into the impact of certain computation patterns on computability properties and their decidability. Since FunBlocks relies on theoretically sound and well-known principles, namely term rewriting systems, all the ideas we have presented can be implemented on top of well-established techniques.

Our tool has yet to be evaluated in the context of the classroom, with actual students and novice learners, which will most certainly tilt future developments toward a specific direction. Conducting such evaluations can prove challenging, in particular to measure the efficacy of the tool at improving the user's understanding of programming, as evidence by the lack of a clear and convincing consensus in related literature. One important difficulty resides in the metric one can use to evaluate perennial and transferable knowledge [26]. Fortunately, the wealth of studies on programming education suggest numerous promising leads in that regard. We are currently working with the department of education in the state of Geneva, Switzerland, to develop computer science introductory courses partially based on FunBlocks.

FunBlocks is distributed under the MIT license. Sources are available on GitHub at https://github.com/kyouko-taiga/FunBlocks.

ACKNOWLEDGMENTS

The authors would like to thank the teachers and administrative staff involved in the preparation of the introduction of computer science at the high-school level in the canton of Geneva for their invaluable feedback on the design of FunBlocks. This project has been financially supported partly by the department of education of the state of Geneva.

REFERENCES

[1] Luca Aceto, Wan J. Fokkink, and Chris Verhoef. 2001. Structural Operational Semantics. In *Handbook of Process Algebra*, Jan A. Bergstra, Alban Ponse, and Scott A. Smolka (Eds.). North-Holland / Elsevier, Amsterdam, Netherlands, 197–292. https://doi.org/10.1016/b978-044482830-9/50021-7

[2] Michal Armoni, Orni Meerbaum-Salant, and Mordechai Ben-Ari. 2014. From Scratch to "Real" Programming. *ACM Transactions on Computing Education* 14, 4 (2014), 25:1–25:15. https://doi.org/10.1145/2677087

[3] Thomas Ball, Abhijith Chatra, Peli de Halleux, Steve Hodges, Michał Moskal, and Jacqueline Russell. 2019. Microsoft MakeCode: Embedded Programming for Education, in Blocks and TypeScript. In *SPLASH-E* (Athens, Greece) *(SPLASH-E 2019)*. ACM, New York, NY, USA, 7––12. https://doi.org/10.1145/3358711.3361630

[4] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. 2007. Tom: Piggybacking Rewriting on Java. In *Term Rewriting and Applications (RTA), (Lecture Notes in Computer Science, Vol. 4533)*, Franz Baader (Ed.). Springer, 36–47. https://doi.org/10.1007/978-3-540-73449-9_5

[5] David Bau, D. Anthony Bau, Matthew Dawson, and C. Sydney Pickens. 2015. Pencil code: block code for a text world. In *14th International Conference on Interaction Design and Children (IDC)*, Marina Umaschi Bers and Glenda Revelle (Eds.). ACM, New York, NY, USA, 445–448. https://doi.org/10.1145/2771839.2771875

[6] Dave Baum. 2013. *Dave Baum's definitive guide to LEGO Mindstorms.* APress, New York, NY, USA.

[7] Gérard Boudol. 1983. Computational semantics of terms rewriting systems. (1983).

[8] Wayne Citrin, Richard Hall, and Benjamin G. Zorn. 1995. Programming with Visual Expressions. In *11th International Symposium on Visual Languages.* IEEE Computer Society, Washington, D.C., USA, 294–301. https://doi.org/10.1109/VL.1995.520822

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1989. An Efficient Method of Computing Static Single Assignment Form. In *16th Symposium on Principles of Programming Languages (PoPL).* ACM Press, New York, NY, USA, 25–35. https://doi.org/10.1145/75277.75280

[10] O. Cugnon de Sevricourt and V. Tariel. 2011. Cameleon language Part 1: Processor. *CoRR* abs/1110.4802 (2011). arXiv:1110.4802 http://arxiv.org/abs/1110.4802

[11] Neil Fraser et al. 2013. Blockly: A visual programming editor. *URL: https://code. google. com/p/blockly* 42 (2013).

[12] David Gouveia, Duarte Lopes, and Carlos Vaz de Carvalho. 2011. Serious gaming for experiential learning. In *Frontiers in Education Conference (FIE).* IEEE Computer Society, 2. https://doi.org/10.1109/FIE.2011.6142778

[13] Shuchi Grover and Satabdi Basu. 2017. Measuring Student Learning in Introductory Block-Based Programming: Examining Misconceptions of Loops, Variables, and Boolean Logic. In *Technical Symposium on Computer Science Education*, Michael E. Caspersen, Stephen H. Edwards, Tiffany Barnes, and Daniel D. Garcia (Eds.). ACM, New York, NY, USA, 267–272. https://doi.org/10.1145/3017680.3017723

[14] John Hogg, Doug Lea, Alan Cameron Wills, Dennis de Champeaux, and Richard C. Holt. 2013. The Geneva Convention on the Treatment of Object Aliasing. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Lecture Notes in Computer Science, Vol. 7850. Springer, Berlin, Germany, 7–14. https://doi.org/10.1007/978-3-642-36946-9_2

[15] Michael Homer and James Noble. 2013. A tile-based editor for a textual programming language. In *1st Working Conference on Software Visualization (VISSOFT)*, Alexandru Telea, Andreas Kerren, and Andrian Marcus (Eds.). IEEE Computer Society, Washington, D.C., USA, 1–4. https://doi.org/10.1109/VISSOFT.2013.6650546

[16] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Transactions on Computing Education* 10, 4 (2010), 16:1–16:15. https://doi.org/10.1145/1868358.1868363

[17] John H. Maloney, Kylie A. Peppler, Yasmin B. Kafai, Mitchel Resnick, and Natalie Rusk. 2008. Programming by choice: urban youth learning programming with scratch. In *39th Technical Symposium on Computer Science Education (SIGCSE)*, J. D. Dougherty, Susan H. Rodger, Sue Fitzgerald, and Mark Guzdial (Eds.). ACM, New York, NY, USA, 367–371. https://doi.org/10.1145/1352135.1352260

[18] Dave Mason and Kruti Dave. 2017. Block-based versus flow-based programming for naive programmers. In *Blocks and Beyond Workshop (B&B).* IEEE Computer Society, Washington, D.C., USA, 25–28.

[19] Michael A. Miljanovic and Jeremy S. Bradbury. 2017. RoboBUG: A Serious Game for Learning Debugging Techniques. In *Conference on International Computing Education Research (ICER)*, Josh Tenenberg, Donald Chinn, Judy Sheard, and Lauri Malmi (Eds.). ACM, New York, NY, USA, 93–100. https://doi.org/10.1145/3105726.3106173

[20] Piotr Moczurad and Maciej Malawski. 2018. Visual-Textual Framework for Serverless Computation: A Luna Language Approach. In *International Conference on Utility and Cloud Computing Companion (UCC)*, Alan Sill and Josef Spillner (Eds.). IEEE Computer Society, Washington, D.C., USA, 169–174. https://doi.org/10.1109/UCC-Companion.2018.00052

[21] Étienne Payet. 2008. Loop detection in term rewriting using the eliminating unfoldings. *Theoretical Computer Science* 403, 2-3 (2008), 307–327. https://doi.org/10.1016/j.tcs.2008.05.013

[22] Matthew Poole. 2019. A block design for introductory functional programming in Haskell. In *Blocks and Beyond Workshop (B&B).* IEEE Computer Society, Washington, D.C., USA, 31–35.

[23] Thomas W. Price and Tiffany Barnes. 2015. Comparing Textual and Block Interfaces in a Novice Programming Environment. In *11th International Conference on International Computing Education Research, (ICER)*, Brian Dorn, Judy Sheard, and Quintin I. Cutts (Eds.). ACM, New York, NY, USA, 91–99. https://doi.org/10.1145/2787622.2787712

[24] Dimitri Racordon and Didier Buchs. 2018. *Démystifier les concepts informatiques par l'expérimentation.* Peter Lang, Bern, Switzerland, 219–233. https://doi.org/10.3726/b13411

[25] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *Summit on Advances in Programming Languages (SNAPL) (LIPIcs, Vol. 32)*, Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

[26] Allison Elliott Tew and Brian Dorn. 2013. The Case for Validated Tools in Computer Science Education Research. *Computer* 46, 9 (2013), 60–66. https://doi.org/10.1109/MC.2013.259

[27] Marie Vasek. 2012. Representing expressive types in blocks programming languages. (2012).

[28] David Weintrop and Uri Wilensky. 2015. Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs. In *qqth International Conference on International Computing Education Research (ICER)*, Brian Dorn, Judy Sheard, and Quintin I. Cutts (Eds.). ACM, New York, NY, USA, 101–110. https://doi.org/10.1145/2787622.2787721

[29] David Weintrop and Uri Wilensky. 2017. Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms. *ACM Transactions on Computing Education* 18, 1 (2017), 3:1–3:25. https://doi.org/10.1145/3089799

[30] Andreas Zeller. 2009. *Why Programs Fail - A Guide to Systematic Debugging, 2nd Edition.* Academic Press, Cambridge, MA, USA. http://store.elsevier.com/product.jsp?isbn=9780123745156&pagename=search

[31] Nikolaos C. Zygouris, Aikaterini Striftou, Antonios N. Dadaliaris, George I. Stamoulis, Apostolos C. Xenakis, and Denis Vavougios. 2017. The use of LEGO mindstorms in elementary schools. In *Global Engineering Education Conference (EDUCON).* IEEE Computer Society, Washington, D.C., USA, 514–516. https://doi.org/10.1109/EDUCON.2017.7942895