Instant playful access to serious programming for non-programmers with a visual functional programming language

Chide Groenouwe John-Jules Meyer

Alan Turing Institute Almere
Almere, the Netherlands
Universiteit Utrecht
Information and Computing Sciences
Utrecht, the Netherlands
{c.groenouwe|j.meyer}@ati-a.nl or {c.n.groenouwe|j.j.c.meyer}@uu.nl

Abstract

It would be highly beneficial if non-programmers could participate in the construction of serious applications. In this article we present aspects of the design of Marama, a visual modern functional programming language for this purpose. Marama's main target audience are non-programming, but moderately analytically trained academics. Functional programming languages are easy to visualise, powerful and known for their reliability. A distinctive pursued feature of Marama is that the visual functional programming constructs are to be entirely visual and self-explanatory, lowering the threshold for non-programmer participation. Concerning experimental validation, we focus on comprehensibility of a selection of designed prototype constructs, which is a necessary condition for non-programmer collaboration in program construction. These constructs were offered to people in an online survey. It turned out that participants ($N = 43$) were quite capable of grasping the workings of these constructs, with an average score of 80%. Limited to the main target audience ($N = 14$), the results were even better: 85%.

1. Introduction

This paper presents and validates parts of the design of Marama, a general purpose Visual Modern Functional Programming Language (VMFPL), intended to reduce the gap between programmers and non-programmers in creating serious, reliable, programs. An essential target audience of Marama is the following. In some organisations (such as ours), critical information systems are created in collaboration between analytically and academically trained, but non-programming, in-house experts in the field of application of the system, and programmers. Marama will allow these experts to participate in constructing systems up to an easily comprehensible layer, and as such speeding up system development. However, where possible it is designed to be accessible to a wider audience.

Modern Functional Programming Languages (MFPLs) are well suited for visual representation due to their transformative nature. Marama's design is based on visual "twins" of modern functional programming constructs using spatial metaphors rooted in common sense or inborn spatial intuition, making the language almost entirely self-explanatory. The implementation strategy is to write a translation to and from an existing mature modern functional programming language, leveraging the extensive amount of work done in the field of the latter. To the best of our knowledge, a language combining the mentioned features, scope and approaches does not yet exist. In this article, we experimentally validate one aspect of the design: the comprehensibility of the designed visual programming constructs for non-programmers. The experiment consists of offering a selection of constructs to testpersons ($N = 43$) in an online survey. Note that the experiment and its results can be read independently of the other design decisions taken within Marama. It turns out that non-programming academics were quite capable of grasping most visual programming constructs.

July 30, 2013

1.1 Overall goal

The overall goal of our research project is to bring a programming environment to non-programmers with the following properties. It should (1) have a very low threshold to start programming, (2) allow a seamless scale from very simple to highly advanced programming, depending on the level of the user, (3) allow users to assist in creating highly reliable programs, (4) allow users to assist in the production of highly optimised, high performance, programs (5) have the benefits of modern programming languages: such as advanced typing systems and (6) take a minimal amount of time to develop it.

2. Basic Approach

This section explains the basic approach, and associated design decisions, chosen to create a language to meet the overall goal. The requirements of the overall goal are printed with emphasised words.

2.1 Marriage between modern functional programming and data flow languages

A marriage between visual dataflow languages and MFPLs is an important ingredient in meeting the requirements of the overall goal of this study. In visual dataflow languages a program is regarded as a set of transformations on representations of data flowing through the system, each box representing a transformation. It is a simple basic concept with an unambiguous and computational meaning, which is easy to understand also by non-programmers. Therefore, it has a *low threshold* for usage by non-programmers. The large majority of dataflow languages, however, are not Turing Complete (e.g. Orange Canvas), which is in conflict with our goal to create a *serious* programming language.

In contrast to this, MFPLs are, as fully fledged programming languages, evidently Turing Complete. We define MFPLs as languages which at least support or contain (1) pure functional programming, (2) Algebraic Data Structures (ADSs), (3) lazy evaluation (4) pattern matching on ADSs, (5) higher order functions, (6) a static, strong, type system, which includes parametric, and adhoc, polymorphism, (6) type inferencing, (7) case-statements with guards and pattern matching and (8) currying of functions [1].

Examples of such languages are Haskell [2], Clean [3] and the functional part of Scala [4]. Their features make that they meet other important requirements of our overall goal. These include the following. First, with MFPLs you can create programs far more *reliably* than with many mainstream programming paradigms (of which language such as Java and C++ are part). This is caused by the fact it easier to verify (or even formally prove) their correctness [5, 6]. Second, their high expressivity leads to more concisely and sharply formulated programs, which in its turn also contributes to their easier verifiability. Third, there are several highly advanced compilers for MFPLs which create highly optimised code for the construction of *serious* programs.

MFPLs and dataflow languages can be easily integrated, merging the advantages of both, cf. [7, 8]. This is so, because MFPLs consist purely of side-effect free transformations of data [2]. Each transformation can be represented by a dataflow box. In a sense, functional languages are Turing complete dataflow languages. In contrast, it is impossible to represent programs in imperative languages (Java, etc.) as pure data flow structures. For example, control structures such as loops, absent in (the purely functional part of) MFPLs, cannot be represented as a dataflow.

2.2 Visual counterparts

An essential part of the basic approach is designing visual counterparts to all functional programming constructs. Each visual counterpart consists of a mix of adequate collective visual metaphors and spatial constructions governed by the laws of physics. The spatial constructions are chosen such that the physical rules which govern their dynamics, semantically align with the programming constructs they epitomise. Moreover, these rules are chosen such that they are obvious to a person with an average spatial intuition. This means we strive for a language in which the need for defining the semantics of constructions textually, or even externally, is minimised. (Sect. 6.3 contains a good example).

The programming constructs which are to be represented visually to have a full coverage of those found in MFPLs include: function application, function composition, function definition, guards, case-statements, recursive definitions, algebraic data structures (ADSs), pattern matching on ADSs, higher order functions, explicit typing including polymorphy, parametric polymorphy, currying and algebraic data types. Note that this list uses a slightly refined categorisation than in the earlier definition of MFPLs, which is more convenient for the design process of the visual constructs. Next to these constructs, the feedback provided by the type inference system should also have a visual counterpart.

2.3 Implementation

The implementation is realised by actually building a 2-way compiler between the visual language and an existing MFPL. In this way, we leverage 100s of humanyears worth of work in MFPL compiler and language design, cf. [7].

2.4 Minimisation of Textual Explanations

One of the most distinctive features in the design of Marama, is that visualisations are designed such that the need to explain them textually is minimised. This has the following advantages. It lowers the threshold to start collaborating in programming, because there is no need to work through piles of tutorials and textbooks. Memorising the meaning of constructs is minimised, which makes it easy to *continue* using the language also after longer interruptions. We may expect such interruptions to occur more frequently among non-programmers. An added benefit is that the language can more easily cross natural language barriers — users from anywhere in the world can start programming without having to wait for a translation of the documentation into their language (provided that the collective visual metaphors, of course, have been chosen adequately).

2.5 Visual and Textual View

Marama features a textual view and a visual view. The textual view is the target language mentioned in 2.3. The motivation is that the visual language is in no way intended to replace textual programming, which has its own benefits.

3. Envisioned Usage

The envisioned uses of the language include and exclude the following. (1) The language is not intended to be a replacement for textual programming. Textual programs have their own distinctive features and advantages. Rather, it is intended as a complement to textual languages, and as an alternative view on the program, cf. [7]. In our institute, for example, we develop multi-agent systems, and intend to use Marama primarily as a glue language, to connect these agents. The agents may (have) be(en) written in any language. (2) There is an inevitable trade off between expressivity and difficulty level of programming languages. Marama allows a program to be expressed in a multi layered structure, using simple functional constructs in the top layer (function application, function composition, which coincides with the expressivity of most dataflow languages), and harder constructs in lower levels. This allows user-friendliness from non-programmers to programmers, and gradations in between. Depending on the talent and motivation of the user, deeper layers of the program are accessible, up to the deepest, which brings Turing complete expressivity at one's fingertips. In this way non-programmers and programmers can meet each other halfway in the programming process, by collaborating in an overlapping range within the multi-layered structure. (3) A disadvantage of functional languages, in comparison with imperative languages, is that it is in general considered cognitively more difficult to write parts of programs dealing with changing structures, such as changes on a computer display (animations), or ongoing I/O. One may consider to use another language for these parts of the program. An example of a domain well-suited for functional programming is datamining, but one may also think of parts of programs, such as plugins for drawing programs and Web browsers. However, several MFPLs have been explicitly designed for dealing with changing structures, while still having the additional benefit of higher reliability than imperative languages. Therefore, usage for these problems is encouraged for those able to surmount the cognitive barrier.

4. Related work

Visual Haskell [1] designed by Reekie [7] has a similar approach: using a translation between an existing modern functional language

[1] Not to confuse with another project with the same name but a completely different goal.

(Haskell [2]) and a visual functional language. However, his focus is on “programmers not familiar with functional programming”, not on non-programmers. Diagrams are elegant, but cryptic and certainly not self-explanatory in the sense described in this article. They require textual explanations, such as case-statements. Another example is that he uses semantically agnostic trees as a base for different structures, in particular algebraic datastructures.

The Visual Functional Programming Environment of Kelso [8], is a proto-type environment to create and compile visual functional programs without being tied to a specific textual functional programming language. However, his approach differs from ours in the same way as Visual Haskell does.

5. Validation

To validate the effectiveness of our current version of visual programming constructs, this study first focusses on to what degree people understand these constructs (without using them to create programs). Understanding is an essential condition towards participation in program creation, albeit not a sufficient one. For this purpose, a selection of these constructs were offered to 43 testpersons, to test their understanding of them. This selection is the following: function application, function composition, function definition, guards, case-statements, recursive definitions, list-ADSs and pattern matching on list-ADSs with a fixed length.

The test consists of an online survey with the following relevant details. Each question may be introduced by a number of visual primers (see Sect. 6.1), or visual examples of the construction being used. Use of written language is minimised as much as possible. The questions are mostly questions in which the participant has to predict the outcome of a small visual program. Examples of questions can be found in Sect. 6. The survey additionally contains questions related to the background of the participant: education, degree of exactness of education(s), professions, age, sex and programming experience. Some of these questions are important to distinguish between programmers and non-programmers, and degrees in between.

The test persons were acquired, among other ways, by approaching arbitrary people on the street and in research facilities, and by word of mouth. Most were not employed by our institute to increase objectivity.

The complete survey can, for the time being, be accessed at: `http://survey.x50x.net/limesurvey/index.php`

6. Designed Visualisations

This section elaborates on some of the visual programming construct designs tested in the online survey, and aspects surrounding them.

6.1 Visual primers

During our study, we used *visual primers* to make understanding some visual programming constructs easier. Visual primers are introductory drawings, which contain analogies or examples of usage of the visual programming constructs to come. For example, a drawing with a person going through a sliding door was used to introduce guarded case-statements.

The notion of visual primers may sound slightly contradictory in the face of our goal to make the visual constructions self-explanatory as much as possible. However, this is mostly an apparent contradiction. First, some unclarities are due to the quality of the drawings. In future versions, drawings of a better quality will used, making many primers redundant (such as the sliding door primer). Second, even if primers continue to be important for some constructs, most of these constructs remain self-explanatory in the sense that the complete meaning is contained in them. The users

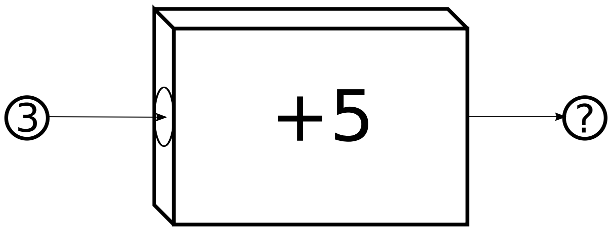

Figure 1. Simple function application

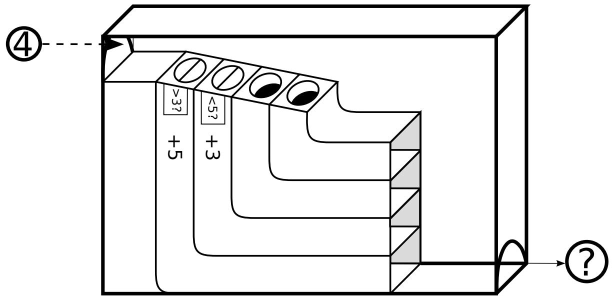

Figure 2. Guarded case statement

may simply require some hints to recognise elements in the picture as they were intended. This is something qualitatively different than for example Visual Haskell, in which many constructs are *defined* in terms of an external definition.

6.2 Function application

Fig. 1 shows function application. A ball which holds the value 3, goes through a box which adds 5 to it. The result is a ball which holds 8. This is immediately an example of a survey question: the question posed here is “Predict what will appear at the ?-mark.”.

6.3 Guarded case statement

Fig. 2 is a guarded case statement. In this case, a ball holding 4 comes in and starts to roll down. When the ball reaches the first door, it opens because it satisfies the guard ($4 > 3$). Then the ball falls through, and continues its way through a box which adds 5 to it, which results in a ball holding 9. The second door is never reached (although it would have opened if it were reached because $4 < 5$ also holds).

This design is a good example of our goal to capture the semantics of the constructions internally: the order in which the doors are tried is enforced by the mechanical dynamics of the ball, and understanding the latter is a matter of common sense. In contrast, case statements in Visual Haskell require an external definition.

6.4 Recursion

Fig. 3 shows a simple recursive definition of box b, integrated with an example of its application. In the example, the ball holding value 1 enters b, rolls past the first door (because the condition =0? is not fulfilled), then falls down the always open second hole. It goes through the -1 sub-box, which subtracts 1, so that a ball holding value 0 appears. Then the ball enters box b again, here elucidated with a magnification of a part of the picture (compare with [7]). The ball continues its way, also passing the identity box, which outputs the input unchanged.

6.5 Algebraic Data Structures

ADSs pose a challenge. We need a structure which is recursive in some way, and allows the same manipulations as ADSs in func-

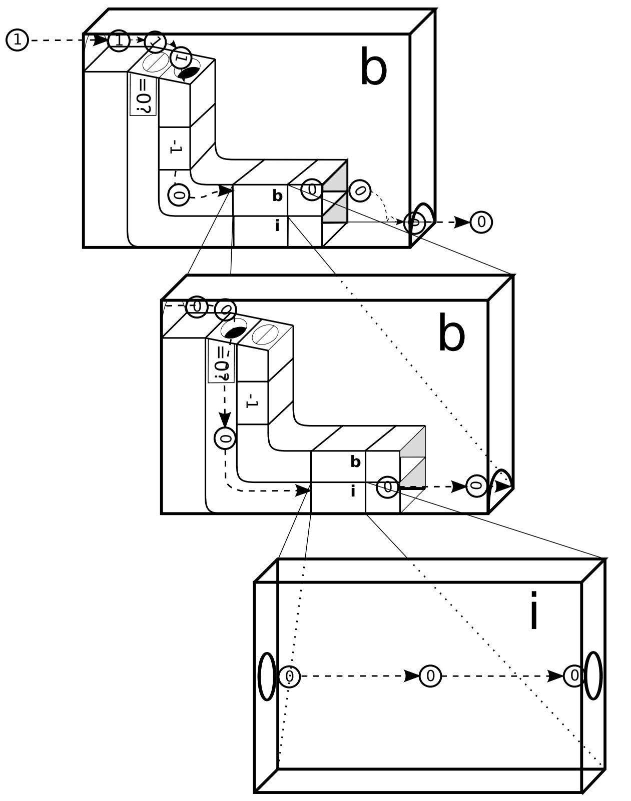

Figure 3. Simple recursion

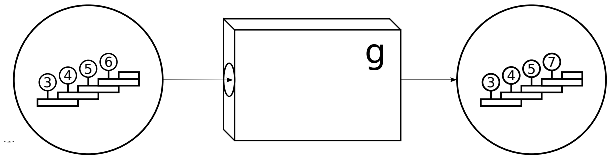

Figure 4. Example of List-Algebraic-Data-Structures (List-ADS)

tional languages, but then naturally enforced by its spatial properties.

In this stage of the design, the structure is simplified, because people only participate in a passive comprehensibility study. If they also create programs, some more restrictions must be imposed on ADSs. Visual designs for this purpose are already designed but are beyond scope of this paper.

Fig. 4 depicts list ADSs. A list is a physical construction which consist of “bricks” which are stacked onto each other. Each brick has two slots: (1) in this case one to hold an integer, which is a little stick which can hold the ball with the integer; (2) another one to hold the rest of the list. A special half-sized brick is reserved to represent “Nil” - the empty list.

6.6 Visual Pattern Matching

Fig. 5 shows visual pattern matching. The list-pattern is nothing but a list which visual structure is equal to that of a normal list, except for the omission of values on the balls. In this example, the first ball is taken from the list, then the function +1 is applied to it, and it is consequently put in as the first value of the output-list. The rest of the list, is transferred unchanged to the same position in the resulting list.

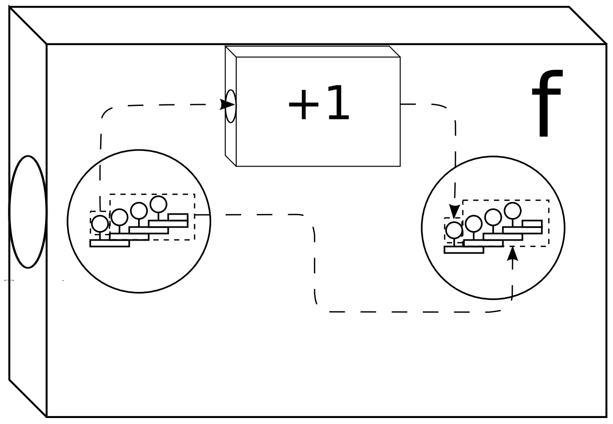

Figure 5. Visual pattern matching

7. Results

In total 43 people participated, and had all kinds of educational and professional backgrounds. Table 1 shows the average score of the total group, and subgroups of participants on different programming comprehension topics. On each topic a participant can score 0 (completely wrong) to 1 (perfect). “n level recursion” refers to the recursiondepth of the function application of a recursive function. For example, 1 means that a function called itself exactly once.

Table 1. Average Scores Participants. Scale is a ratio (0-1), unless indicated otherwise.

Programming experience (1-4)	Any	Any	Any	1	1 or 2
Highest educational level	Any	≥Bach.	≥Bach.	≥Bach.	≥Bach.
Exactness of highest edu. (1-5)	Any	< 5	5	Any	3 or 4
Sample size *N*	43	26	11	20	14
Function application	0.98	1	1	1	1
Guards	0.9	1	0.82	1	1
Guarded case	0.83	0.87	0.91	0.88	0.86
Function definition	0.95	0.65	0.95	0.98	1
Funct. compos. of nested defs	0.78	0.92	0.64	0.93	0.93
1 level recursion	0.65	0.58	0.91	0.5	0.71
2 level recursion	0.53	0.5	0.82	0.45	0.45
ADS introduction	0.79	0.85	0.82	0.8	0.79
ADS basic	0.81	0.88	0.82	0.85	0.93
Average score	0.8	0.81	0.85	0.82	0.85

7.1 Discussion

The most important overall observation, is that our main target audience, the subgroup of people who followed Bachelor’s degree education or higher, were analytically trained (exactness level 3 or 4), but had very limited prior programming experience (level 1 or 2) (N = 14), are quite capable of grasping the visual programming constructs, with an average score of 0.85. Notable problems were in the recursion questions, where we observe a drop to about 0.7 (1 level recursion) and 0.45 (2 level recursion). These are still promising numbers: most of these non-programming participants also seemed to have understood this concept using the wordless, visual designs.

The overall score (of people with any background) was quite high, a score of 0.8 (N = 43), and the group without any prior programming experience, who followed any Bachelor’s degree education or higher, performed even better, with an average score of 0.82 (N = 20).

Some remarkable deviations are that people with a Bachelor’s degree education or higher, in a very exact discipline, scored lower on the topics function composition and guards. Perhaps they were overconfident and were more prone to make simple calculation errors. As could be expected, however, they scored much better than others on the cognitively more challenging topic of recursion.

Qualitative observations, grouped by topic, include the following. (1) *Motivation*. Several participants indicated the survey having been "fun". A biology student said he was enthusiastic about the approach, and he had the feeling the language would give him access to a certain degree of programming, without explicitly having to study programming. A professional photographer indicated: "this is one of the most important research topics in this era!". A biologist recalled a computer game he likes to play, which indicates the playfulness of the designs (which may help lowering the threshold to start programming). Negative responses, which were in the minority and not inside the main target audience, included: "it is just like a school examination, I lost hope after some time". (2) *Design*: specifically for non-programmers, it was difficult to grasp the notion of recursion without an example. The recurring function was often not regarded as a self-reference, but as a new, other function. Adding an example execution, seemed to lead to major improvements. During a preliminary field experiment (before the survey was constructed), it turned out that the case-statement required a primer drawing (see 6.1) of a ball rolling down. Otherwise, most people interpreted the drawing as a ball "flying" over the guarded doors, which doesn't enforce any order in which the doors are tried.

8. Future Work

Future work will (or may) include the following. First, the existing designs will be improved to increase their comprehensibility, in particular recursion. For this purpose, the participants will be interviewed to discover the source of misinterpretations. Second, we will create and test designs which cover more of the remaining mentioned constructs (see Sect. 2.2), to wit general ADSs (not only lists of fixed length), general pattern matching on ADSs, higher order functions, explicit typing including polymorphy, parametric polymorphy and currying. Third, it may be quite effective to create animations to exemplify the working of a construct, as was also suggested by some of the participants.

Most importantly, we will extend the test beyond a comprehensibility study toward actual program (co-)creation by non-programmers, by further developing the prototype language Marama.

9. Conclusion

The main target audience of Marama, analytically trained academics, with minimal to no prior exposure to programming ($N = 14$), turned out to be capable of grasping the visual programming constructs quite well, with an average score of 0.85. More than half of them were capable of grasping the principle of the most difficult concept in the experiment: recursion.

The overall score (of people with any background) was quite high, a score of 0.8 ($N = 43$), and the group without any prior programming, who followed any Bachelor's degree education or higher, performed even better, with an average score of 0.82 ($N = 20$).

Acknowledgment

The authors would like to thank the participants who volunteered in the online survey. Without them this study would not have been possible.

References

[1] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Comput. Surv.*, vol. 21, no. 3, pp. 359–411, Sep. 1989. [Online]. Available: http://doi.acm.org/10.1145/72551.72554

[2] P. Hudak, S. Peyton Jones, and P. Wadler (editors), "Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2)," *ACM SIGPLAN Notices*, vol. 27, no. 5, May 1992.

[3] T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer, and M. J. Plasmeijer, "Clean: A language for functional graph writing," in *FPCA*, ser. Lecture Notes in Computer Science, G. Kahn, Ed., vol. 274. Springer, 1987, pp. 364–384.

[4] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, L. Spoon, E. Stenman, and M. Zenger, "An Overview of the Scala Programming Language (2. edition)," Tech. Rep., 2006.

[5] J. Hughes, "Why Functional Programming Matters," *Computer Journal*, vol. 32, no. 2, pp. 98–107, 1989.

[6] R. Bird, *Pearls of Functional Algorithm Design*. Cambridge University Press, 2010. [Online]. Available: http://www.cambridge.org/gb/knowledge/isbn/item5600469

[7] H. J. Reekie, "Visual haskell: A first attempt," Tech. Rep., 1994.

[8] J. Kelso, "A visual programming environment for functional languages," 2002.